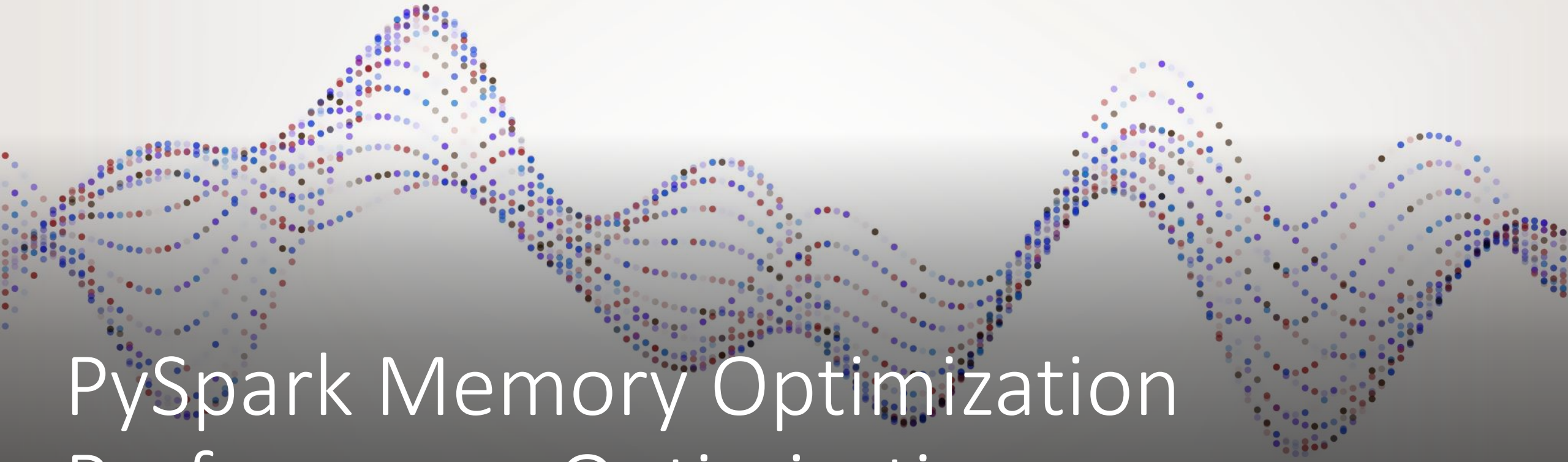

PySpark Memory Optimization Performance Optimization

Surendra Panpaliya

Introduction

- Optimizing memory usage
- Performance in PySpark
- Essential for efficiently processing
- large datasets and improving
- overall execution speed

Strategies for Memory optimization and Performance improvement

Surendra Panpaliya

Data Serialization

PySpark provides options

for choosing different data serialization formats.

Parquet is a columnar storage format that

can greatly reduce memory consumption and

improve I/O performance.

Data Serialization

Use

spark.conf.set("spark.sql.parquet.compression.codec", "snappy")

to enable snappy compression for Parquet files.

Set the Parquet compression codec

to snappy for better storage efficiency.

Snappy compression

- Refers to a fast and efficient
- data compression algorithm
- used to reduce the size of files or data streams
- while maintaining relatively high compression
- decompression speeds.

Snappy compression

Designed for situations

where both compression and decompression

performance are important factors.

Use DataFrame API

- Prefer using the DataFrame API over the RDD API.
- DataFrames offer optimizations such as
- Catalyst query optimizer and
- Tungsten execution engine,
- Significantly improve query performance.

Memory Management

Adjust memory settings using

spark.executor.memory

spark.driver.memory

configuration options

to allocate an appropriate amount of memory

to executors and the driver.

Memory Management

Set spark.memory.fraction

to control the fraction of heap space

used for storage memory.

Increase this if you have

more memory available and

need better caching.

Memory Management

Tune memory allocation

For executors and driver.

spark.conf.set("spark.executor.memory", "4g")

spark.conf.set("spark.driver.memory", "2g")

Memory Management

Optimize memory fractions.

spark.conf.set("spark.memory.fraction", "0.7")

spark.conf.set("spark.memory.storageFraction", "0.5")

Caching and Persistence

Cache intermediate results

using the cache() or

persist() methods

to avoid recomputation and

reduce memory pressure.

Caching and Persistence

Prioritize caching smaller datasets

that are reused frequently.

