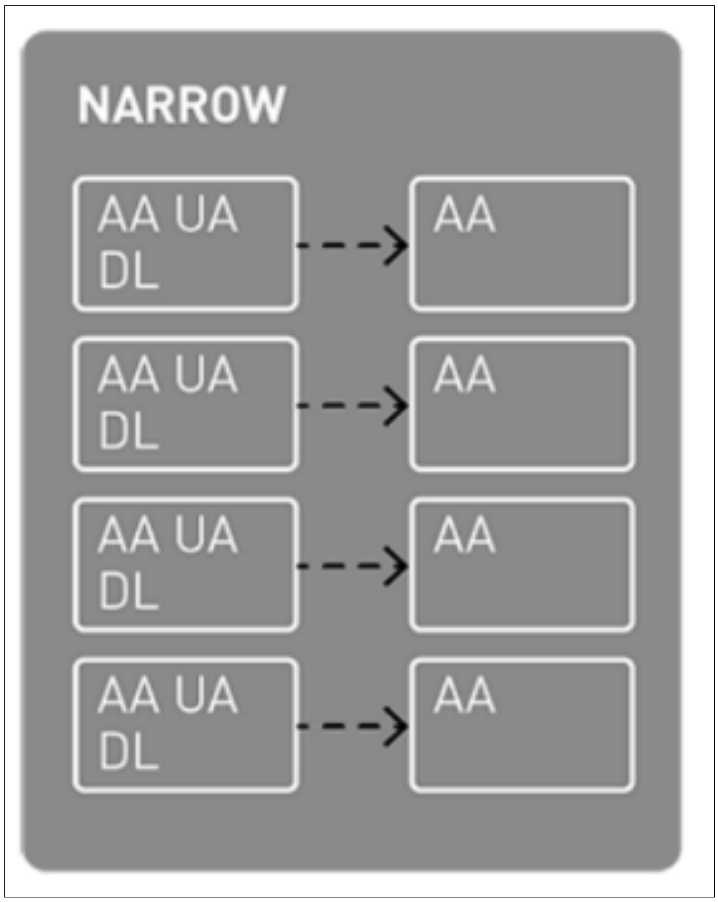
Reading: Common Transformations and Optimization Techniques in Spark

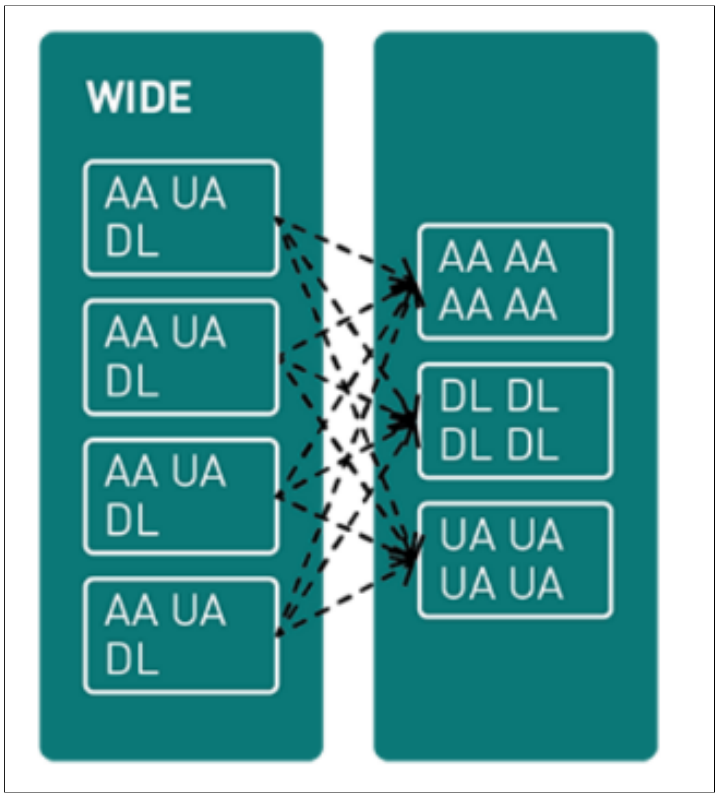
Estimated time needed: 30 minutes

When you're working with PySpark DataFrames for data processing, it's important to know about the two types of transformations: narrow and wide. Narrow transformations in Spark work within partitions without shuffling data between them. They're applied locally to each partition, avoiding data exchange. On the other hand, wide transformations in Spark involve redistributing and shuffling data between partitions, often leading to more resource-intensive and complex operations.

