Data 603 – Big Data Platforms

Lecture 12
Optimization and Tuning

GTTs

https://forms.gle/ZwJjBQCgQ9MF2qpW8

Viewing and Setting Configurations

Three ways to get and set Spark properties:

- Through a set of configuration files
  - Within conf directory under $SPARK_HOME.
  - spark-defaults.conf, log4j.properties, spark-env.sh
  - Configuration changes in the conf/spark-defaults.conf file apply to the Spark cluster and all Spark applications submitted to the cluster.
- Specify Spark configurations directly in the Spark application or on the command line when submitting the application with spark-submit, using the --conf flag.
  - Spark config settings can be accessed using SparkSession object
- Through a programmatic interface via the Spark shell

Viewing and Setting Configurations

```
# Viewing Spark SQL-specific Spark configs
spark.sql("SET -v").select("key", "value").\
            show(n=5, truncate=False)

spark.conf.get("spark.sql.shuffle.partitions")
spark.conf.set("spark.sql.shuffle.partitions", 5)
spark.conf.get("spark.sql.shuffle.partitions")
```

Configuration Precedence

An order of precedence determines which values are honored when setting Spark properties:

1. All values or flags defined in spark-defaults.conf will be read first
2. Configs supplied on the command line after that
3. Lastly, the configs set via SparkSession in the Spark application

All the properties will be merged. The properties specified in the Spark application takes the precedence. The value specified on the command line overrides the settings in the configuration file.

Scaling for Large Workloads

Sources of job failures:

- Resource starvation
- Gradual performance degradation

Spark Configurations can affect three components:

- Spark Driver – responsible for coordinating with the cluster manager for resources and launch executors in a cluster and schedule Spark tasks to on them.
- The executor
- Shuffle service running on the executor

Dynamic Resource Allocation

- Spark provides a mechanism to dynamically adjust the resources a Spark application occupies based on the workload.
- The application may give resources back to the cluster if they are no longer used and request them again later when there is demand.
- Useful:
  - Multiple applications sharing resources in a Spark cluster.
  - Streaming, where the data flow volume may be uneven
  - Data analytics, where there is high traffic of SQL queries during peak hours.

Dynamic Resource Allocation

- `spark.dynamicAllocation.enabled`
  - Whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload.
  - By default, `spark.dynamicAllocation.enabled` is false.
- `spark.dynamicAllocation.minExecutors`
  - Lower bound for the number of executors if dynamic allocation is enabled.

Dynamic Resource Allocation

- `spark.dynamicAllocation.schedulerBacklogTimeout`
  - If dynamic allocation is enabled and there have been pending tasks backlogged for more than this duration, new executors will be requested.
- `spark.dynamicAllocation.maxExecutors`
  - Upper bound for the number of executors if dynamic allocation is enabled.
- `spark.dynamicAllocation.executorIdleTimeout`
  - If dynamic allocation is enabled and an executor has been idle for more than this duration, the executor will be removed

Executor's memory and shuffle service

- `spark.executor.memory` – determines the amount of memory available to each executor.
- This is divided into three sections:
  - Execution memory (default: 60%), used for Spark shuffles, joins, sorts and aggregations.
  - Storage memory (default: 40%), primarily used for caching user data structures and partitions derived from DataFrames.
  - Reserve memory (default: 300 MB), safeguard against OOM errors.
- When storage memory is not being used, Spark can take and use it in execution memory for execution purposes, and vice versa.

