What are Spark optimization techniques with real-time use cases and examples?

1. **Data Partitioning:**
   - Use Case: Distributing data based on a specific column to optimize join operations.
   - Scenario: Suppose you have a large dataset with customer information, and you often filter data based on customer IDs. Partition the data by customer ID to improve query performance.
   - Technique: Use the `repartition` or `partitionBy` transformation to distribute data across partitions based on a specific column.

   - Example: Partitioning sales data by customer ID to improve performance when joining with customer information.

```python
partitioned_data = sales_data.repartition("customer_id")
```

2. **Broadcasting:**
   - Use Case: Efficiently joining a small lookup table with a larger fact table.
Technique: Use the `broadcast` function before joining the smaller DataFrame to avoid unnecessary shuffling.
   - Example: Broadcasting a small product lookup table to improve performance when joining with a large sales transaction table.

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

joined_data = sales_transactions.join(broadcast(product_lookup), "product_id")
```

3. **Caching and Persistence:**
   - Use Case: Reusing intermediate DataFrames in iterative machine learning algorithms.
   - Scenario: When performing iterative algorithms, like machine learning training, cache intermediate DataFrames to avoid recomputing them in every iteration.
   - Technique: Use the `cache` or `persist` transformation to store DataFrames in memory or disk.

   - Example: Caching a DataFrame used in a machine learning training loop to avoid recomputation.

```python
```

```
cached_data = data_to_cache.cache()
```

4. **Predicate Pushdown:**
   - Use Case: Reading only required columns from Parquet files to reduce I/O.
     - Scenario: While reading data from a source like Parquet, only read the columns necessary for your analysis.
     - Technique: Use the `select` transformation to read only the required columns from the source.

   - Example: Selectively reading only the relevant columns from a Parquet file.

```python
selected_columns_data = spark.read.parquet("data.parquet").select("col1", "col2")
```

5. **Coalesce and Repartition:**
   - Use Case: Balancing partitions after filtering large datasets.
     - Scenario: After filtering a large DataFrame, you may end up with a skewed partition distribution. Coalesce or repartition the DataFrame to balance the partitions.
     - Technique: Use the `coalesce` or `repartition` transformation to control the number of partitions.

   - Example: Coalescing or repartitioning a DataFrame to reduce skew after filtering.

```python
coalesced_data = large_data.coalesce(4)
```

6. **Avoiding Shuffling:**
   - Use Case: Minimizing data shuffling during aggregations.
     - Scenario: When performing joins or aggregations, minimize data shuffling by using transformations like `reduceByKey` instead of `groupByKey`.
     - Technique: Use transformations that perform local aggregations before shuffling data.

   - Example: Using `reduceByKey` instead of `groupByKey` for better performance.

```python
aggregated_data = data.rdd.reduceByKey(lambda x, y: x + y).toDF()
```

7. **Column Pruning:**

- Use Case: Reducing memory consumption by selecting only required columns.
    - Scenario: If your application only needs a few columns from a DataFrame, prune the unnecessary columns to reduce memory consumption.
    - Technique: Use the `select` transformation to keep only the required columns.

- Example: Selecting only the necessary columns for analysis.

```python
selected_columns_data = full_data.select("col1", "col2")
```

8. **UDFs Optimization:**
   - Use Case: Choosing built-in functions over UDFs for efficiency.
    - Scenario: While using User-Defined Functions (UDFs), prefer built-in Spark functions whenever possible, as they are optimized for distributed processing.
    - Technique: Use Spark's built-in functions like `withColumn`, `when`, `concat`, etc., instead of UDFs for common operations

   - Example: Using built-in Spark functions for common operations.

```python
from pyspark.sql.functions import col, when

transformed_data = full_data.withColumn("new_col", when(col("old_col") > 0, col("old_col")).otherwise(0))
```

9. **Dynamic Allocation:**
   - Use Case: Dynamically adjusting the number of executors based on workload.
    - Scenario: In a dynamic workload scenario, allow Spark to automatically adjust the number of executors based on resource availability.
    - Technique: Enable dynamic allocation in the Spark configuration.

   - Example: Enabling dynamic allocation in Spark configuration.

