05 Bucketing in Spark

Apache Spark Bucketing: Comprehensive Lecture Notes

Introduction

Bucketing is a data organization technique in Apache Spark that divides datasets into manageable chunks based on hash values. This technique significantly improves performance for operations like:

- **Filtering**
- **Aggregations (GROUP BY)**
- **Joins**

Key Benefit

Bucketing pre-shuffles data during write time, avoiding expensive shuffle operations during query execution.

What is Bucketing?

Definition

Bucketing divides data into a fixed number of buckets based on the hash value of one or more columns. Each row is assigned to a bucket using the formula:

```
Bucket Number = hash(column_value) % number_of_buckets
```

Example Datasets

Orders Table:

order_id	product_id	customer_id	quantity	total_amt	order_date
1	22	105	10	104	2023-03-10
2	19	542	12	114	2023-03-10
3	11	199	4	2567	2023-03-10
4	87	76	10	1268	2023-03-10
5	9	225	6	1136	2023-03-10
6	24	980	11	658	2023-03-10
7	22	14	55	572	2023-03-10
8	10	5	12	7768	2023-03-10

Products Table:

product_id	product_name	category	brand	price
19	product_19	groceries	brand_2	9.5
11	product_11	electronics	brand_3	641.75
87	product_87	electronics	brand_4	126.8
9	product_9	groceries	brand_5	189.33
24	product_24	apparel	brand_6	59.81
10	product_10	electronics	brand_7	647.33
22	product_22	electronics	brand_1	10.4

Bucketing vs Other Techniques

Comparison Table

Approach	Pros	Cons	Best Use Case
No Organization	Simple to implement	Full table scan required; Shuffle on every operation	Small datasets only
Partitioning	Good for low-cardinality columns	Creates small file problem with high-cardinality columns	Date-based columns, categories
Bucketing	Avoids shuffle; Optimal for high-cardinality columns	Requires upfront planning	High-cardinality join/filter columns

When NOT to Use Each Technique

❌ Raw Storage (No Organization)

- Problem: Scans all records for every query
- Result: Poor performance on large datasets

❌ Partitioning on High-Cardinality Columns

- Problem: Creates too many small files (small file problem)
- Example: Partitioning by `product_id` with thousands of unique products
- Result: Metadata overhead, slow queries

✅ Bucketing on High-Cardinality Columns

- Solution: Fixed number of buckets regardless of unique values
- Example: 1000 unique products → 4 buckets (not 1000 files)

How Bucketing Works

The Bucketing Formula

```
Bucket Assignment = hash(bucketing_column) % number_of_buckets
```

Visual Example: 4 Buckets

Let's bucket the Orders table by `product_id` into 4 buckets:

```
For simplification: hash(product_id) = product_id

Product ID 22: 22 % 4 = 2 → Bucket 2
Product ID 19: 19 % 4 = 3 → Bucket 3
Product ID 11: 11 % 4 = 3 → Bucket 3
Product ID 87: 87 % 4 = 3 → Bucket 3
Product ID  9:  9 % 4 = 1 → Bucket 1
Product ID 24: 24 % 4 = 0 → Bucket 0
Product ID 22: 22 % 4 = 2 → Bucket 2
Product ID 10: 10 % 4 = 2 → Bucket 2
```

Bucket Distribution

```
┌──────────────┐  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│  Bucket 0    │  │  Bucket 1    │  │  Bucket 2    │  │  Bucket 3    │
├──────────────┤  ├──────────────┤  ├──────────────┤  ├──────────────┤
│ product: 24  │  │ product: 9   │  │ product: 22  │  │ product: 19  │
│              │  │              │  │ product: 22  │  │ product: 11  │
│              │  │              │  │ product: 10  │  │ product: 87  │
└──────────────┘  └──────────────┘  └──────────────┘  └──────────────┘
```

Bucketing for Different Operations

1. Filter Operations

Without Bucketing

```
Query: WHERE product_id = 22
Problem: Scans ALL records
```

With Bucketing

```
Query: WHERE product_id = 22
Optimization:
  1. Calculate: 22 % 4 = 2
  2. Read ONLY Bucket 2
  3. Skip Buckets 0, 1, 3

Result: Reduced search space by 75%!
```

