Spark Optimization Techniques

Code-Level Optimizations

a. Avoid Using collect() Unnecessarily

```
# ❌ Bad
data = df.collect()

# ✅ Good
df.show(10)      # for preview
df.take(10)      # for small samples
```

Code-Level Optimizations

b. Use mapPartitions Instead of map

```
# ❌ map (inefficient for DB connections)
rdd.map(lambda x: db_write(x))

# ✅ mapPartitions
def write_partition(iter):
    conn = create_db_connection()
    results = [db_write(record, conn) for record in iter]
    conn.close()
    return results

rdd.mapPartitions(write_partition)
```

Code-Level Optimizations

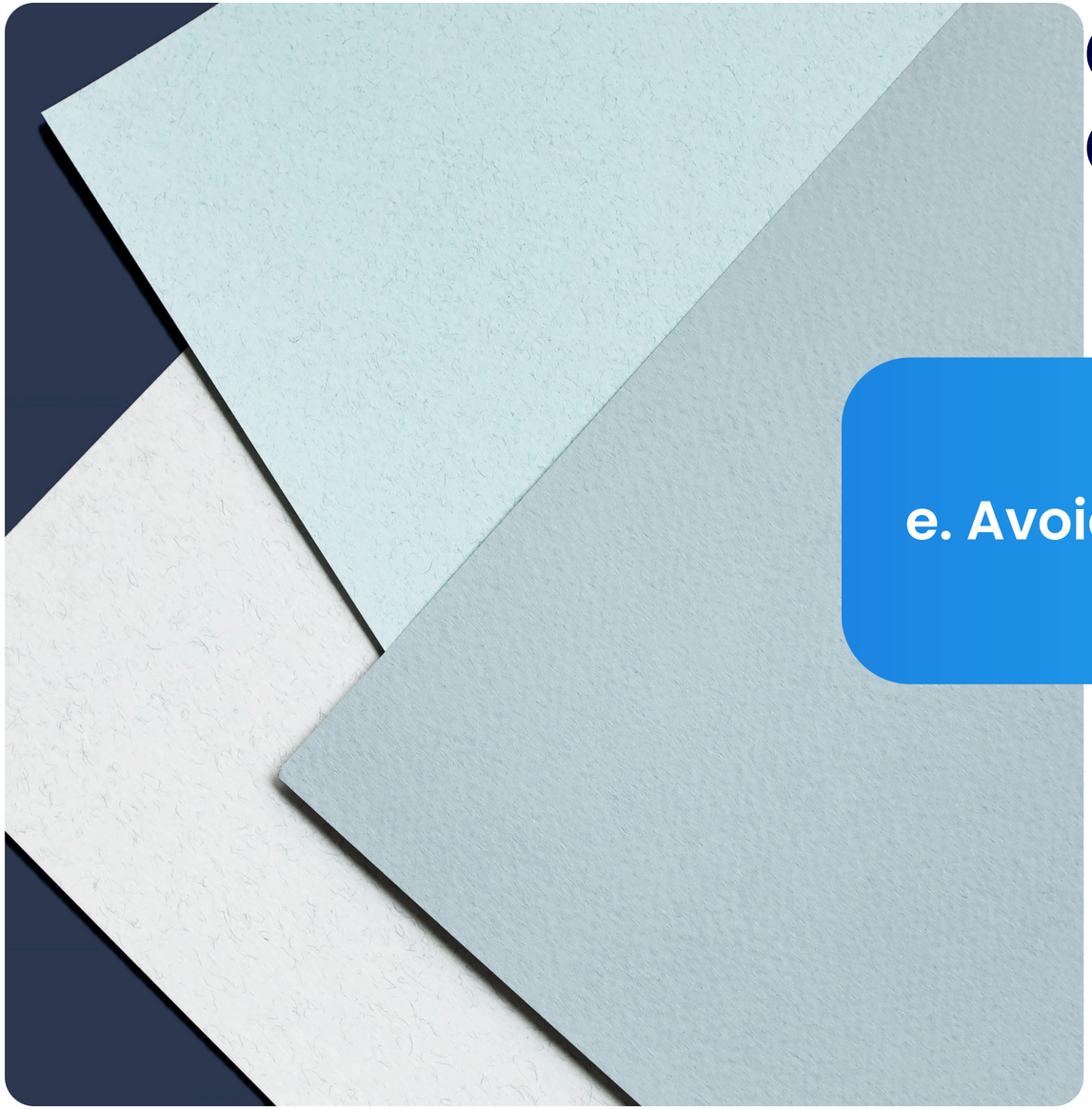
c. Use broadcast for Small Lookup Tables

```
from pyspark.sql import functions as F

# ❌ Normal join (causes shuffle)
df_large.join(df_small, "id")

# ✅ Broadcast join
from pyspark.sql import SparkSession
from pyspark.sql.functions import broadcast

df_large.join(broadcast(df_small), "id")
```