Cache frequently used DataFrames

to avoid recomputation.

df.cache()

Shuffle Optimization

- Shuffling involves redistributing data
- Across partitions or nodes,
- Can be a resource-intensive
- Time-consuming process.

Shuffle Optimization

Tune shuffle configurations

spark.conf.set("spark.shuffle.service.enabled", "true")

spark.conf.set("spark.reducer.maxSizeInFlight", "128m")

spark.conf.set("spark.sql.shuffle.partitions", "200")

Data Skew Handling

Address data skew by

using techniques like

bucketing or salting

to distribute data evenly among partitions.

Data Skew Handling

Use techniques like

skewed joins

to handle skew in

join operations.

Data skew

Data skew refers to an uneven distribution

of data across partitions, buckets, or

nodes in a distributed computing environment.

Data skew

Some partitions or nodes

contain significantly more data than others,

creating an imbalance in the workload and

potentially leading to performance bottlenecks.

Data Skew Handling : Salting

```
from pyspark.sql.functions import col
def add_salt(col_name):
    return col(col_name) + (rand() * 100).cast("int")

df = df.withColumn("salted_col", add_salt("key"))
```

UDFs and Pandas UDFs

Use User Defined Functions (UDFs) judiciously

as they can impact performance

due to Python serialization overhead.

Consider using Pandas UDFs

For vectorized processing

on Pandas DataFrames within PySpark.

UDFs and Pandas UDFs

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def square_udf(x):
    return x * x

square_udf = udf(square_udf, IntegerType())
df = df.withColumn("square", square_udf(df["value"]))
```

Broadcasting

Use broadcast joins for small tables

to reduce data shuffling and

improve performance.

Broadcasting

Use broadcast joins for small tables

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

```
result = df1.join(broadcast(df2), "key")
```

Hardware Configuration

Choose appropriate

hardware configurations

based on the size of your data

processing needs.

Use cluster management tools

to scale resources as needed.

Monitoring and Profiling

Use Spark's built-in monitoring tools and

UI to monitor memory usage,

query execution plans, and

other performance metrics.

Profile your code

using tools like

pyspark.sql.execution.debug

Partitioning

Optimize data partitioning

based on the nature of your queries

to minimize data movement

during operations

like joins and aggregations.

Partitioning

Optimize data partitioning for join operations

```
df1 = df1.repartition("key")
```

```
df2 = df2.repartition("key")
```

```
result = df1.join(df2, "key")
```

Garbage Collection

Tune garbage collection settings

to avoid excessive pauses

due to garbage collection activities.

Using Vectorized Expressions

Use Spark's built-in vectorized expressions

for optimized operations

from pyspark.sql.functions import expr

df = df.withColumn("result", expr("col1 + col2 * 2"))

Conclusion

Remember that the impact of each optimization technique

can vary based on your specific use case and dataset.

Experimentation and profiling are key

to identifying the most effective optimizations

for your PySpark applications.

PySpark Caching and Persistence

Caching and persistence in PySpark are techniques to store intermediate DataFrames or RDDs in memory to avoid recomputation and improve query performance.

PySpark Caching and persistence

```
from pyspark.sql import SparkSession
# Initialize a Spark session
spark = SparkSession.builder \
    .appName("CachingExample") \
    .getOrCreate()
# Read data from a CSV file
input_path = "/path/to/input/file.csv"
```

PySpark Caching and persistence

```
df = spark.read.csv(input_path, header=True, inferSchema=True)
# Cache the DataFrame in memory
df.cache()
# Perform some operations on the DataFrame
filtered_df = df.filter(df["age"] > 25)
```

PySpark Caching and persistence

```
grouped_df = filtered_df.groupBy("gender").agg({"salary": "avg"})
# Show the results
grouped_df.show()
```

Persistence example

```
# Persist the DataFrame to disk in memory and deserialized format
df.persist()

# Perform more operations on the persisted DataFrame
selected_df = df.select("name", "age")

# Show the results
selected_df.show()

# Unpersist the DataFrame from memory
df.unpersist()
```

PySpark Caching and persistence example

```
# Perform additional operations without caching
processed_df = df.withColumn("bonus", df["salary"] * 0.1)