This is called "Bucket Pruning"

2. Join Operations

Scenario: Join Orders with Products on product_id

Without Bucketing - 3 Expensive Steps:

```
Step 1: SHUFFLE - Redistribute data across nodes (COSTLY!)
Step 2: SORT    - Sort by join key
Step 3: MERGE   - Perform the join
```

Physical Plan:

```
Exchange/Shuffle (Orders)  ⟵ EXPENSIVE
    ↓
Sort (Orders)
    ↓
Exchange/Shuffle (Products) ⟵ EXPENSIVE
    ↓
Sort (Products)
    ↓
SortMergeJoin
```

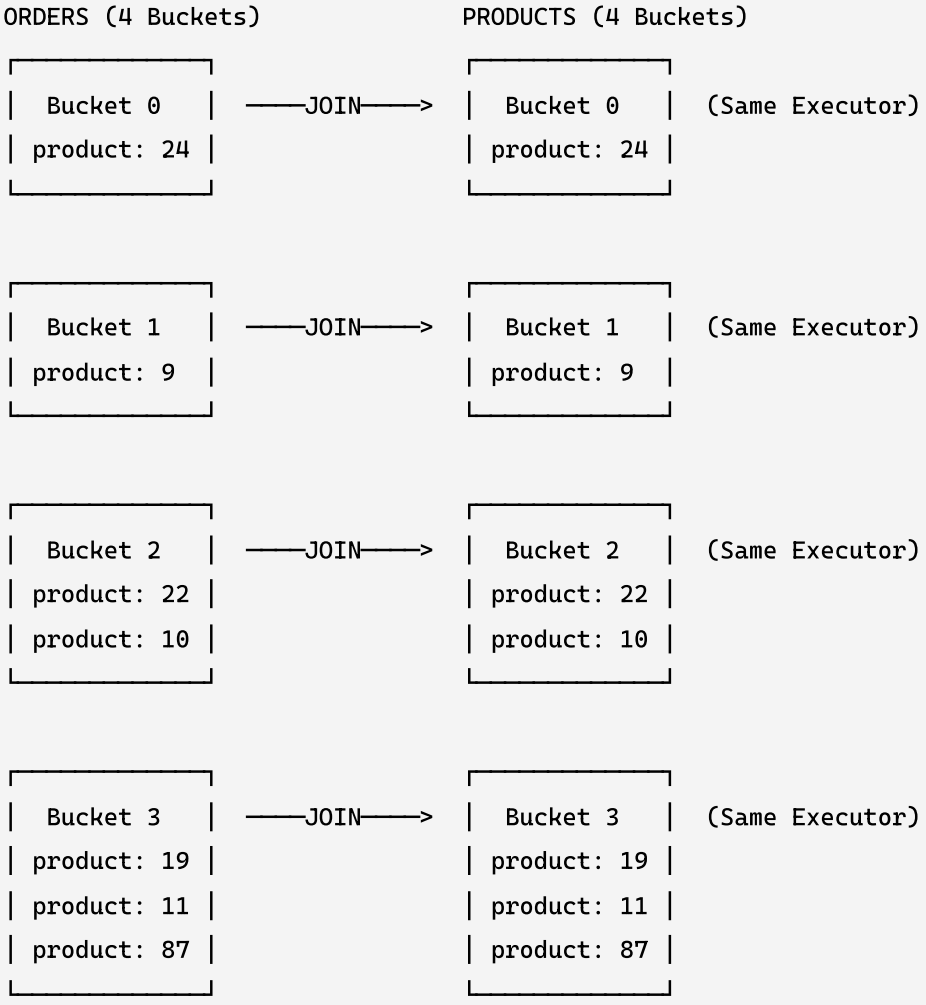
With Bucketing - Only 2 Steps:

```
Step 1: SORT    - Data already co-located (shuffle done at write time)
Step 2: MERGE   - Perform the join

✅ SHUFFLE ELIMINATED!
```

Physical Plan:

```
Sort (Orders Bucketed)
    ↓
Sort (Products Bucketed)
    ↓
SortMergeJoin

Note: No Exchange/Shuffle operations!
```

