## Broadcast Hash Join

Suitable for scenarios where one of the datasets is small enough to be broadcasted to all executor nodes. The smaller dataset is broadcasted to all nodes in the cluster, and then each node joins the broadcasted dataset with partitions of the larger dataset locally. Efficient when one dataset is significantly smaller than the other.

Syntax:

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

spark =
SparkSession.builder.appName("JoinStrategies").getOrCreate()

smallDF = spark.read.csv("small_dataset.csv", header=True)
largeDF = spark.read.csv("large_dataset.csv", header=True)

joinedDF = largeDF.join(broadcast(smallDF), "key")
joinedDF.show()
```

## Sort Merge Join

Default join strategy for large datasets that do not fit into memory. Both datasets are sorted by the join key and then merged. This requires a shuffle to sort the data, which can be expensive. Suitable for large datasets where both tables are too large to broadcast.

Syntax:

```
df1 = spark.read.csv("dataset1.csv", header=True)
df2 = spark.read.csv("dataset2.csv", header=True)
```

```
joinedDF = df1.join(df2, "key")
joinedDF.show()
```

## Shuffle Hash Join

Used when the datasets are too large for a broadcast join, but one of the datasets is still relatively small. The smaller dataset is hashed and the data is partitioned based on the hash of the join keys. Each partition of the larger dataset is then joined with the corresponding partition of the smaller dataset. Efficient when one dataset is smaller but not small enough to broadcast.

Syntax:

```
df1 = spark.read.csv("large_dataset.csv", header=True)
df2 = spark.read.csv("medium_dataset.csv", header=True)
joinedDF = df1.hint("SHUFFLE_HASH").join(df2, "key")
joinedDF.show()
```

## Broadcast Nested Loop Join

Used when no other join strategy is applicable, often as a fallback method. A Cartesian product of the datasets is produced, and then a filter is applied to select the matching rows. It broadcasts the smaller table and then iterates through it for each row of the larger table. Applicable in cross joins or when join conditions are complex and other joins are not feasible.

Synatx:

```
df1 = spark.read.csv("dataset1.csv", header=True)
df2 = spark.read.csv("dataset2.csv", header=True)
```

```
joinedDF = df1.crossJoin(df2)
joinedDF.show()
```

## Cartesian Product Join

Computes the Cartesian product of two datasets. Every row of the first dataset is joined with every row of the second dataset. Used explicitly for cross joins or when the join condition is not specified.

Syntax:

```
df1 = spark.read.csv("dataset1.csv", header=True)
df2 = spark.read.csv("dataset2.csv", header=True)
joinedDF = df1.crossJoin(df2)
joinedDF.show()
```

## Skew Join

Special strategy to handle data skew in join keys. Skewed keys are detected, and their corresponding partitions are split into smaller partitions to distribute the load evenly across the cluster. Useful when certain keys in the join have disproportionately large amounts of data.

Syntax:

```
spark.conf.set("spark.sql.auto.skewJoin.enabled", "true")
df1 = spark.read.csv("large_skewed_dataset.csv", header=True)
df2 = spark.read.csv("large_skewed_dataset.csv", header=True)
joinedDF = df1.join(df2, "key")
joinedDF.show()
```