Executor's memory and shuffle service

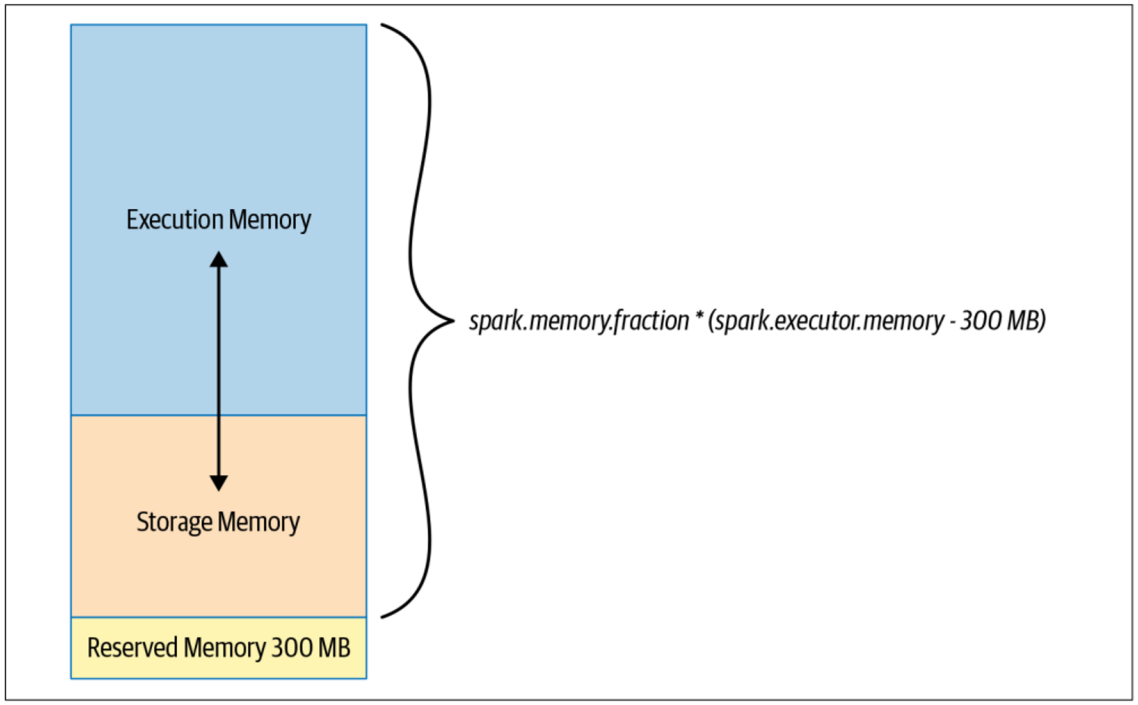
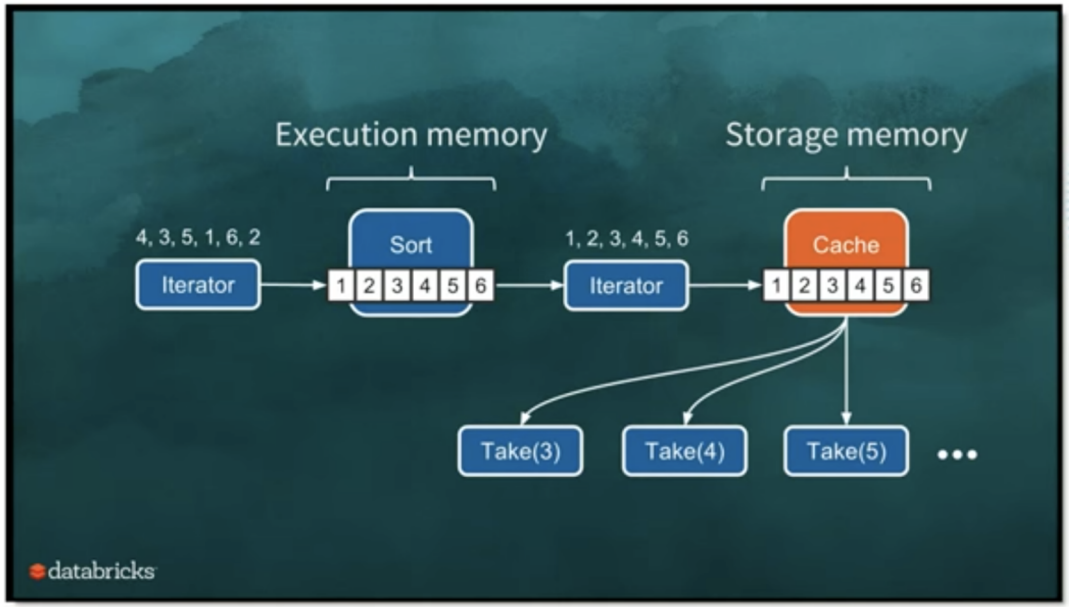


Figure 7-2. Executor memory layout

Executor's memory and shuffle service



SPARK SUMMIT 2016

Deep Dive: Apache Spark Memory Management:
https://www.youtube.com/watch?v=dPHrykZL8Cg

Executor's memory and shuffle service

- During map and shuffle operations, Spark writes to and reads from the local disk's shuffle files
  - Heavy I/O activity
  - Results in a bottleneck
  - Default configurations are suboptimal for large-scale Spark jobs.
- Knowing what configurations to tweak can mitigate this risk during this phase of a Spark job
- Tuning the shuffle service running on each executor can increase overall performance for large Spark workloads

Executor's memory and shuffle service

Table 7-1. Spark configurations to tweak for I/O during map and shuffle operations

