Explanation

Manually persisting RDDs in Spark prevents them from being garbage collected.
This statement is incorrect, and thus the correct answer to the question. Spark's garbage collector will remove even persisted objects, albeit in an "LRU" fashion. LRU stands for least recently used. So, during a garbage collection run, the objects that were used the longest time ago will be garbage collected first.
See the linked StackOverflow post below for more information.

Serialized caching is a strategy to increase the performance of garbage collection.
This statement is correct. The more Java objects Spark needs to collect during garbage collection, the longer it takes. Storing a collection of many Java objects, such as a DataFrame with a complex schema, through serialization as a single byte array thus increases performance. This means that garbage collection takes less time on a serialized DataFrame than an unserialized DataFrame.

Optimizing garbage collection performance in Spark may limit caching ability.
This statement is correct. A full garbage collection run slows down a Spark application. When taking about "tuning" garbage collection, we mean reducing the amount or duration of these slowdowns.
A full garbage collection run is triggered when the Old generation of the Java heap space is almost full. (If you are unfamiliar with this concept, check out the link to the Garbage Collection Tuning docs below.) Thus, one measure to avoid triggering a garbage collection run is to prevent the Old generation share of the heap space to be almost full.
To achieve this, one may decrease its size. Objects with sizes greater than the Old generation space will then be discarded instead of cached (stored) in the space and helping it to be "almost full" . This will decrease the number of full garbage collection runs, increasing overall performance.
Inevitably, however, objects will need to be recomputed when they are needed. So, this mechanism only works when a Spark application needs to reuse cached data as little as possible.
Garbage collection information can be accessed in the Spark UI's stage detail view.
This statement is correct. The task table in the Spark UI's stage detail view has a "GC Time" column, indicating the garbage collection time needed per task.

In Spark, using the G1 garbage collector is an alternative to using the

default Parallel garbage collector.
This statement is correct. The G1 garbage collector, also known as garbage first garbage collector, is an alternative to the default Parallel garbage collector. While the default Parallel garbage collector divides the heap into a few static regions, the G1 garbage collector divides the heap into many small regions that are created dynamically. The G1 garbage collector has certain advantages over the Parallel garbage collector which improve performance particularly for Spark workloads that require high throughput and low latency.
The G1 garbage collector is not enabled by default, and you need to explicitly pass an argument to Spark to enable it. For more information about the two garbage collectors, check out the Databricks article linked below.

Hierarchy

Stages with narrow dependencies can be grouped into one task.
Wrong, tasks with narrow dependencies can be grouped into one stage.

Tasks with wide dependencies can be grouped into one stage.
Wrong, since a wide transformation causes a shuffle which always marks the boundary of a stage. So, you cannot bundle multiple tasks that have wide dependencies into a stage.

Deploy Mode

cluster	In **cluster** mode, the driver runs on one of the worker nodes, and this node shows as a driver on the Spark Web UI of your application. cluster mode is used to run production jobs.
client	In **client** mode, the driver runs locally from where you are submitting your application using spark-submit command. client mode is majorly used for interactive and debugging purposes. Note that in client mode only the driver runs locally and all tasks run on cluster worker nodes.

In client mode, the cluster manager runs on the same host as the driver, while in cluster mode, the cluster manager runs on a separate node.
No. In both modes, the cluster manager is typically on a separate node – not on the same host as the driver. It only runs on the same host as the driver in local execution mode.

Dynamic partition pruning

Ref - https://dzone.com/articles/dynamic-partition-pruning-in-spark-30

is intended to skip over the data you do not need in the results of a query.
Correct - Dynamic partition pruning provides an efficient way to selectively read data from files by skipping data that is irrelevant for the query. For example, if a query asks to consider only rows which have numbers >12 in column purchases via a filter, Spark would only read the rows that match this criteria from the underlying files. This method works in an optimal way if the purchases data is in a nonpartitioned table and the data to be filtered is partitioned.

Dynamic partition pruning reoptimizes query plans based on runtime statistics collected during query execution.
No – this is what adaptive query execution does, but not dynamic partition pruning.

Dynamic partition pruning reoptimizes physical plans based on data types and broadcast variables.
It is true that dynamic partition pruning works in joins using broadcast variables. This actually happens in both the logical optimization and the physical planning stage. However, data types do not play a role for the reoptimization.

