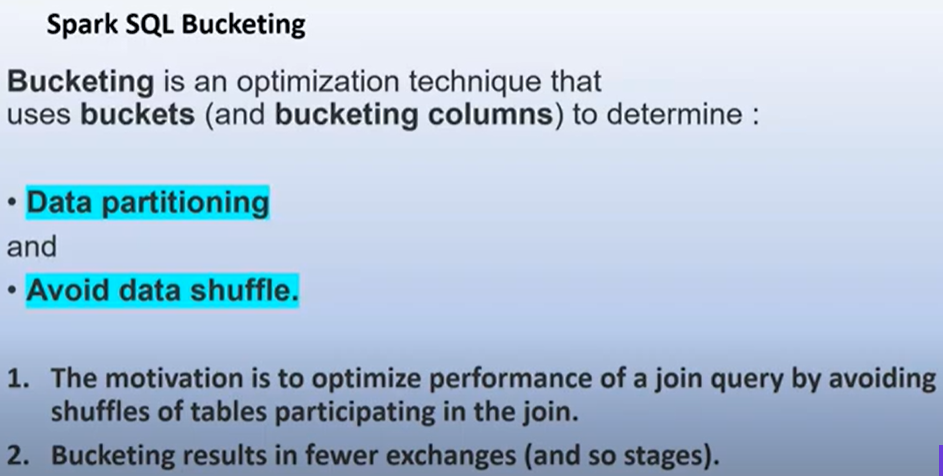
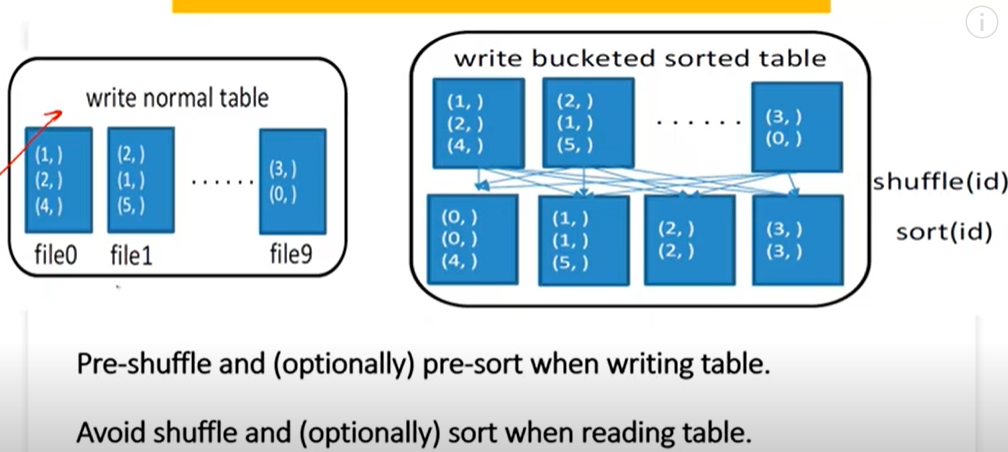
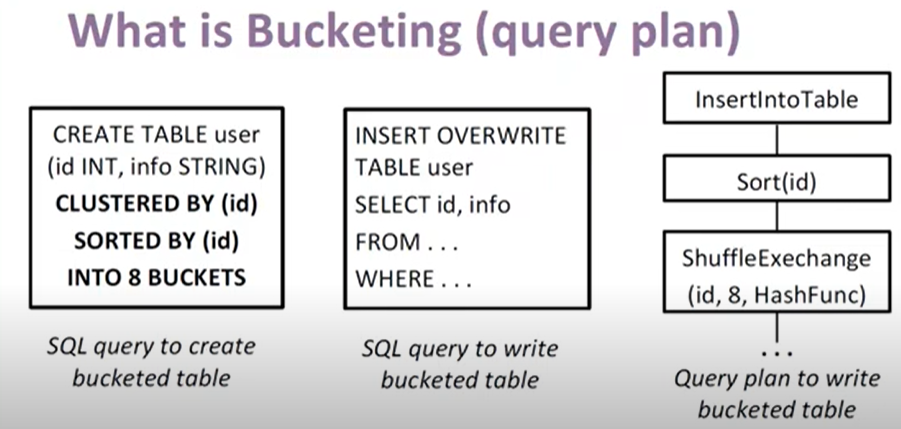
**BUCKETING IN SPARK SQL**