Configuration	Default value, recommendation, and description
`spark.driver.memory`	Default is `1g` (1 GB). This is the amount of memory allocated to the Spark driver to receive data from executors. This is often changed during `spark-submit` with `--driver-memory`. Only change this if you expect the driver to receive large amounts of data back from operations like `collect()`, or if you run out of driver memory.
`spark.shuffle.file.buffer`	Default is 32 KB. Recommended is 1 MB. This allows Spark to do more buffering before writing final map results to disk.
`spark.file.transferTo`	Default is `true`. Setting it to `false` will force Spark to use the file buffer to transfer files before finally writing to disk; this will decrease the I/O activity.
`spark.shuffle.unsafe.file.output.buffer`	Default is 32 KB. This controls the amount of buffering possible when merging files during shuffle operations. In general, large values (e.g., 1 MB) are more appropriate for larger workloads, whereas the default can work for smaller workloads.
`spark.io.compression.lz4.blockSize`	Default is 32 KB. Increase to 512 KB. You can decrease the size of the shuffle file by increasing the compressed size of the block.
`spark.shuffle.service.index.cache.size`	Default is 100m. Cache entries are limited to the specified memory footprint in byte.
`spark.shuffle.registration.timeout`	Default is 5000 ms. Increase to 120000 ms.
`spark.shuffle.registration.maxAttempts`	Default is 3. Increase to 5 if needed.

Maximizing Spark parallelism

- Read and process as much data in parallel as possible
  - Need to understand how Spark reads data into memory from storage and what partitions mean to Spark
- A partition is a way to arrange data into a subset of configurable and readable chunks or blocks of contiguous data on disk.
  - Partitions of data can be read or processed independently and in parallel by more than a single thread in a process
  - This allows for massive parallelism of data processing
  - In best scenario, Spark schedules a thread per task task per core, and each thread processes a distinct partition.
  - For optimizing resource utilization and maximum parallelism, ideally there are as many partitions as there are cores on the executor.