Modes

In client mode, the cluster manager runs on the same host as the driver, while in cluster mode, the cluster manager runs on a separate node.
No. In both modes, the cluster manager is typically on a separate node – not on the same host as the driver. It only runs on the same host as the driver in local execution mode.

Difference of Dataset and DataFRame

DataFrames
DataFrames gives a schema view of data basically, it is an abstraction. In dataframes, view of data is organized as columns with column name and types info. In addition, we can say data in dataframe is as same as the table in relational database.
As similar as RDD, execution in dataframe too is lazy triggered. Moreover, to

allow efficient processing datasets is structure as a distributed collection of data. Spark also uses catalyst optimizer along with dataframes.
DataSets
In Spark, datasets are an extension of dataframes. Basically, it earns two different APIs characteristics, such as strongly typed and untyped. Datasets are by default a collection of strongly typed JVM objects, unlike dataframes. Moreover, it uses Spark's Catalyst optimizer. For exposing expressions & data field to a query planner.
Now we will see the difference in both based on certain features:

Spark Release
DataFrame- In Spark 1.3 Release, dataframes are introduced.
whereas,
DataSets- In Spark 1.6 Release, datasets are introduced.
Data Formats
DataFrame- Dataframes organizes the data in the named column. Basically, dataframes can efficiently process unstructured and structured data. Also, allows the Spark to manage schema.
whereas,
DataSets- As similar as dataframes, it also efficiently processes unstructured and structured data. Also, represents data in the form of a collection of row object or JVM objects of row. Through encoders, is represented in tabular forms.

Data Representation
DataFrame- In dataframe data is organized into named columns. Basically, it is as same as a table in a relational database.
whereas,
DataSets- As we know, it is an extension of dataframe API, which provides the functionality of type-safe, object-oriented programming interface of the RDD API. Also, performance benefits of the Catalyst query optimizer.

Compile-time type safety
DataFrame- There is a case if we try to access the column which is not on the table. Then, dataframe APIs does not support compile-time error.
whereas,
DataSets- Datasets offers compile-time type safety.

Data Sources API
DataFrame- It allows data processing in different formats, for example, AVRO, CSV, JSON, and storage system HDFS, HIVE tables, MySQL.
whereas,
DataSets- It also supports data from different sources.

Immutability and Interoperability
DataFrame- Once transforming into dataframe, we cannot regenerate a domain object.

whereas,
DataSets- Datasets overcomes this drawback of dataframe to regenerate the RDD from dataframe. It also allows us to convert our existing RDD and dataframes into datasets.

Memory Management Overview

Memory usage in Spark largely falls under one of two categories: execution and storage. Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster. In Spark, execution and storage share a unified region (M). When no execution memory is used, storage can acquire all the available memory and vice versa. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R). In other words, R describes a subregion within M where cached blocks are never evicted. Storage may not evict execution due to complexities in implementation.
This design ensures several desirable properties. First, applications that do not use caching can use the entire space for execution, obviating unnecessary disk spills. Second, applications that do use caching can reserve a minimum storage space (R) where their data blocks are immune to being evicted. Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.
Although there are two relevant configurations, the typical user should not need to adjust them as the default values are applicable to most workloads:

- spark.memory.fraction expresses the size of M as a fraction of the (JVM heap space - 300MiB) (default 0.6). The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records.
- spark.memory.storageFraction expresses the size of R as a fraction of M (default 0.5). R is the storage space within M where cached blocks immune to being evicted by execution.

The value of spark.memory.fraction should be set in order to fit this amount of heap space comfortably within the JVM's old or "tenured" generation. See the discussion of advanced GC tuning below for details.

spark Config

Increase values for the properties spark.dynamicAllocation.maxExecutors, spark.default.parallelism, and spark.sql.shuffle.partitions
The property spark.dynamicAllocation.maxExecutors is only in effect if dynamic allocation is enabled, using the spark.dynamicAllocation.enabled property. It is disabled by default. Dynamic allocation can be useful when running multiple applications on the same cluster in parallel. However, in this case there is only a single application running on the cluster, so enabling dynamic allocation would

not yield a performance benefit.

Adaptive Query Execution

Adaptive Query Execution is enabled in Spark by default.
No, Adaptive Query Execution is disabled in Spark needs to be enabled through the spark.sql.adaptive.enabled property.