Code-Level Optimizations

d. Cache/Persist Reused Data

```
# ❌ Without cache → recomputes each time
df_filtered = df.filter(df["age"] > 30)
print(df_filtered.count())
print(df_filtered.collect())

# ✅ With cache
df_filtered = df.filter(df["age"] > 30).cache()
```



Code-Level Optimizations

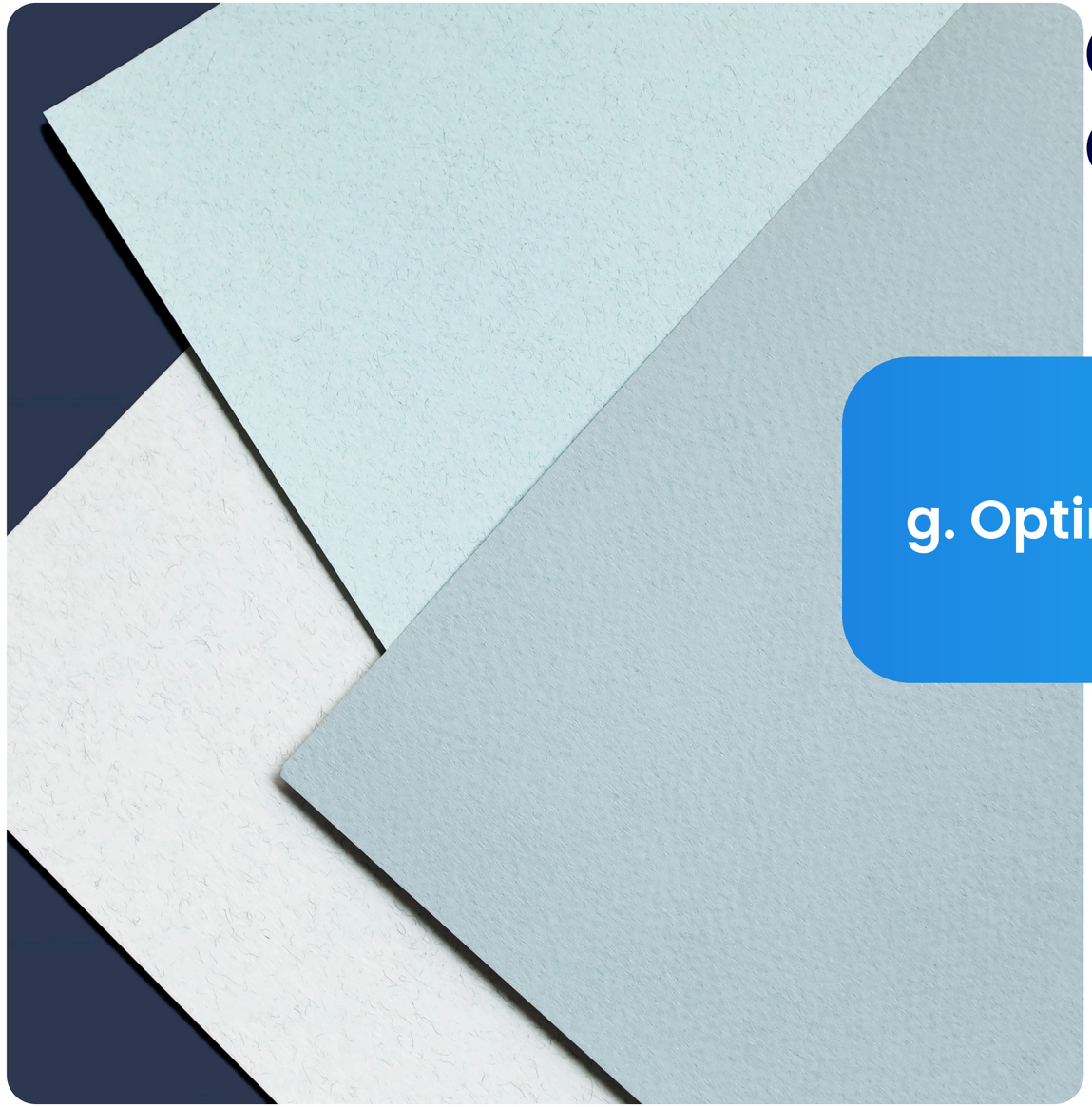
e. Avoid UDFs, Use Spark SQL Functions

```
from pyspark.sql.functions import col, upper

# ❌ UDF
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
@udf(returnType=StringType())
def to_upper(s): return s.upper()
df.withColumn("name_upper", to_upper(col("name")))

# ✅ Built-in function
df.withColumn("name_upper", upper(col("name")))
```

