PySpark – Dynamic Partition Pruning (DPP)

Introduction to Dynamic Partition Pruning (DPP)

Dynamic Partition Pruning (DPP) is an optimization technique used in PySpark that reduces the amount of data read during query execution. When dealing with large partitioned tables, scanning all partitions can be inefficient. DPP dynamically reduces the partitions being scanned based on filter conditions, improving query performance.

How Dynamic Partition Pruning Works

DPP allows Spark to skip unnecessary partitions at runtime based on the data distribution in other tables. This typically happens during a join operation between a large partitioned fact table and a smaller dimension table. By pushing down the filter conditions from the dimension table to the fact table, DPP avoids reading the partitions that do not contain relevant data.

Example Scenario

Let's assume we have two tables:

- **Orders Table**: Partitioned by `order_date`
- **Customers Table**: Not partitioned

We want to join these two tables based on `customer_id` and filter on `customer_region`. With DPP, PySpark will identify the relevant partitions of the `orders` table based on the regions in the `customers` table, and it will prune the partitions that do not satisfy the join conditions.

Steps to Enable Dynamic Partition Pruning

Dynamic Partition Pruning is enabled by default in Spark 3.0 and above. To explicitly control DPP, you can use the following Spark configurations:

```
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")
```

This configuration ensures that DPP is used in queries that are eligible for this optimization.

PySpark Example with Dynamic Partition Pruning

Let's walk through an example with two datasets: `orders` and `customers`. We will perform a join operation between these two tables and see how DPP improves query performance.

Data Setup

1. **Create the Orders DataFrame**

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Initialize Spark Session
spark = SparkSession.builder.appName("Dynamic Partition Pruning").getOrCreate()

# Sample Orders Data
orders_data = [
    (1, "2023-01-01", 1001, 500),
    (2, "2023-01-01", 1002, 1000),
    (3, "2023-02-01", 1003, 750),
    (4, "2023-02-01", 1001, 400),
    (5, "2023-03-01", 1002, 1200),
```

```
    (6, "2023-03-01", 1003, 1100),
]

orders_columns = ["order_id", "order_date", "customer_id", "amount"]

# Create Orders DataFrame
orders_df = spark.createDataFrame(orders_data, schema=orders_columns)

# Write the orders data to partitioned Parquet table (partitioned by order_date)
orders_df.write.partitionBy("order_date").mode("overwrite").parquet("/path/to/orders/")
```

2. **Create the Customers DataFrame**

```
# Sample Customers Data
customers_data = [
    (1001, "John", "USA"),
    (1002, "Alice", "UK"),
    (1003, "Bob", "Canada"),
]

customers_columns = ["customer_id", "customer_name", "customer_region"]

# Create Customers DataFrame
customers_df = spark.createDataFrame(customers_data, schema=customers_columns)

# Write customers data as a parquet file
customers_df.write.mode("overwrite").parquet("/path/to/customers/")
```

Query with Dynamic Partition Pruning

```
# Read the partitioned orders table
orders_df = spark.read.parquet("/path/to/orders/")

# Read the customers table
customers_df = spark.read.parquet("/path/to/customers/")

# Join operation with filter
joined_df = orders_df.join(customers_df, "customer_id")\
    .filter(col("customer_region") == "USA")

# Show the result
joined_df.show()
```

Explanation of the Query

- The `orders_df` is partitioned by `order_date`. Without DPP, Spark would need to scan all the partitions of the `orders` table during the join.
- When we filter the `customers` table by `customer_region == "USA"`, DPP enables Spark to prune the partitions of `orders` that do not contain relevant data based on the join condition.
- This results in scanning only the partitions of the `orders` table that are related to customers in the USA.

Checking if DPP is Applied

To verify if DPP is applied during the query execution, you can inspect the **Spark UI** under the **SQL tab** or look at the **query execution plan**.

```
# Print the query execution plan
joined_df.explain(True)
```

Look for the following in the plan:

- **DynamicPartitionPruning**: This indicates that DPP has been applied and is used to prune unnecessary partitions.

Example output from `explain()`:

```
lessCopy code== Physical Plan ==
*(5) SortMergeJoin [customer_id#55L], [customer_id#78L], Inner
:- *(2) Project [customer_id#55L, order_id#53L, order_date#54, amount#56L]
:  +- *(2) DynamicPartitionPruningSubquery [customer_id#78L]
```

Performance Considerations

1. **Reduced Data Scanning**: By pruning unnecessary partitions, DPP reduces the amount of data Spark reads, thereby improving query performance.
2. **Optimized Joins**: DPP is especially effective when joining large partitioned tables with smaller dimension tables.
3. **Automatic Optimization**: Starting from Spark 3.0, DPP is enabled by default for eligible queries. However, ensure that your queries are optimized for this feature by partitioning large tables appropriately.

Configuration Options for Tuning DPP

- `spark.sql.optimizer.dynamicPartitionPruning.enabled` (default `true`): Enable or disable DPP.
- `spark.sql.optimizer.dynamicPartitionPruning.reuseBroadcastOnly` (default `true`): Use DPP only when the broadcast hash join is involved. If set to `false`, DPP is applied to all types of joins.
- `spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio` (default `0.5`): Set the threshold ratio of partitions to prune. If fewer partitions than this ratio can be pruned, DPP is not triggered.

Conclusion

Dynamic Partition Pruning is a powerful optimization technique in PySpark that can significantly improve the performance of queries involving large partitioned tables. By selectively reading only the necessary partitions, DPP reduces I/O and speeds up query execution.