Maximizing Spark parallelism

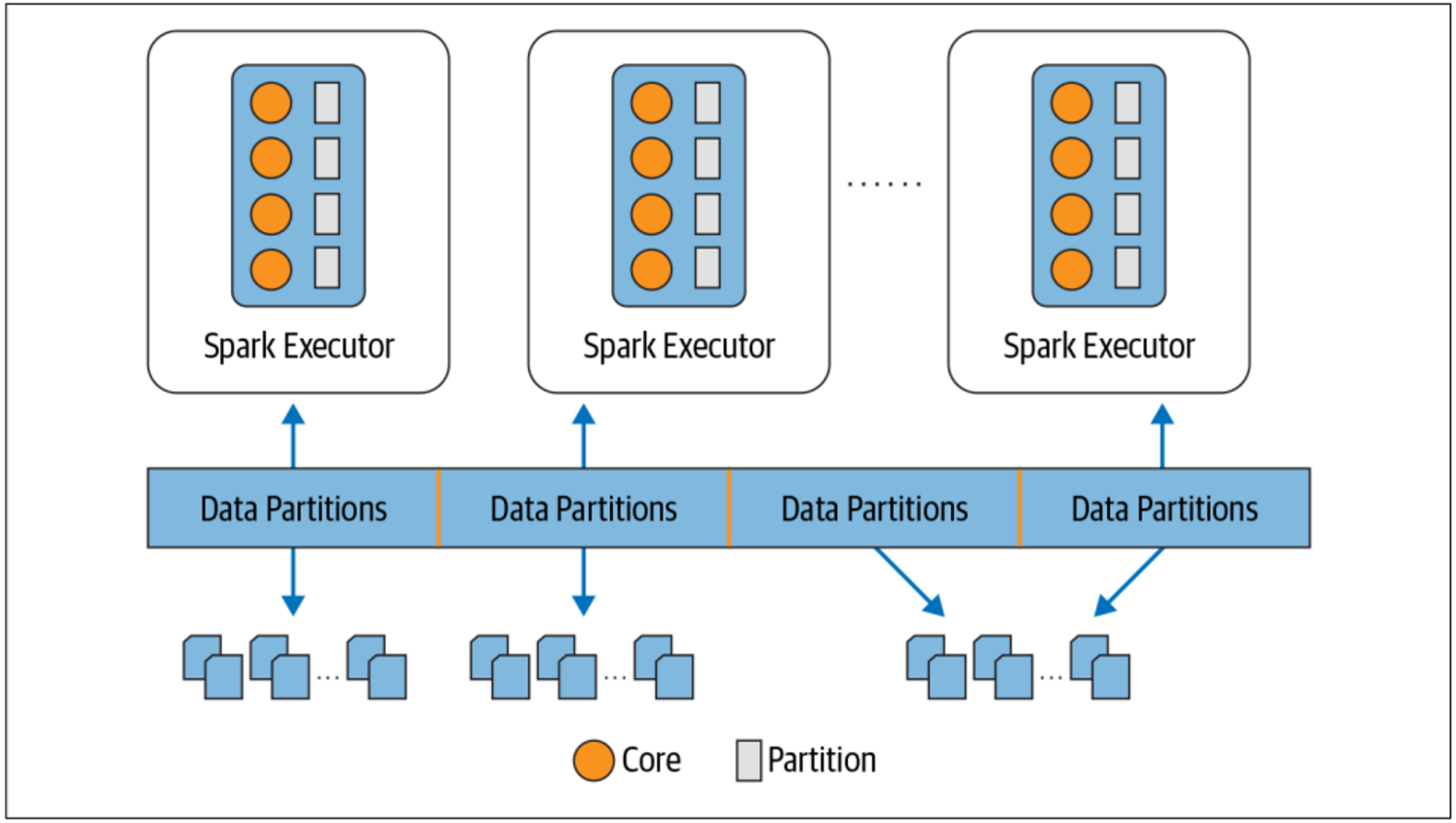


Figure 7-3. Relationship of Spark tasks, cores, partitions, and parallelism

Maximizing Spark parallelism

- Spark's tasks process data as partitions read from disk into memory.
- Data on disk is laid out in chunks or contiguous file blocks (depending on the store)
  - By default: file blocks on data stores range in size from 64 MB to 128 MB. (e.g. HDFS and S3 have default size of 128 MB)
  - The file block size is configurable
  - A contiguous collection of the blocks constitute a partition.
- The size of a partition in Spark is dictated by `spark.sql.files.maxPartitionBytes` (default 128 MB).
  - Decreasing the size can cause "small file problem" caused by file system operations (opening, closing, listing directories) which on a distributed system can be slow.

Shuffle Partitions

- Partitions are also created explicitly when certain methods of the DataFrame API are used.
  - When creating a large DataFrame or reading a large file from disk, Spark can be instructed to create a certain number of partitions. e.g `repartition(16)`
  - During operations like groupBy() or join(), known as wide transformations
    - The shuffle spills results to executor's local disks (`spark.local.directory`).
  - Shuffle partitions are created during the shuffle stage.
    - By default, the number of shuffle partitions is set to 200 (`spark.sql.shuffle.partitions`). This parameter can be adjusted depending on the size of the data set to reduce the amount of small partitions being sent across the network to executor's tasks.
    - The default value for spark.sql.shuffle.partitions is too high for smaller or streaming workloads. Reduce it to a lower value (# of cores on the executors or less).
- Shuffling partitions consume both network and disk I/O resources

Shuffle Partitions

- No magic formula for the number of shuffle partitions to set for the shuffle stage
- The number may vary depending on:
  - Use case
  - Data set
  - Number of cores
  - The amount of executor memory available
- This is a trial-and-error approach
- Consider caching or persisting frequently accessed DataFrames or tables.