Code-Level Optimizations

f. Partition Data Before Writing

```
# ❌ Single output file (bad parallelism)
df.write.csv("output.csv")

# ✅ Partition by column (parallel, scalable)
df.write.partitionBy("year",
"month").parquet("/output/path/")
```



Code-Level Optimizations

g. Optimize Shuffle Partitions

```
# Default 200 partitions → too many small files
spark.conf.set("spark.sql.shuffle.partitions", "200")

# ✅ Tune based on data size
spark.conf.set("spark.sql.shuffle.partitions", "50")
```

Code-Level Optimizations

h. Push Down Filters / Column Pruning

```
# ❌ Select all columns
df = spark.read.parquet("data.parquet")

# ✅ Select only needed
df = spark.read.parquet("data.parquet").select("id",
"name").filter("age > 30")
```

Code-Level Optimizations

i. Repartition vs Coalesce

```
df.repartition(100)   # Increase partitions
df.coalesce(10)       # Reduce partitions
```

Cluster-Level Optimizations

Cluster-Level Optimizations

a. Executor & Core Tuning

- Don’t overload executors with too many cores.
- Rule of thumb: 5 cores per executor.

```
--executor-memory 4G
--executor-cores 5
--num-executors 10
```

Cluster-Level Optimizations

b. Use Dynamic Allocation

- Enable Spark to scale resources automatically.

```
--conf spark.dynamicAllocation.enabled=true
--conf spark.shuffle.service.enabled=true
--conf spark.dynamicAllocation.minExecutors=2
--conf spark.dynamicAllocation.maxExecutors=50
```

Cluster-Level Optimizations

c. Choose Right File Format

- Prefer Parquet/ORC over CSV/JSON (columnar, compressed).
- Use Snappy compression for Parquet (splittable & fast).

```
df.write.mode("overwrite").parquet("/path/output/", compression="snappy")
```

Cluster-Level Optimizations

d. Data Locality

- Co-locate Spark with HDFS nodes.
- Ensure spark.locality.wait is tuned (default = 3s).

--conf spark.locality.wait=1s

Cluster-Level Optimizations

e. Adaptive Query Execution (AQE)

- Introduced in Spark 3.x — optimizes joins/shuffles at runtime.

```
--conf spark.sql.adaptive.enabled=true
--conf spark.sql.adaptive.shuffle.targetPostShuffleInputSize=64MB
```

Cluster-Level Optimizations

g. Serialization

- Use Kryo instead of Java serialization (faster, smaller).

--conf spark.serializer=org.apache.spark.serializer.KryoSerializer

Cluster-Level Optimizations

h. Shuffle & Spill Optimization

- Store shuffle data on SSDs if possible.
- Increase memory fraction for shuffle:

```
--conf spark.memory.fraction=0.6
```

Summary Code-Level

01 Avoid collect(), prefer actions (take, show).

02 Use mapPartitions, broadcast, cache, SQL functions.

03 Filter early, select fewer columns.

04 Optimize shuffle partitions, repartition/coalesce wisely.

Cluster-Level Optimizations

h. Shuffle & Spill Optimization

- Store shuffle data on SSDs if possible.
- Increase memory fraction for shuffle:

```
--conf spark.memory.fraction=0.6
```

Summary Cluster-Level

01 Tune executor memory/cores.

02 Enable dynamic allocation.

03 Use Parquet/ORC + Snappy. Use Kryo serialization.

04 Enable AQE and Handle small files & shuffle efficiently.