To understand this concept better, let's take a look at the following illustration.



Within narrow transformations, data is transferred without executing data shuffling operations.



Wide transformations involve the shuffling of data across partitions.

Examples of narrow transformations

Narrow transformations can be compared to performing straightforward operations on distinct data sets. Consider having various types of data in separate containers. You can perform actions on each data container or shift data between containers independently without requiring interaction or transfer. Examples of narrow transformations include modifying individual pieces of data, selecting specific items, or combining two data containers.

1. **Map:** Applying a function to each element in the data set.

```
from pyspark import SparkContext
sc = SparkContext("local", "MapExample")
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
mapped_rdd = rdd.map(lambda x: x * 2)
mapped_rdd.collect() # Output: [2, 4, 6, 8, 10]
```

2. **Filter:** Selecting elements based on a specified condition.

```
from pyspark import SparkContext
sc = SparkContext("local", "FilterExample")
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
filtered_rdd.collect() # Output: [2, 4]
```

3. Union: Combining two data sets with the same schema.

```
from pyspark import SparkContext
sc = SparkContext("local", "UnionExample")
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([4, 5, 6])
union_rdd = rdd1.union(rdd2)
union_rdd.collect() # Output: [1, 2, 3, 4, 5, 6]
```

Examples of wide transformations

Wide transformations can be compared to tasks accomplished with teamwork and where information is needed from different groups to conclude. Imagine you have a group of friends, each with a puzzle piece. In order to put the puzzle together, you might need to trade pieces between your friends to make everything fit. These kinds of tasks are a good example of wide transformation. Such tasks can be a little more complicated because everyone needs to collaborate and move pieces around.

1. **GroupBy:** Aggregating data based on a specific key.

```
from pyspark import SparkContext
sc = SparkContext("local", "GroupByExample")
data = [("apple", 2), ("banana", 3), ("apple", 5), ("banana", 1)]
rdd = sc.parallelize(data)
grouped_rdd = rdd.groupBy(lambda x: x[0])
sum_rdd = grouped_rdd.mapValues(lambda values: sum([v[1] for v in values]))
sum_rdd.collect() # Output: [('apple', 7), ('banana', 4)]
```

2. **Join:** Combining two data sets based on a common key.

```
from pyspark import SparkContext
sc = SparkContext("local", "JoinExample")
rdd1 = sc.parallelize([("apple", 2), ("banana", 3)])
rdd2 = sc.parallelize([("apple", 5), ("banana", 1)])
joined_rdd = rdd1.join(rdd2)
joined_rdd.collect() # Output: [('apple', (2, 5)), ('banana', (3, 1))]
```

3. Sort: Rearranging data based on a specific criterion.

```
from pyspark import SparkContext
sc = SparkContext("local", "SortExample")
data = [4, 2, 1, 3, 5]
rdd = sc.parallelize(data)
sorted_rdd = rdd.sortBy(lambda x: x, ascending=True)
sorted_rdd.collect() # Output: [1, 2, 3, 4, 5]
```

Wide transformations are similar to reshuffling and redistributing data between different groups. Imagine having data sets that you want to combine or organize in a new way. However, this task is not as straightforward as working with just one data set. You need to coordinate and move data between these sets, which involves more complexity. For example, merging two data sets based on a common attribute requires rearranging the data between them, making it a wide transformation in data engineering.

PySpark DataFrame: Rule-based common transformations

The DataFrame API in PySpark offers various transformations based on predefined rules. These transformations are designed to improve how queries are executed and boost overall performance. Let's take a look at some common rule-based transformations.

1. **Predicate pushdown:** Pushing filtering conditions closer to the data source before processing to minimize data movement.

2. **Constant folding:** Evaluating constant expressions during query compilation to reduce computation during runtime.

3. **Column pruning:** Eliminating unnecessary columns from the query plan to enhance processing efficiency.