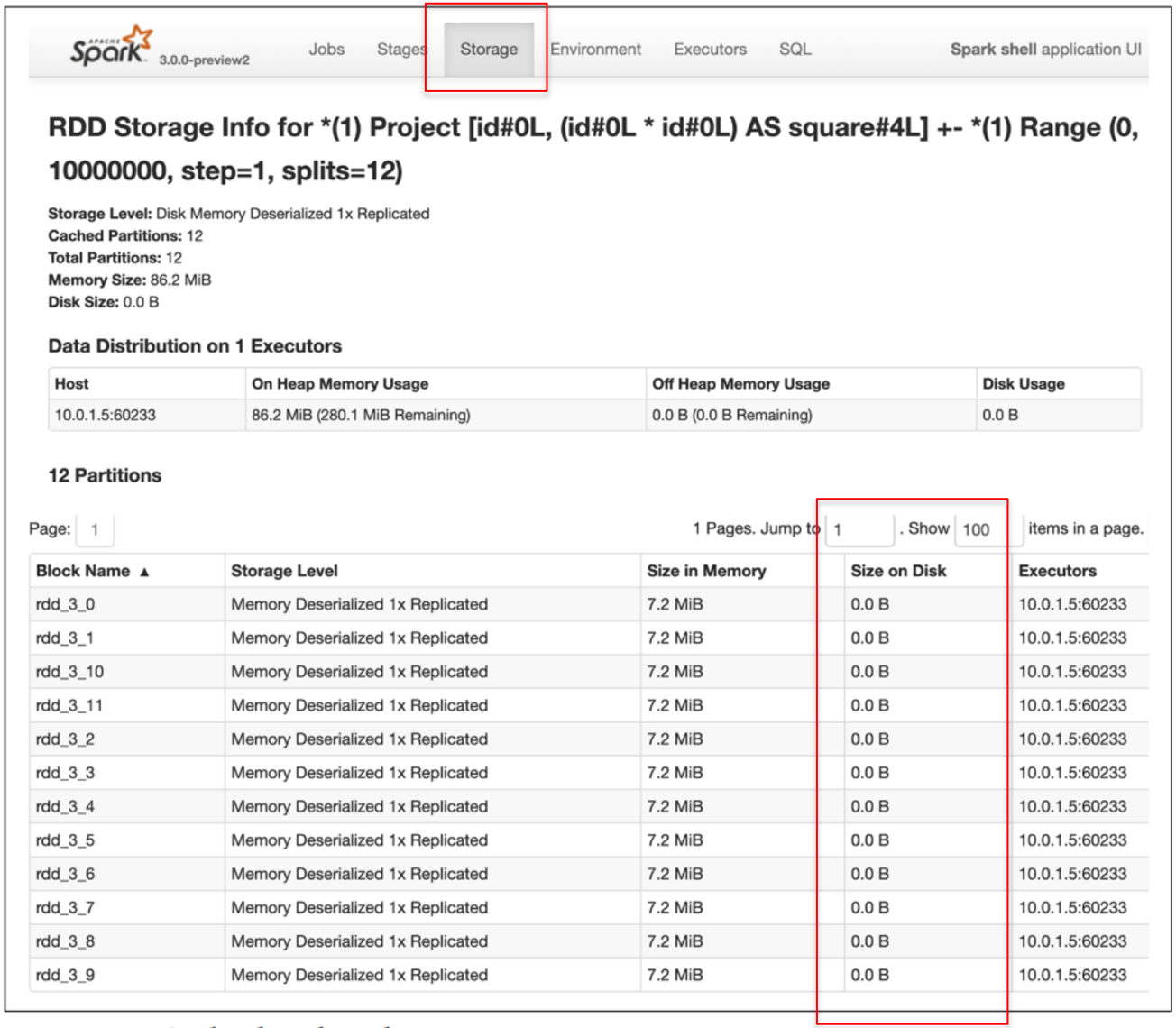
Caching and Persisting Data

- `cache()` and `persist()` are synonymous in Spark
  - `persist()` provides more control over how and where the data is stored (in memory/disk, serialized/unserialized).
- `DataFrame.cache()`
  - Store as many of the partitions read in memory across Spark executors as memory allows
    - DataFrames may be fractionally cached. Partitions cannot be fractionally cached
    - E.g. If there are 8 partitions, and the memory has space for only 4.5 partition, then only. 4 partitions will be cached.
    - If not all partitions cached, the partitions that are not cached will have to be recomputed when accessed, causing slow down on the Spark job

Caching and Persisting Data

```
// In Scala
// Create a DataFrame with 10M records
val df = spark.
    range(1 * 10000000).
    toDF("id").
    withColumn("square", $"id" * $"id")
df.cache() // Cache the data
df.count() // Materialize the cache
res3: Long = 10000000
Command took 5.11 seconds
df.count() // Now get it from the cache
res4: Long = 10000000
Command took 0.44 seconds
```

- DataFrame is not fully cached until you invoke an action that goes through every record (e.g., count()).
- Action like take(1) will cache only one partition because Catalyst realizes that you do not need to compute all the partitions just to retrieve one record.



Spark 3.0.0-preview2 Jobs Stages Storage Environment Executors SQL **Spark shell** application UI

RDD Storage Info for *(1) Project [id#0L, (id#0L * id#0L) AS square#4L] +- *(1) Range (0, 10000000, step=1, splits=12)

Storage Level: Disk Memory Deserialized 1x Replicated
Cached Partitions: 12
Total Partitions: 12
Memory Size: 86.2 MiB
Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
10.0.1.5:60233	86.2 MiB (280.1 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

12 Partitions

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page.

Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_3_0	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_1	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_10	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_11	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_2	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_3	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_4	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_5	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_6	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_7	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_8	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_9	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233

Figure 7-4. Cache distributed across 12 partitions in executor memory

Caching and Persisting Data

- `DataFrame.persist()`
  - Provides more control over how and where the data is stored (memory/disk, serialized/unserialized)
  - persist(StorageLevel.LEVEL)

Caching and Persisting Data

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Note: *In Python, stored objects will always be serialized with the Pickle library, so it does not matter whether you choose a serialized level. The available storage levels in Python include* `MEMORY_ONLY`, `MEMORY_ONLY_2`, `MEMORY_AND_DISK`, `MEMORY_AND_DISK_2`, `DISK_ONLY`, `DISK_ONLY_2`, *and* `DISK_ONLY_3`.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

Caching and Persisting Data

- `DataFrame.persist(StorageLevel.LEVEL)`
  - Provides more control over how and where the data is stored (memory/disk, serialized/unserialized)
- Each StorageLevel (except OFF_HEAP) has an equivalent LEVEL_NAME_2.
  - Replicate twice on different Spark executors.
  - Cons: can be expensive
  - Pros: allows data locality in two places
    - Provides fault tolerance
    - Gives Spark the option to schedule a task local to a copy of the data.