# Show the results
processed_df.show()

# Stop the Spark session
spark.stop()
```

Summary

Start by reading data from a CSV file and

creating a DataFrame df

Use the cache() method

to cache the DataFrame in memory.

Summary

Perform operations like filtering and

aggregation on the cached DataFrame.

Use the persist() method

to persist the DataFrame

to memory and disk in the deserialized format.

Summary

Perform additional operations

on the persisted DataFrame.

use the unpersist() method

to remove the DataFrame from memory.

Continue to perform operations

on the original DataFrame without caching.

Summary

By caching and persisting DataFrames,

Can avoid re-computation

of the same transformations

Improve the overall performance

Of PySpark applications.

Summary

It's important to consider

memory availability and

the frequency of data access

when deciding which

DataFrames to cache or persist.

PySpark Broadcast variables and accumulators

Surendra Panpaliya

Broadcast Variables

Broadcast variables allow you

to efficiently share a read-only variable

across all the tasks in a Spark job,

reducing data transfer overhead.

Broadcast Variables

```
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("BroadcastExample") \
    .getOrCreate()

# Create a broadcast variable
broadcast_var = spark.sparkContext.broadcast([1, 2, 3, 4, 5])
```

Broadcast Variables

```
# Create an RDD and use the broadcast variable in a transformation
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
filtered_rdd = rdd.filter(lambda x: x in broadcast_var.value)
```

Broadcast Variables

```
# Collect and print the results
result = filtered_rdd.collect()
print(result)
# Stop the Spark session
spark.stop()
```

Accumulators

Accumulators are variables that

Can be used to accumulate values

Across multiple tasks in a parallel manner

While performing associative

Commutative operations.

Accumulators

```
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("AccumulatorExample") \
    .getOrCreate()

# Create an accumulator

accumulator = spark.sparkContext.accumulator(0)
```

Accumulators

```
# Create an RDD and use the accumulator in a transformation

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

rdd.foreach(lambda x: accumulator.add(x))
```

Accumulators

```
# Print the value of the accumulator
print("Accumulated value:", accumulator.value)

# Stop the Spark session
spark.stop()
```

Summary

The first example demonstrates

how to use a broadcast variable

to efficiently share a list of values

across RDD transformations.

Summary

The second example shows

how to use an accumulator

to accumulate values from

RDD in a parallel manner.

Conclusion

Both broadcast variables and accumulators

are useful tools for optimizing and

aggregating data in

distributed computations

within PySpark.

PySpark Working with large datasets and out-of-memory data processing

Surendra Panpaliya

Introduction

Working with large datasets and

handling out-of-memory data processing

are common challenges in big data analytics.

PySpark provides several techniques

to tackle these challenges effectively

1. Distributed Computing

Allowing it to process large datasets

by distributing the workload

across multiple nodes in a cluster.

2. Lazy Evaluation

Transformations are not immediately executed.

Instead, a logical plan is built, and

computations are executed only when

an action is performed.

Helps in optimizing the execution plan.

3. Partitioning and Repartitioning

Use partitioning

to split data into smaller chunks,

allowing parallel processing.

Repartitioning can be used

to control the number of partitions and

balance the data distribution.

4. Caching and Persistence

Cache intermediate DataFrames or

RDDs in memory using .cache() or .persist()

to avoid recomputation.

This is especially helpful

when data is reused in multiple computations.

5. Broadcast Joins

For small tables,

use broadcast joins

to minimize data shuffling

during join operations.

6. Aggregation Strategies

Use efficient aggregation strategies

like using the Catalyst query optimizer

for optimizing aggregations and

grouping operations.

7. Off-Heap Memory Management

Allocate memory off-heap

to manage the memory consumption of

JVM objects.

8. Data Compression

Use compression codecs like

Snappy or Gzip to store data

more efficiently.

9. Using Spark SQL and DataFrames

Leverage Spark SQL's built-in optimizations and

Catalyst query optimizer

for efficient SQL querying

on large datasets.

10. Leveraging Parquet Format

Store data in Parquet format,

which is columnar and

provides better compression

data skip capabilities.

11. Incremental Processing

Use incremental processing techniques

to avoid recomputing the entire dataset

when new data arrives.

12. Sampling and Approximations

Use sampling techniques

for exploratory data analysis

to avoid processing the entire dataset.

13. Handling Skewed Data

Apply techniques like salting or bucketing to distribute skewed data more evenly across partitions.

14. Out-of-Memory Processing

In case of very large datasets that

don't fit in memory,

consider using the iterative or

disk persistence levels.

15. External Storage

Use external storage

like HDFS or cloud storage

for storing and accessing

large datasets.

Summary and Conclusion

Effective optimization

depends on the specific characteristics

of your data

Analytical requirements.

Summary and Conclusion

Experimentation and profiling are

key to finding the optimal combination

of techniques for your use case.