Bucketing Join Visualization




```
Key Point: Matching buckets are co-located on same executor!
           No network shuffle needed!
```

3. Aggregation Operations (GROUP BY)

Example Query:

```sql
SELECT product_id, SUM(total_amt) as total_sales
FROM orders
GROUP BY product_id
```

Without Bucketing:

```
Step 1: Local/Partial Aggregation
Step 2: SHUFFLE (Exchange HashPartitioning) ⟵ EXPENSIVE
Step 3: Global Aggregation
```

With Bucketing:

```
Step 1: Local/Partial Aggregation (per bucket on same executor)
Step 2: Global Aggregation (no shuffle needed!)

✅ SHUFFLE ELIMINATED!
```

Why? All rows with the same `product_id` are already in the same bucket on the same executor.

Join Scenarios with Bucketing

Scenario Matrix

Dataset 1	Dataset 2	Shuffle Required?	Performance
Bucketed (X buckets, col A)	Bucketed (X buckets, col A)	❌ No	⭐⭐⭐ Excellent
Bucketed (X buckets, col A)	Bucketed (Y buckets, col A)	⚠️ Partial (one dataset)	⭐⭐ Good
Bucketed (X buckets, col A)	Bucketed (X buckets, col A), but JOIN on col B	✅ Full Shuffle	⭐ Poor
Not Bucketed	Not Bucketed	✅ Full Shuffle	⭐ Poor

Key Takeaways:

1. ✅ **Best**: Both datasets bucketed with same number of buckets on the join column
2. ⚠️ **Acceptable**: Different bucket counts (partial shuffle)
3. ❌ **Avoid**: Bucketing on different columns than join key

Determining Optimal Bucket Count

Formula

```
Number of Buckets = Dataset Size (MB) / Optimal Bucket Size (MB)

where Optimal Bucket Size = 128–200 MB
```

Example Calculation

```
Dataset Size = 1000 MB (1 GB)
Optimal Bucket Size = 200 MB

Number of Buckets = 1000 / 200 = 5 buckets
```

Estimating Dataset Size

When data is not yet written to disk, use this formula:

```
Dataset Size (MB) = (N × V × W) / (1024²)

where:
  N = Number of records
  V = Number of variables (columns)
  W = Average width in bytes per variable
```

Column Width Guidelines

Data Type	Width (bytes)
Small Integer (TINYINT)	1
Medium Integer (SMALLINT)	2
Integer (INT)	4
Large Integer (BIGINT)	8
Float/Double	4-8
String	Variable (avg length)

Note: Calculating size requires one full scan of the dataset, but this is a one-time cost.

Implementation Examples

Colab Notebook: https://colab.research.google.com/drive/1bj4CZ7FUAlY-ae28M3EwmKTodtcQT6yt?usp=sharing

1. Creating Bucketed Tables

PYTHON

```python
# Create Spark Session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Bucketing").getOrCreate()

# Load data
df_orders = spark.read \
  .format("csv") \
  .option("header", "true") \
  .option("inferSchema", "true") \
  .load("/content/orders.csv")

df_orders.show(5, False)
df_orders.printSchema()



df_products = spark.read \
  .format("csv") \
  .option("header", "true") \
  .option("inferSchema", "true") \
  .load("products.csv")

df_products.show(5, False)
df_products.printSchema()

# Bucket orders table
(
    df_orders
    .write.bucketBy(4, col="product_id")
    .mode("overwrite")
    .saveAsTable("orders_bucketed")
)

# Bucket products table
(
    df_products
    .write.bucketBy(4, col="product_id")
    .mode("overwrite")
    .saveAsTable("products_bucketed")
)



#
```

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.saveAsTable.html

2. Join with Bucketed Tables

PYTHON

```python
# Read bucketed tables
orders_bucketed = spark.table("orders_bucketed")
products_bucketed = spark.table("products_bucketed")

# Perform join
result = orders_bucketed.join(
    products_bucketed,
    on="product_id",
    how="inner"
)

# Check physical plan
result.explain()
```

Physical Plan Output (Key Observations):

```
== Physical Plan ==
*(3) SortMergeJoin [product_id#1], [product_id#8], Inner
:- *(1) Sort [product_id#1 ASC NULLS FIRST]
:  +- FileScan parquet [product_id#1, ...]
:     ReadSchema: struct<product_id:int, ...>
+- *(2) Sort [product_id#8 ASC NULLS FIRST]
   +- FileScan parquet [product_id#8, ...]
      ReadSchema: struct<product_id:int, ...>
```

✅ Notice: NO "Exchange" (shuffle) operations!

