Joins

Join by broadcast

- **Example:**
  - Joining a `transactions` table with 100M rows with a `cities` lookup table with 100 rows.
- `transactions = transactions.join(broadcast(cities), on=['CityId'], how='inner')`
- The smaller table is copied across all the nodes and joins are performed locally => maximum parallelization.
- Maximum size of the broadcast table is 8GB.
- Spark maintains a default value to apply broadcast joins.

```
spark.conf.get('spark.sql.autoBroadcastJoinThreshold')
spark.conf.set('spark.sql.autoBroadcastJoinThreshold', -1) # Disable
```

Replace Joins and Aggregations with Windows

- Sometimes we need to aggregate data and add it as a feature inside the original data.
- Using window functions is more effective.
- In PySpark, this can be done using the `Window` class.

```
# first approach
df_agg = df.groupBy('city', 'team').agg(F.mean('job').alias('job_mean'))
df = df.join(df_agg, on=['city', 'team'], how='inner')
# second approach
from pyspark.sql.window import Window
window_spec = Window.partitionBy(df['city'], df['team'])
df = df.withColumn('job_mean', F.mean(col('job')).over(window_spec))
```

Join - Shuffles

- Physical movement of data across the network that needs to be written in disk.
- Network, disk I/O and serialization overhead.
- Triggered by operations like `join`, `groupby`, `distinct`.

Join - Shuffles (cont.)

- **Example:** Join a large and a medium-sized data frame.
- If the medium data frame is not small enough to be broadcasted, we can broadcast its keysets to filter the large dataframe.
- Even if we end up shuffling, there will be less overhead.

```
ids_to_broadcast = df_medium.select('id').rdd.flatMap(lambda x: x).collect()
df_reduced = df_large.filter(df_large['id'].isin(ids_to_broadcast))
df_join = df_reduced.join(df_medium, on=['id'], how='inner')
```

Bucketing

- Data is redistributed across a fixed number of buckets by a hash on the bucket value.
- This should be done before operations that involve data shuffling.
- In the case of joins, it is important to have the same number of buckets on both tables.

```
df = df.bucketBy(32, ‘key’).sortBy(‘value’)
```

Skew Data

- Skewed data can impact performance.
- Spark 3.0 comes with Adaptive Query Execution that automatically balances skewness across partitions.
- Salting
  - Adding randomization to a skewed key to distribute more uniformly.
- Repartition and Coalesce
  - Repartition redoes data distribution.
  - Coalesce increases or decreases the number of blocks in the partition. This may not reduce skewness.

Caching

Advantages

- Speed up results when a table is used in multiple queries.
- Gives the node faster access by storing it locally in memory.

```
CACHE TABLE table;
```

- Caching can be lazy, e.g. not right now but until required.

```
CACHE LAZY TABLE table;
```

- When no longer needed, is good idea to uncache.

```
UNCACHE TABLE table;
```

Python analogues

```
df.cache()
df.unpersist()
```

Delta Lake caching

- For tables in Delta Lake, caching is automatic and it happens in *disk*, not in *memory*.
- Delta Lake caching can be set when choosing the worker type when you configure your cluster.
- Some types of workers support it, but not enabled by default, see docs.

Example: PySpark

```
%sql
use taxidata;
create table yellow_tripdata as select * from yellow_tripdata_2021_01_csv;
```

```
df = spark.read.table('yellow_tripdata')

from pyspark.sql.functions import sum
distance_sum = df \
                .select(["trip_distance","payment_type", "store_and_fwd_flag"])\
                .filter(df.trip_distance>2)\
                .groupBy("payment_type", "store_and_fwd_flag")\
                .agg(sum("trip_distance"))
```

Example: PySpark (cont.)

```
trips_cached =  df \
                .select(["trip_distance","payment_type", "store_and_fwd_flag"])\
                .filter(df.trip_distance>2)\
                .cache()
trips_cached.count()

distance_sum_cached = trips_cached \
                      .groupBy("payment_type", "store_and_fwd_flag")\
                      .agg(sum("trip_distance"))
```

Example: PySpark (cont.)

```
distance_sum.show()
distance_sum_cached.show()
```

Comparison with SparkSQL

```
sql = spark.sql(' \
         SELECT payment_type, store_and_fwd_flag, SUM(trip_distance)\
         FROM yellow_tripdata \
         WHERE trip_distance > 2 \
         GROUP BY payment_type, store_and_fwd_flag;')

sql.show()
```

Random Tips

- Do not use `count()` if not needed. You can check if a data frame is empty with `len(df.head(1))>0` which is faster.
- Do not use `show()` in production code.
- Minimize data size by filtering irrelevant data (rows and columns) before joins.
- Avoid using loops.
- Use built-in functions as much as possible.

Random Tips

- Compute statistics of tables before processing.
  - Calculate separately statistics for columns involved in joining and filtering.
- Larger tables go on the left.
- Unpersist the data in cache when no longer needed.
- Prefer data frames over RDDs.
- Choose wisely your file format, if possible.
  - CSV and JSON have high write performance, but slow reading.
  - Parquet is faster for read and slower for writing.

Random Tips

- The recommended partition size is 128Mb.
  - For 128Gb of data, that means around 1000 partitions.
  - The number of partitions is obtained with `df.rdd.getNumPartitions()`

-NOT EXISTS instead of NOT IN

References

- Apache Spark Core – Practical Optimization - Talk
- Performance Tuning, Spark 3.2.0 Docs
- Apache Spark Performance Boosting