Dynamically injecting scan filters, are part of Adaptive Query Execution.
Dynamically injecting scan filters for join operations to limit the amount of data to be considered in a query is part of Dynamic Partition Pruning and not of Adaptive Query Execution.

Spark will only perform the broadcast operation if this behavior has been enabled on the Spark cluster.
No, it is enabled by default, since the spark.sql.autoBroadcastJoinThreshold property is set to 10 MB by default. If that property would be set to -1, then broadcast joining would be disabled.

Shuffle

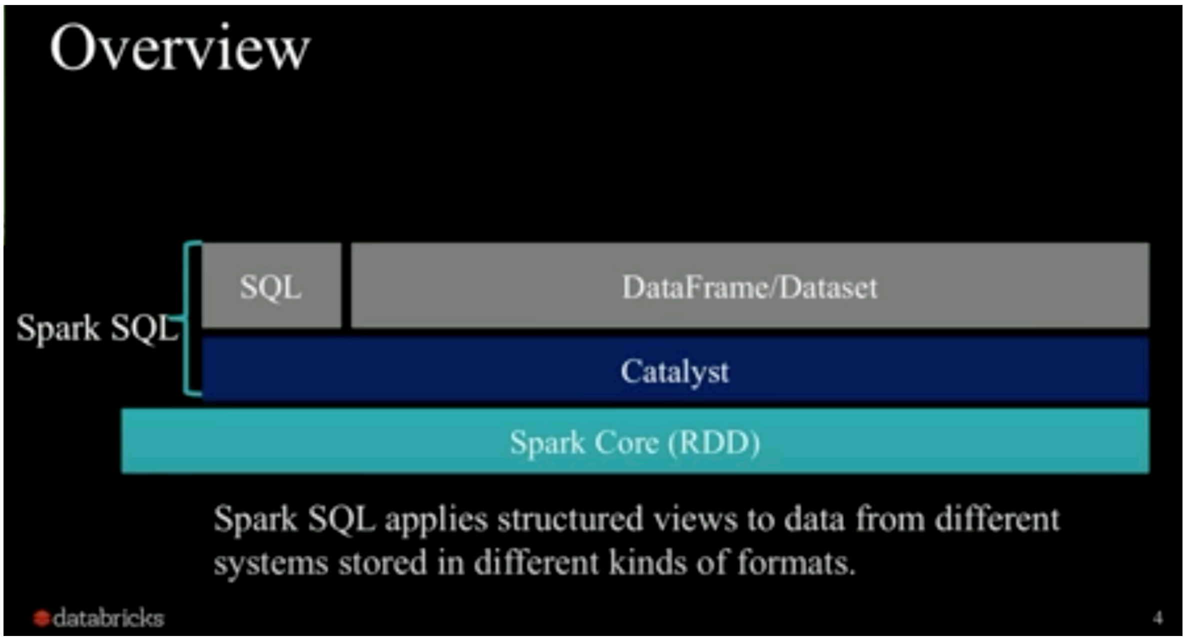
A shuffle is a process that compares data between partitions.
This is correct. During a shuffle, data is compared between partitions because shuffling includes the process of sorting. For sorting, data need to be compared. Since per definition, more than one partition is involved in a shuffle, it can be said that data is compared across partitions. You can read more about the technical details of sorting in the blog post linked below.

A shuffle is a Spark operation that results from DataFrame.coalesce().
No. DataFrame.coalesce() does not result in a shuffle.