3. Aggregation with Bucketing

PYTHON

```python
# Without bucketing
df_orders.groupBy("product_id") \
    .agg({"total_amt": "sum"}) \
    .explain()

# Output includes: Exchange hashpartitioning (SHUFFLE!)

# With bucketing
orders_bucketed.groupBy("product_id") \
    .agg({"total_amt": "sum"}) \
    .explain()

# Output: NO Exchange operations (NO SHUFFLE!)
```

4. Filter with Bucket Pruning

PYTHON

```python
# Filter on bucketed column
result = orders_bucketed.filter("product_id = 1") \
    .groupBy("product_id") \
    .agg({"total_amt": "sum"})

result.explain()
```

Physical Plan Output:

```
== Physical Plan ==
*(2) HashAggregate( ... )
+- *(1) FileScan parquet [product_id#1, total_amt#5]
       SelectedBucketsCount: 1 out of 4  <-- BUCKET PRUNING!

✅ Only 1 bucket scanned instead of 4!
```

Best Practices

✅ DO:

1. **Use bucketing for high-cardinality columns**
   - Example: user_id, product_id, transaction_id
2. **Bucket on join/filter columns**

- Bucket by the columns you frequently join or filter on

3. **Choose appropriate bucket count**
   - Use the formula: Dataset Size / 128-200 MB
4. **Use for repeated operations**
   - Bucketing overhead is justified when queries run multiple times
5. **Combine with sorting**
   - `sortBy()` in addition to `bucketBy()` for better performance

❌ DON'T:

1. **Don't bucket on low-cardinality columns**
   - Example: gender, status (use partitioning instead)
2. **Don't create too many buckets**
   - Results in small files problem
3. **Don't create too few buckets**
   - Reduces parallelism and benefit of bucketing
4. **Don't bucket if data changes frequently**
   - Rewriting bucketed data is expensive
5. **Don't bucket for one-time queries**
   - Overhead not worth it for single-use datasets

Performance Summary

Operation	Without Bucketing	With Bucketing	Improvement
Filter	Full table scan	Bucket pruning (1/N scan)	N× faster
Join	Shuffle + Sort + Merge	Sort + Merge	30-50% faster
GROUP BY	Partial Agg + Shuffle + Global Agg	Partial Agg + Global Agg	20-40% faster

Where N = number of buckets

Key Concepts Summary

🎯 Core Principle

Bucketing pre-shuffles data at write time to avoid expensive shuffles at query time.

🔑 Key Formula

```
Bucket Number = hash(column_value) % number_of_buckets
```

💡 When to Use Bucketing

- High-cardinality columns
- Frequent joins on same columns
- Repeated aggregations
- Filter operations on specific values

⚡ Performance Gains

- **Eliminates shuffle** in joins and aggregations
- **Enables bucket pruning** in filters
- **Co-locates related data** on same executors
- **Reduces network I/O** significantly

Conclusion

Bucketing is a powerful optimization technique in Apache Spark that trades write-time cost for significant query-time performance gains. By understanding hash-based distribution and co-location principles, you can dramatically improve the performance of your Spark applications, especially for workloads involving frequent joins, filters, and aggregations on high-cardinality columns.

Remember: The key to successful bucketing is choosing the right columns, the right number of buckets, and using it for operations that will be executed multiple times to amortize the initial bucketing cost.

Suggested Readings:

1. https://umbertogriffo.gitbook.io/apache-spark-best-practices-and-tuning/parallelism/sparksqlshufflepartitions_draft
2. https://medium.com/globant/how-to-solve-a-large-number-of-small-files-problem-in-spark-21f819eb36d3