Caching and Persisting Data

- Tables and views derived from DataFrames can also be cached.

```
df.createOrReplaceTempView("dfTable")
spark.sql("CACHE TABLE dfTable")
```

When to Cache and Persist

When to cache and persist

- Accessing a large data set repeatedly
  - DataFrames commonly used during iterative machine learning training
  - DataFrames commonly accessed for doing frequent transformations during ETL or building data pipelines

When not to cache and persist

- DataFrames that are too big to fit in memory
- An inexpensive transformation on a DataFrame not requiring frequent use, regardless of size

General Rule

- Use memory caching judiciously
  - Can incur resource costs in serializing and deserializing (depends on the StorageLevel used)

Data Serialization

Serialization plays an important role in the performance of distributed application. It has a direct impact on the speed of computation.

Spark provides two serialization libraries:

- Java serialization: By default, Spark serializes objects using Java's ObjectOutputStream framework. Java serialization is slower and results in large serialized formats for many classes.
- Kryo serialization: Significantly faster and more compact than Java serialization (often as much as 10x)
- Does not support all Serializable types.

Data Serialization

Switch to using Kryo by initializing your job with a SparkConf and calling conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer").

- This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk.

Family of Spark Joins

- Need to be mindful of join operations that trigger expensive movement of data
  - Increased demand on compute and network resources from the cluster
  - Re-organizing data can alleviate data movement
- Join operations are a common type of transformation
- Spark SQL offers join transformations
  - inner joins, outer joins, left joins, outer joins, etc.
  - All of the join operations trigger a large amount of data movement across Spark executors

Family of Spark Joins

- During join transformations, Spark computes
  - What data to produce
  - What keys and associated data to write to the disk
  - How to transfer those keys and data to nodes as part of operations like `groupBy()`, `join()`, `agg()`, `sortBy()`, and `reduceByKey()`; referred to as the *shuffle*.
- Spark's five distinct join strategies for exchanging, moving, sorting, grouping, and merging data across executors:
  - **broadcast hash join (BHJ)**, shuffle hash join (SHJ), **shuffle sort merge join (SMJ)**, broadcast nested loop join (BNLJ) and shuffle-and-replicate nested loop join (Cartesian product join)
  - BHJ and SMJ are most common ones

Family of Spark Joins - BHJ

Broadcast Hash Join

- Also known as a *map-side-only join*.
- Employed when two data sets need to be joined over certain conditions or columns:
  - One small data set fitting in the driver's and executors memory
  - Another large enough data set that can be spared from movement
- Using a Spark broadcast variable, the smaller data set is broadcasted by the driver to all Spark executors which is joined with the larger data set on each executor
- By default Spark uses BHJ if the smaller data set is less than 10 MB (configured via `spark.sql.autoBroadcastJoinThreshold`)
- Common use case: when there is a common key between two DataFrames