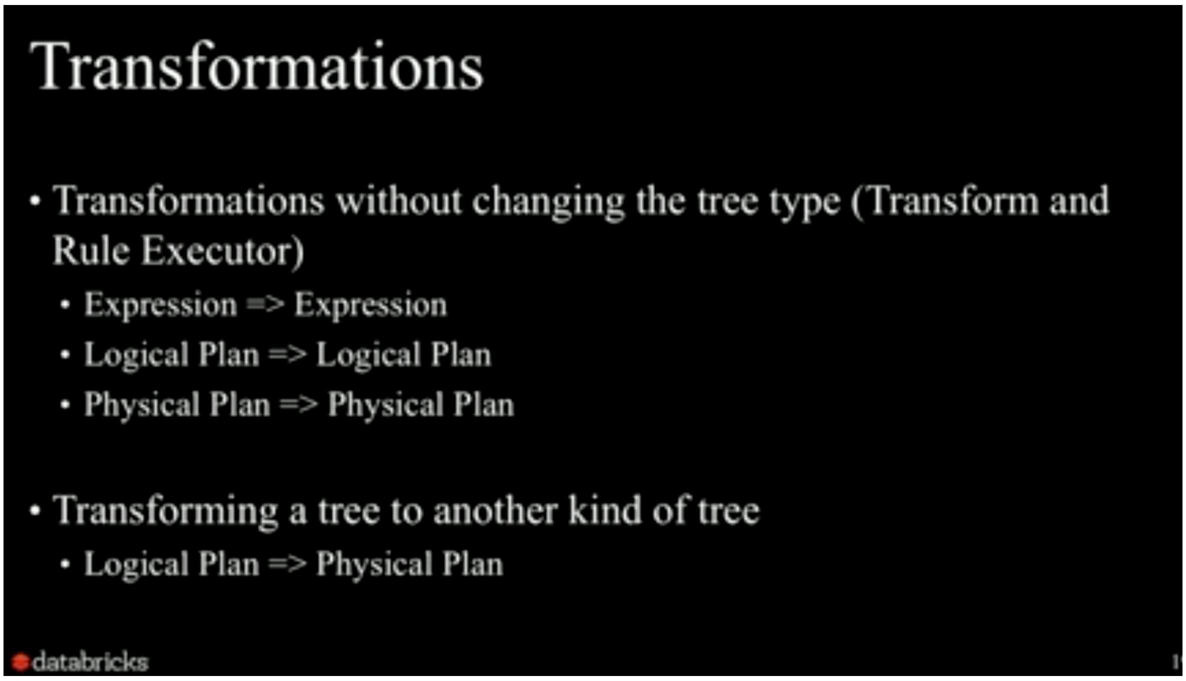
A shuffle is a process that allocates partitions to executors.
This is incorrect. A shuffle is not allocating partitions to executors, this happens in a different process in Spark.

A shuffle is a process that is executed during a broadcast hash join.
No, broadcast hash joins avoid shuffles and yield performance benefits if at least one of the two tables is small in size (<= 10 MB by default). Broadcast hash joins can avoid shuffles because instead of exchanging partitions between executors, they broadcast a small table to all executors that then perform the rest of the join operation locally.

A shuffle is a process that compares data across executors.
No, in a shuffle, data is compared across partitions, and not executors.


Overview

Spark SQL applies structured views to data from different systems stored in different kinds of formats.