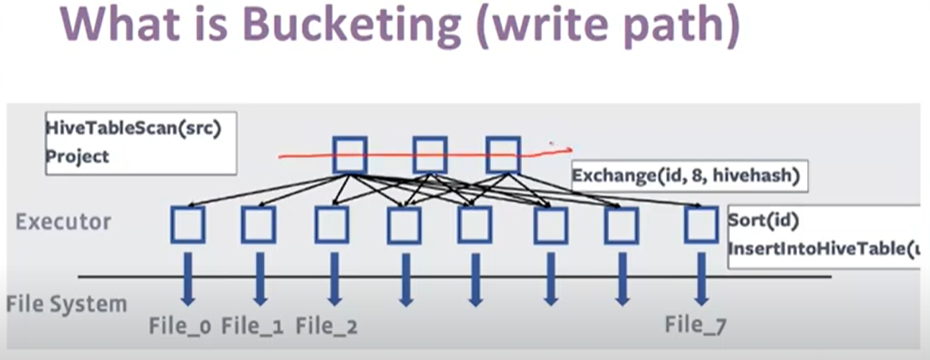
Bucketing is one of the performance optimization technique which can be leveraged to avoid the shuffle in the entire spark ecosystem. It is very good performance enhancement technique where the table is reused in a very massive way because the first time you try to write that table or data into the bucket table or you apply the bucketing first time, the time will take to sort the data or shuffle the data in different buckets but post that if you try to use that table in joins or in aggregation, it will result in no shuffling at all, lot of time will reduce.



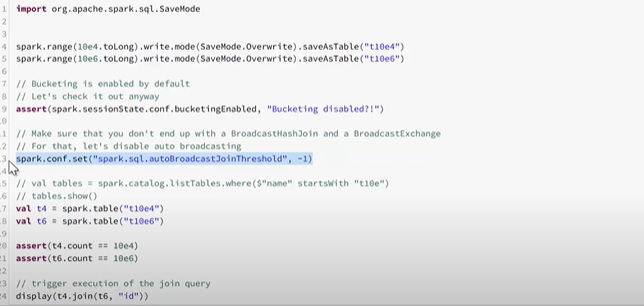




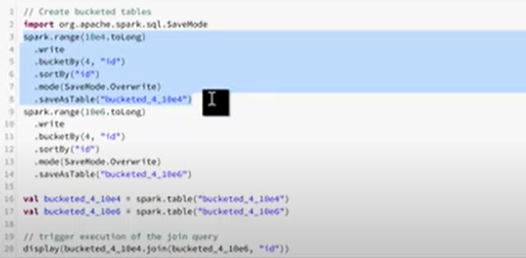




In this case no bucketing is done and exchange state is arrived as shuffing is done



By applying bucketing the exchange state not arrived as now shuffle is done here



We've got two tables and we do one simple inner join by one column:

t1 = spark.table('unbucketed1')

t2 = spark.table('unbucketed2')

t1.join(t2, 'key').explain()

In the physical plan, what you will get is something like the following:

== Physical Plan ==

\*(5) Project [key#10L, value#11, value#15]

+- \*(5) SortMergeJoin [key#10L], [key#14L], Inner

:- \*(2) Sort [key#10L ASC NULLS FIRST], false, 0

: +- Exchange hashpartitioning(key#10L, 200)

: +- \*(1) Project [key#10L, value#11]

: +- \*(1) Filter isnotnull(key#10L)

: +- \*(1) FileScan parquet default.unbucketed1[key#10L,value#11] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/opt/spark/spark-warehouse/unbucketed1], PartitionFilters: [], PushedFilters: [IsNotNull(key)], ReadSchema: struct<key:bigint,value:double>

+- \*(4) Sort [key#14L ASC NULLS FIRST], false, 0

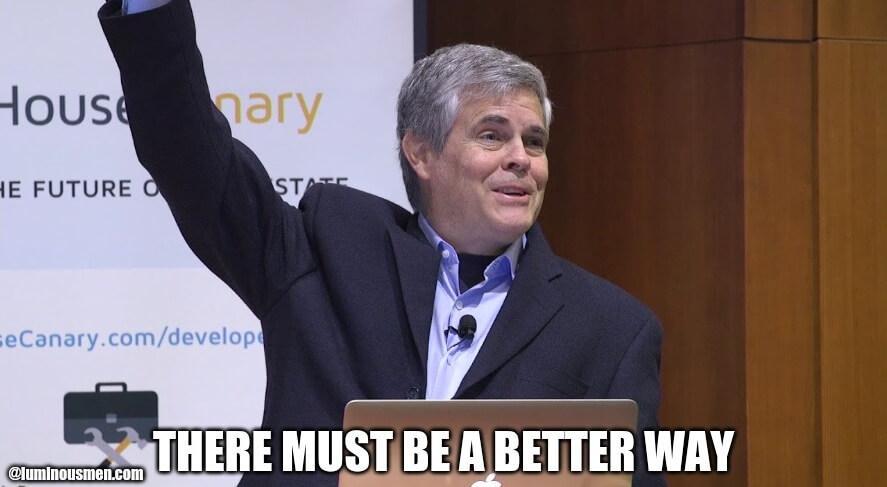
+- Exchange hashpartitioning(key#14L, 200)