Family of Spark Joins - BHJ

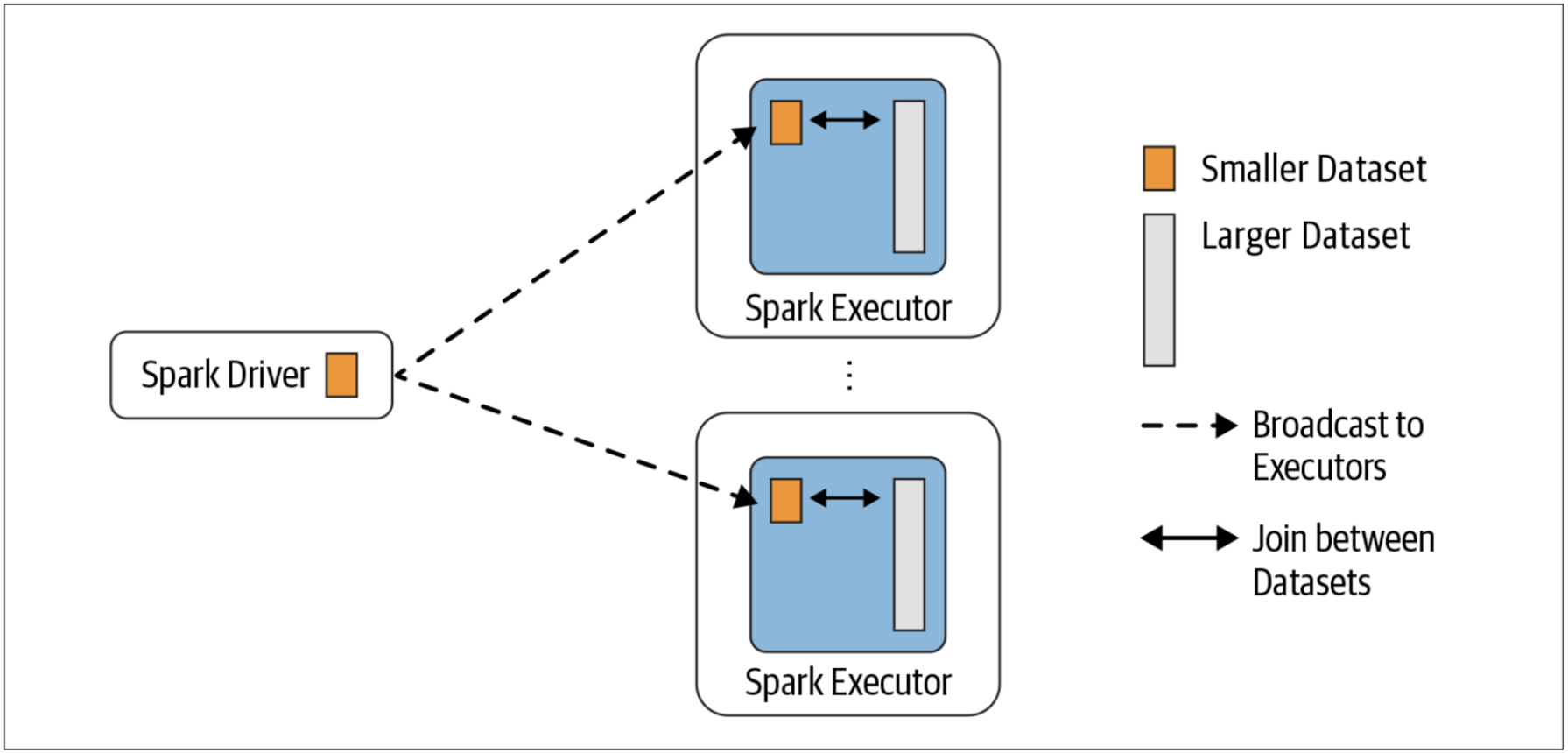


Figure 7-6. BHJ: the smaller data set is broadcast to all executors

Family of Spark Joins - BHJ

```
// In Scala
import org.apache.spark.sql.functions.broadcast
val joinedDF = playersDF.join(broadcast(clubsDF),
        "key1 === key2")
```

Code above forces Spark to do a broadcast join.

- It will resort to this type of join by default if the size of the smaller data set is below the `spark.sql.autoBroadcastJoinThreshold`.
- BHJ is the easiest and fastest join.
  - Does not involve any shuffle of the data set
  - All the data is available locally to the executor after a broadcast
  - Need to make sure that there is enough memory on the driver and the executors.

Family of Spark Joins - BHJ

Use BHJ when

- Each key within the smaller and larger data sets is hashed to the same partition by Spark
- One data set is much smaller than the other (and within the default config of 10 MB, or more if you have sufficient memory)
- You only want to perform an equi-join, to combine two data sets based on matching unsorted keys
- You are not worried by excessive network bandwidth usage or OOM errors, because the smaller data set will be broadcast to all Spark executors

Note: Specifying a value of -1 in spark.sql.autoBroadcastJoinThreshold will cause Spark to always resort to a shuffle sort merge join,

Family of Spark Joins - SMJ

- The sort-merge (or Shuffle Sort Merge Join) algorithm is an efficient way to merge two large data sets over a common key that is sortable, unique, and can be assigned to or stored in the same partition (two data sets with a common hashable key that ends up being on the same partition).
  - All rows within each data set with the same key are hashed on the same partition on the same executor.
  - This requires data to be collocated or exchanged between executors.
- Has two phases:
  1. Sort phase – Sorts each data set by its desired join key.
  2. Merge phase – Iterates over each key in the row from each data set and merges rows if two keys match
- By default, the SortMergeJoin is enabled via `spark.sql.join.preferSortMergeJoin`.