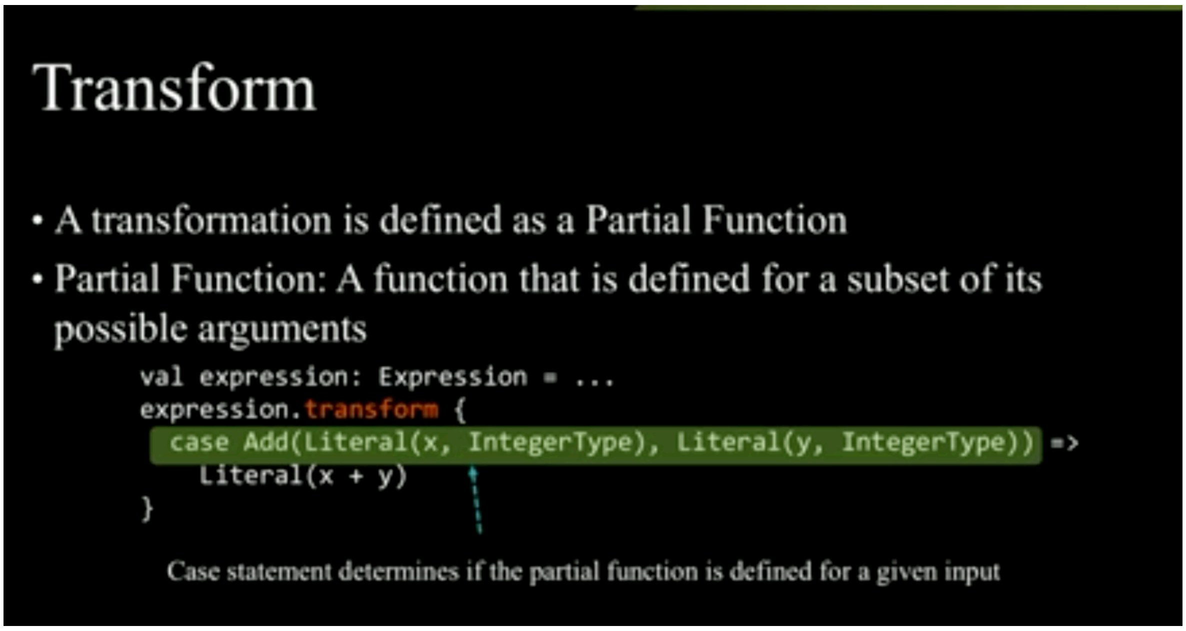

Transformations

- Transformations without changing the tree type (Transform and Rule Executor)
  - Expression => Expression
  - Logical Plan => Logical Plan
  - Physical Plan => Physical Plan
- Transforming a tree to another kind of tree
  - Logical Plan => Physical Plan

databricks

Transformations in pyspark like select,groupBy, orderBy,forEach etc are lazily evaluated and return DataFrame, Dataset, or RDD only when it is triggered by an action.

where actions are computed at the same moment of time.

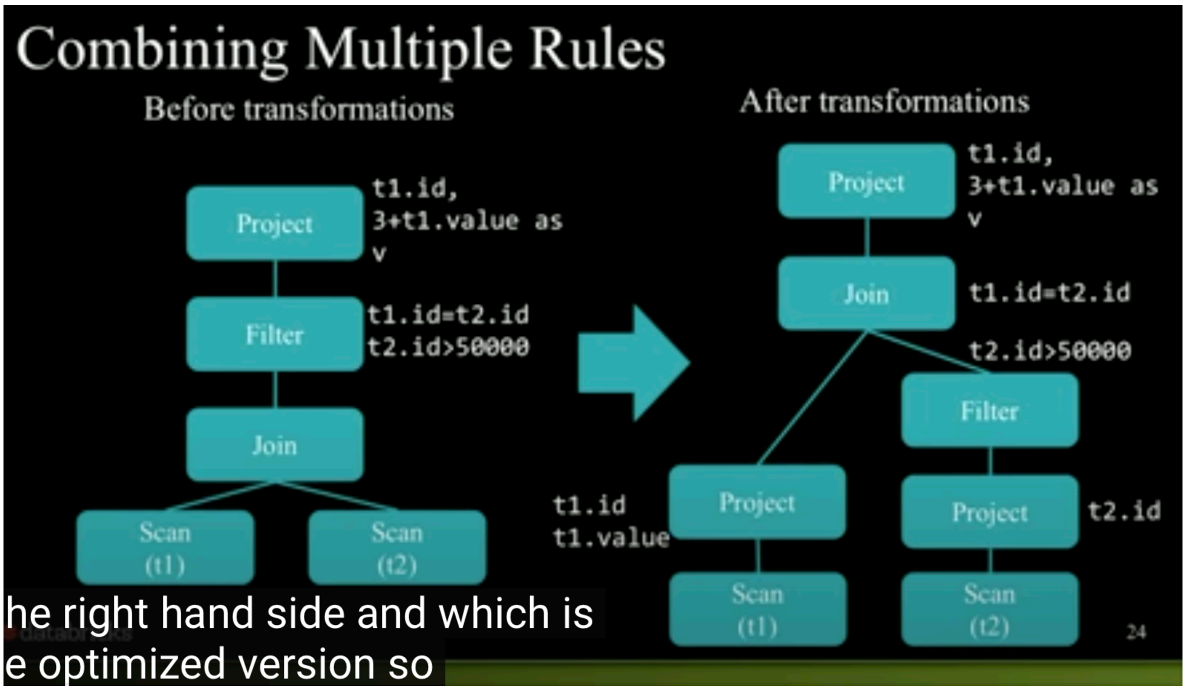


Transform

- A transformation is defined as a Partial Function
- Partial Function: A function that is defined for a subset of its possible arguments