+- \*(3) Project [key#14L, value#15]

+- \*(3) Filter isnotnull(key#14L)

+- \*(3) FileScan parquet default.unbucketed2[key#14L,value#15] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/opt/spark/spark-warehouse/unbucketed2], PartitionFilters: [], PushedFilters: [IsNotNull(key)], ReadSchema: struct<key:bigint,value:double>, SelectedBucketsCount: 16 out of 16

SortMergeJoin is the default Spark join, but now we're more worried about two other things in the execution plan. These are the two Exchange operations. We always worry about exchanges because they shuffle our data — we want to avoid it, well, unless we don't have a choice. But...



We know that we joined by the key column, so we will use this information to get rid of these two exchanges.

*How?*

**Use bucketing**

Bucketing is an optimization method that breaks down data into more manageable parts (buckets) to determine the data partitioning while it is written out. The motivation for this method is to make successive reads of the data more performant for downstream jobs if the SQL operators can make use of this property. In our example, we can optimize the execution of join queries by avoiding [shuffles](https://luminousmen.com/post/spark-core-concepts-explained)(also known as exchanges) of the tables involved in the join. Using bucketing leads to a smaller number of exchanges (and, consequently, stages), because shuffling may not be required — both DataFrames may already be located in the same partitions.

Bucketing is on by default. Spark uses the configuration property spark.sql.sources.bucketing.enabled to control whether or not it should be enabled and used to optimize requests.

Bucketing determines the physical layout of the data, so we shuffle the data beforehand because we want to avoid such shuffling later in the process.

*Okay, do I really need to do an extra step if the shuffle is to be executed anyway?*

If you join several times, then yes. The more times you join, the better the performance gains.

An example of how to create a bucketed table:

df.write\

.bucketBy(16, 'key') \

.sortBy('value') \

.saveAsTable('bucketed', format='parquet')

Thus, here bucketBy distributes data to a fixed number of buckets (16 in our case) and can be used when the number of unique values is not limited. If the number of unique values is limited, it's better to use a [partitioning](https://luminousmen.com/post/spark-partitions) instead of a bucketing.

t2 = spark.table('bucketed')

t3 = spark.table('bucketed')

# bucketed - bucketed join.

# Both sides have the same bucketing, and no shuffles are needed.

t3.join(t2, 'key').explain()

And the resulting physical plan:

== Physical Plan ==

\*(3) Project [key#14L, value#15, value#30]

+- \*(3) SortMergeJoin [key#14L], [key#29L], Inner

:- \*(1) Sort [key#14L ASC NULLS FIRST], false, 0

: +- \*(1) Project [key#14L, value#15]

: +- \*(1) Filter isnotnull(key#14L)

: +- \*(1) FileScan parquet default.bucketed[key#14L,value#15] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/opt/spark/spark-warehouse/bucketed], PartitionFilters: [], PushedFilters: [IsNotNull(key)], ReadSchema: struct<key:bigint,value:double>, SelectedBucketsCount: 16 out of 16

+- \*(2) Sort [key#29L ASC NULLS FIRST], false, 0

+- \*(2) Project [key#29L, value#30]

+- \*(2) Filter isnotnull(key#29L)

+- \*(2) FileScan parquet default.bucketed[key#29L,value#30] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/opt/spark-warehouse/bucketed], PartitionFilters: [], PushedFilters: [IsNotNull(key)], ReadSchema: struct<key:bigint,value:double>, SelectedBucketsCount: 16 out of 16

Here we have not only fewer code-gen stages but also no exchanges.

Apart from the single-stage sort-merge join, bucketing also supports quick data sampling. As of Spark 2.4, Spark SQL supports bucket pruning to optimize filtering on the bucketed column (by reducing the number of bucket files to scan).

**Summary**

Overall, bucketing is a relatively new technology which in some cases can be a big improvement in terms of both stability and performance. However, I found that its use is not trivial and has many drawbacks.

Bucketing works well when the number of unique values is unlimited. Columns that are often used in queries and provide high selectivity are a good choice for bucketing. Bucketed Spark tables store metadata about how they are bucketed and sorted, which helps optimize joins, aggregations, and queries for bucketed columns.