```python
conf = SparkConf().set("spark.dynamicAllocation.enabled", "true")
```

10. **Bucketing and Sorting:**
    - Use Case: Improving query performance by organizing data in Hive or Parquet.

- Scenario: When writing data to Hive or Parquet, use bucketing and sorting for efficient data storage and retrieval.
- Technique: Use the `bucketBy` and `sortBy` options while writing data.

- Example: Writing data to Parquet with bucketing and sorting.

```python
bucketed_sorted_data.write.bucketBy(10, "column_name").sortBy("column_name").parquet("bucketed_sorted_data.parquet")
```

Each technique has its own use case and benefits, but the actual effectiveness depends on your specific data and workload. Always profile and monitor your Spark jobs to identify bottlenecks and apply the appropriate optimization strategies.

Summary:

Spark Optimization steps:

1. Understand your hardware:-

    - core count and speed
    - memory per core(storage & working)
    - local disk type, size, count and speed
    - network speed and topology
    - data lake properties(rate limits)
    - Cost/Core/ hour
        - financial for cloud
        - opportunity for shared and on-prem
    - Get a baseline threshold per workload of the cluster dev/prod e.g 70%

2. Minimize data scan(lazy loads)

    - data skipping
        - hive partitions

- bucketing
- databricks delta z ordering

3. Check on Spark Partitions:-

- input: control size

spark.sql.files.maxPartitionBytes()

-Shuffles: control count spark.sql.shuffle.partitions

- Output: control size
  - Coelesce(n) to shrink
  - Repartition(n) to increase or balance(shuffle)
  - df.write.option("maxRecordsPerFile", N)

Partirtions Right Sizing- Shuffle:

Largest Shuffle stage- target size<= 200MB/partition

partition count = stage input data/ target size

Input Partirtions right sizing:

Use spark default 128mb size unless -

- increase parallelism
- Heavily nested/ repeatitive data
- Generating data (explode)
- upstream source structure is not optimal
- UDFs

Output Right partition sizing:

Write Once => Read many

- more time to write but faster to read

Perfect writes limit parallelism

- compactions(minor & major)

4. Advanced Optimizations:

- Finding Imbalances: joins, groupBy, etc.
- Persistence(memory, disk) when you have repeatitive data scans then unpersist after job
- Join Optimizations
  - SortMergeJoin(standard)- when both sides are large

  - BroadcastJoin(fastest)- when one side is small

    Automatic if one side < spark.sql.autoBroadcastJoinThreshold(default 10m)

    Risks:

    - Not enough driver memory
    - df > spark.driver.maxResultSize
    - df > Single Executer available working memory

    Prod: mitigate risks- validation functions

  - Skew Joins

    Salting- add column to each side with random int between 0 to spark.sql.partitions-1 to both sides

    - add join clause to include join on generated column above
    - drop temp cols from result

  - Skew Aggregates: df.groupby('city', 'state').agg(avg('sales')).orderBy(col).desc()

    solution with salting -

    saltval = random(0, spark.conf.get(org.shuffle.partitions)-1)

    df.withColumn('salt', lit(saltval)).groupBy('city', 'state','salt').agg(avg('sales')).drop('salt').orderBy(col).desc()

- BroadcastNestedLoopJoins: instead of IN , NOT IN use EXISTS, NOT EXISTS with AND operation

- Range join optimization

  - poin in interval range join: predicate specify value in one relation that is between two values from the other relation

  - interval overlap range join: predicate specifies an overlap of intervals between two values from each relation

5. Omit Expensive Operations:

- Repartition - use coelesce or shuffle partition count

- Count, CountDistinct avoid whereever not required and approxCountDistinct()

- If Distincts are required put them in right place: Use dropDuplicates() before the join operation and before groupBy.

6.UDF penalties:

Traditional UDFs can't use Tungsten

- use org.apache.spark.sql.functions

- use pandas UDFs it uses apache arrow