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

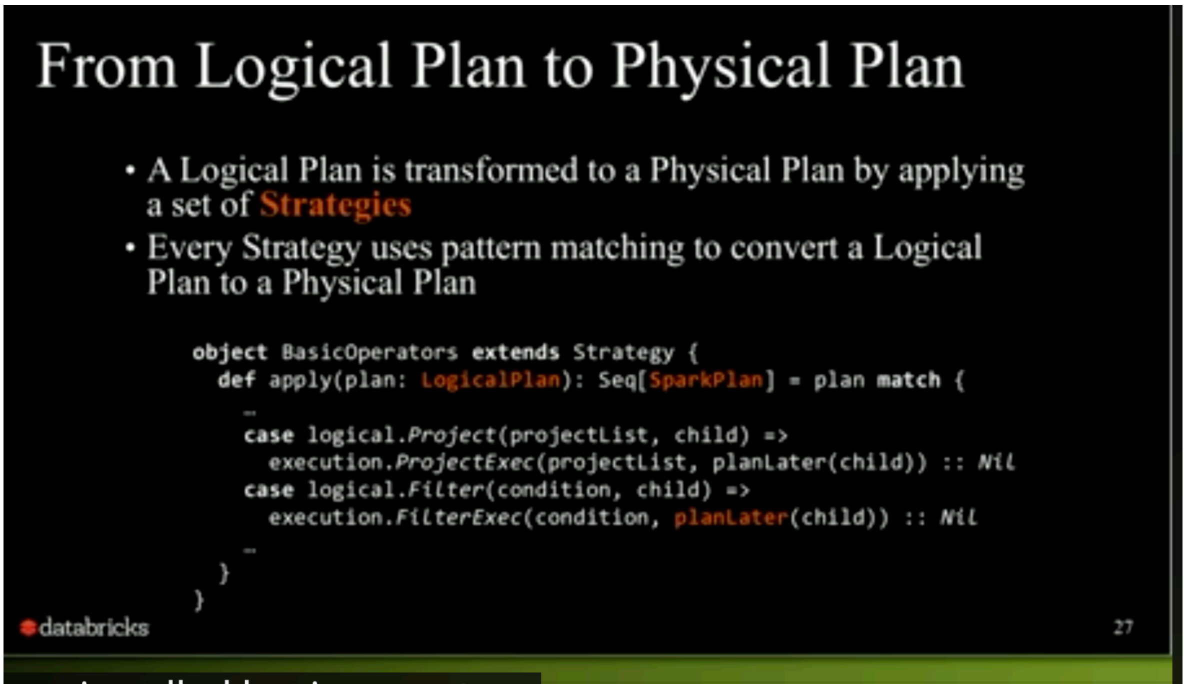
Case statement determines if the partial function is defined for a given input