4. **Join reordering:** Rearranging join operations to minimize the intermediate data size and enhance the join performance.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
# Create a Spark session
spark = SparkSession.builder.appName("RuleBasedTransformations").getOrCreate()
# Sample input data for DataFrame 1
data1 = [
    ("Alice", 25, "F"),
    ("Bob", 30, "M"),
    ("Charlie", 22, "M"),
```

```
    ("Diana", 28, "F")
]
# Sample input data for DataFrame 2
data2 = [
    ("Alice", "New York"),
    ("Bob", "San Francisco"),
    ("Charlie", "Los Angeles"),
    ("Eve", "Chicago")
]
# Create DataFrames
columns1 = ["name", "age", "gender"]
df1 = spark.createDataFrame(data1, columns1)
columns2 = ["name", "city"]
df2 = spark.createDataFrame(data2, columns2)
# Applying Predicate Pushdown (Filtering)
filtered_df = df1.filter(col("age") > 25)
# Applying Constant Folding
folded_df = filtered_df.select(col("name"), col("age") + 2)
# Applying Column Pruning
pruned_df = folded_df.select(col("name"))
# Join Reordering
reordered_join = df1.join(df2, on="name")
# Show the final results
print("Filtered DataFrame:")
filtered_df.show()
print("Folded DataFrame:")
folded_df.show()
print("Pruned DataFrame:")
pruned_df.show()
print("Reordered Join DataFrame:")
reordered_join.show()
# Stop the Spark session
spark.stop()
```

Optimization techniques used in Spark SQL

1. **Predicate pushdown:** Apply a filter to DataFrame "df1" to only select rows where the "age" column is greater than 25.
2. **Constant folding:** Perform an arithmetic operation on the "age" column in the folded_df, adding a constant value of 2.
3. **Column pruning:** Select only the "name" column in the pruned_df, eliminating unnecessary columns from the query plan.
4. **Join reordering:** Perform a join between df1 and df2 on the "name" column, allowing Spark to potentially reorder the join for better performance.

Cost-Based optimization techniques in Spark

Spark employs cost-based optimization techniques to enhance the efficiency of query execution. These methods involve estimating and analyzing the costs associated with queries, leading to more informed decisions that result in improved performance.

1. **Adaptive query execution:** Dynamically adjusts the query plan during execution based on runtime statistics to optimize performance.
2. **Cost-based join reordering:** Optimizes join order based on estimated costs of different join paths.
3. **Broadcast hash join:** Optimizes small-table joins by broadcasting one table to all nodes, reducing data shuffling.
4. **Shuffle partitioning and memory management:** Efficiently manages data shuffling during operations like groupBy and aggregation and optimizes memory usage.

By utilizing these methods, Spark endeavors to deliver efficient and scalable data processing capabilities. It is essential to grasp the effective application of these transformations and optimizations to attain the best possible query performance and optimal utilization of system resources.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
# Create a Spark session
spark = SparkSession.builder.appName("CostBasedOptimization").getOrCreate()
# Sample input data for DataFrame 1
data1 = [
    ("Alice", 25),
    ("Bob", 30),
    ("Charlie", 22),
    ("Diana", 28)
]
# Sample input data for DataFrame 2
data2 = [
    ("Alice", "New York"),
    ("Bob", "San Francisco"),
    ("Charlie", "Los Angeles"),
    ("Eve", "Chicago")
]
# Create DataFrames
columns1 = ["name", "age"]
df1 = spark.createDataFrame(data1, columns1)
columns2 = ["name", "city"]
df2 = spark.createDataFrame(data2, columns2)
# Enable adaptive query execution
spark.conf.set("spark.sql.adaptive.enabled", "true")
# Applying Adaptive Query Execution (Runtime adaptive optimization)
optimized_join = df1.join(df2, on="name")
# Show the optimized join result
print("Optimized Join DataFrame:")
optimized_join.show()
# Stop the Spark session
spark.stop()
```

In this example, we created two DataFrames (**df1** and **df2**) with sample input data. Then, we enabled the adaptive query execution feature by setting the configuration parameter **"spark.sql.adaptive.enabled"** to **"true"**. Adaptive Query Execution allows Spark to adjust the query plan during execution based on runtime statistics.

The code performs a join between **df1** and **df2** on the "name" column. Spark's adaptive query execution dynamically adjusts the query plan based on runtime statistics, which can result in improved performance.

Author(s)

Raghul Ramesh