Combining Multiple Rules

he right hand side and which is
e optimized version so

in above, **Predicate pushdown is a feature which leverages lazy evaluation.**

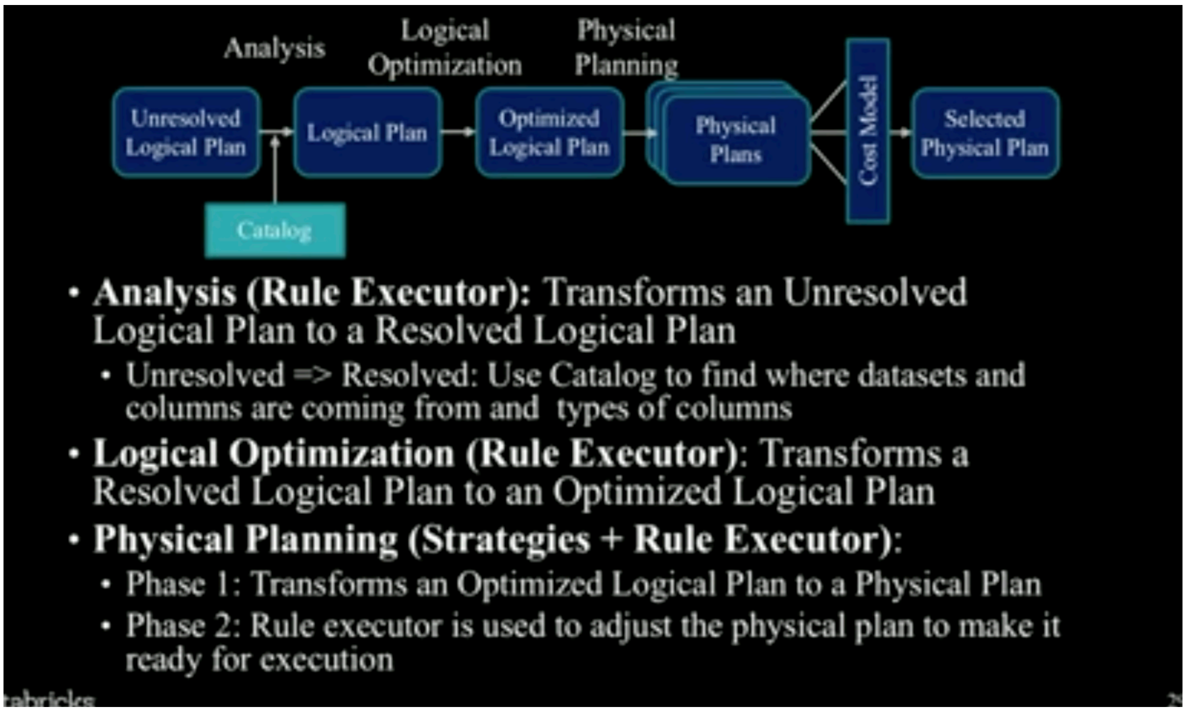


From Logical Plan to Physical Plan

- A Logical Plan is transformed to a Physical Plan by applying a set of Strategies
- Every Strategy uses pattern matching to convert a Logical Plan to a Physical Plan

```
object BasicOperators extends Strategy {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    …
    case logical.Project(projectList, child) =>
      execution.ProjectExec(projectList, planLater(child)) :: Nil
    case logical.Filter(condition, child) =>
      execution.FilterExec(condition, planLater(child)) :: Nil
    …
  }
}
```

databricks 27



- **Analysis (Rule Executor):** Transforms an Unresolved Logical Plan to a Resolved Logical Plan
  - Unresolved => Resolved: Use Catalog to find where datasets and columns are coming from and types of columns
- **Logical Optimization (Rule Executor):** Transforms a Resolved Logical Plan to an Optimized Logical Plan
- **Physical Planning (Strategies + Rule Executor):**
  - Phase 1: Transforms an Optimized Logical Plan to a Physical Plan
  - Phase 2: Rule executor is used to adjust the physical plan to make it ready for execution

From many physical model, the cost analysis is done. The final physical paln is then selected with minimum cost.

Explanation
The executed physical plan depends on a cost optimization from a previous stage.
Correct! Spark considers multiple physical plans on which it performs a cost analysis and selects the final physical plan in accordance with the lowest-cost outcome of that analysis. That final physical plan is then executed by Spark.

Spark uses the catalog to resolve the optimized logical plan.
No. Spark uses the catalog to resolve the unresolved logical plan, but not the optimized logical plan. Once the unresolved logical plan is resolved, it is then optimized using the Catalyst Optimizer. The optimized logical plan is the input for physical planning.

The catalog assigns specific resources to the physical plan.
No. The catalog stores metadata, such as a list of names of columns, data types, functions, and databases. Spark consults the catalog for resolving the references in a logical plan at the beginning of the conversion of the query into an execution plan. The result is then an optimized logical plan.

Depending on whether DataFrame API or SQL API are used, the physical plan may differ.
Wrong – the physical plan is independent of which API was used. And this is one of the great strengths of Spark!

The catalog assigns specific resources to the optimized memory plan.
There is no specific "memory plan" on the journey of a Spark computation.

Accumulators

Accumulators - https://sparkbyexamples.com/pyspark/pyspark-accumulator-with-example/#:~:text=What%20is%20PySpark%20Accumulator%3F,automatically%20to%20the%20driver%20program.

A shared variable that can be accumulated, i.e., has a commutative and associative “add” operation. Worker tasks on a Spark cluster can add values to an Accumulator with the *+=* operator, but only the driver program is allowed to access its value, using *value*. Updates from the workers get propagated automatically to the driver program.

While **SparkContext** supports accumulators for primitive data types like **int** and **float**, users can also define accumulators for custom types by providing a custom **AccumulatorParam** object.

Datetime functions related to convert StringType to/from DateType or TimestampType. For example, unix_timestamp, date_format, to_unix_timestamp, from_unixtime, to_date, to_timestamp, from_utc_timestamp, to_utc_timestamp.

# We can convert Unix Timestamp to regular date or timestamp and vice versa.

# We can use unix_timestamp to convert regular date or timestamp to a unix timestamp value.
#		For example unix_timestamp(lit("2019-11-19 00:00:00"))

# We can use from_unixtime to convert unix timestamp to regular date or timestamp.
#		For example from_unixtime(lit(1574101800))