Family of Spark Joins - SMJ

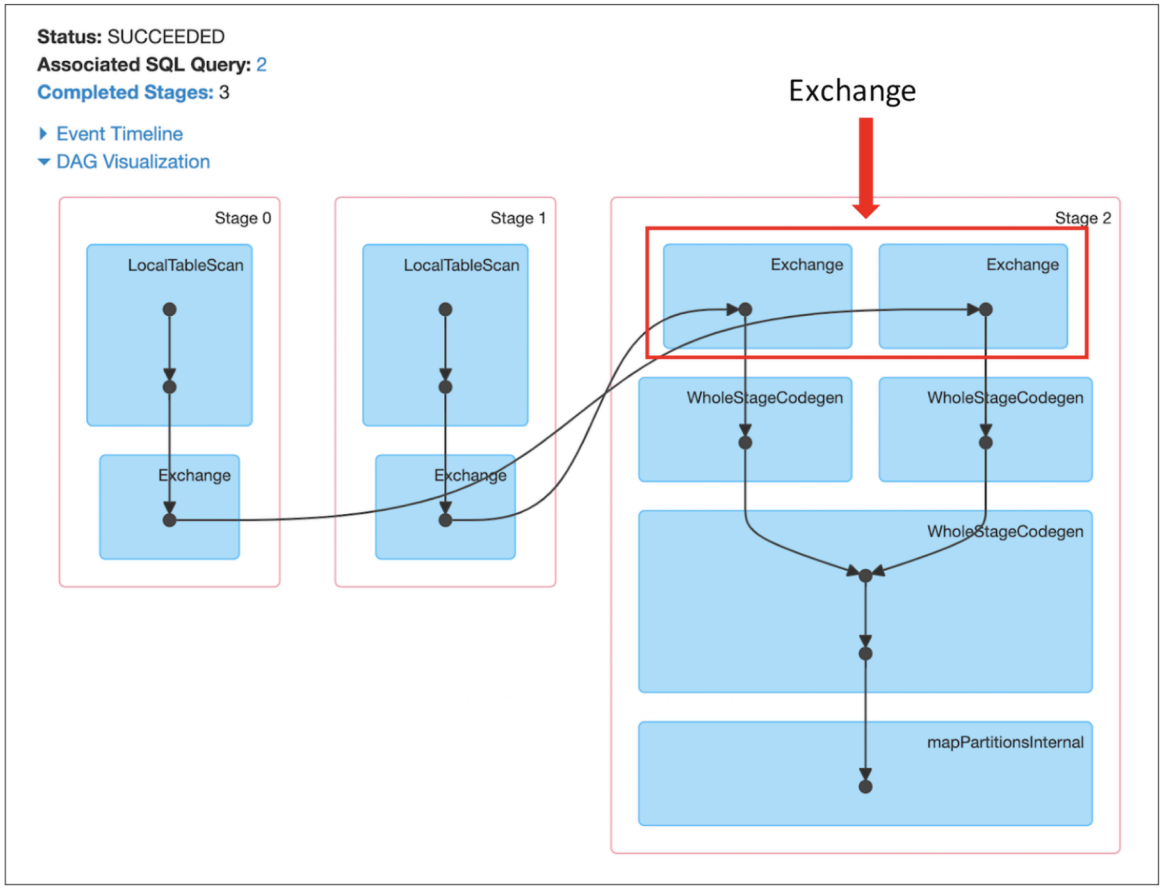


Figure 7-7. Before bucketing: stages of the Spark

The Exchange is expensive and requires partitions to be shuffled across the network between executors.

Family of Spark Joins - SMJ

Optimizing the shuffle sort merge join

- Within SMJ, the Exchange (shuffle) step can be eliminated
  - Create partitioned buckets for common sorted keys or columns on which to perform frequent equi-joins.
  - Explicit number of buckets can be created to store specific sorted columns (one key per bucket).
  - Presorting and reorganizing data in this way can boost performance.

Family of Spark Joins - SMJ

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SaveMode
// Save as managed tables by bucketing them in Parquet format
usersDF.orderBy(asc("uid"))
    .write.format("parquet")
    .bucketBy(8, "uid")
    .mode(SaveMode.OverWrite)
    .saveAsTable("UsersTbl")

ordersDF.orderBy(asc("users_id"))
    .write.format("parquet")
    .bucketBy(8, "users_id")
    .mode(SaveMode.OverWrite)
    .saveAsTable("OrdersTbl")

spark.sql("CACHE TABLE UsersTbl")
spark.sql("CACHE TABLE OrdersTbl")
val usersBucketDF = spark.table("UsersTbl")
val ordersBucketDF = spark.table("OrdersTbl")
val joinUsersOrdersBucketDF = ordersBucketDF
.join(usersBucketDF, $"users_id" === $"uid")
joinUsersOrdersBucketDF.show(false)
```

Family of Spark Joins - SMJ

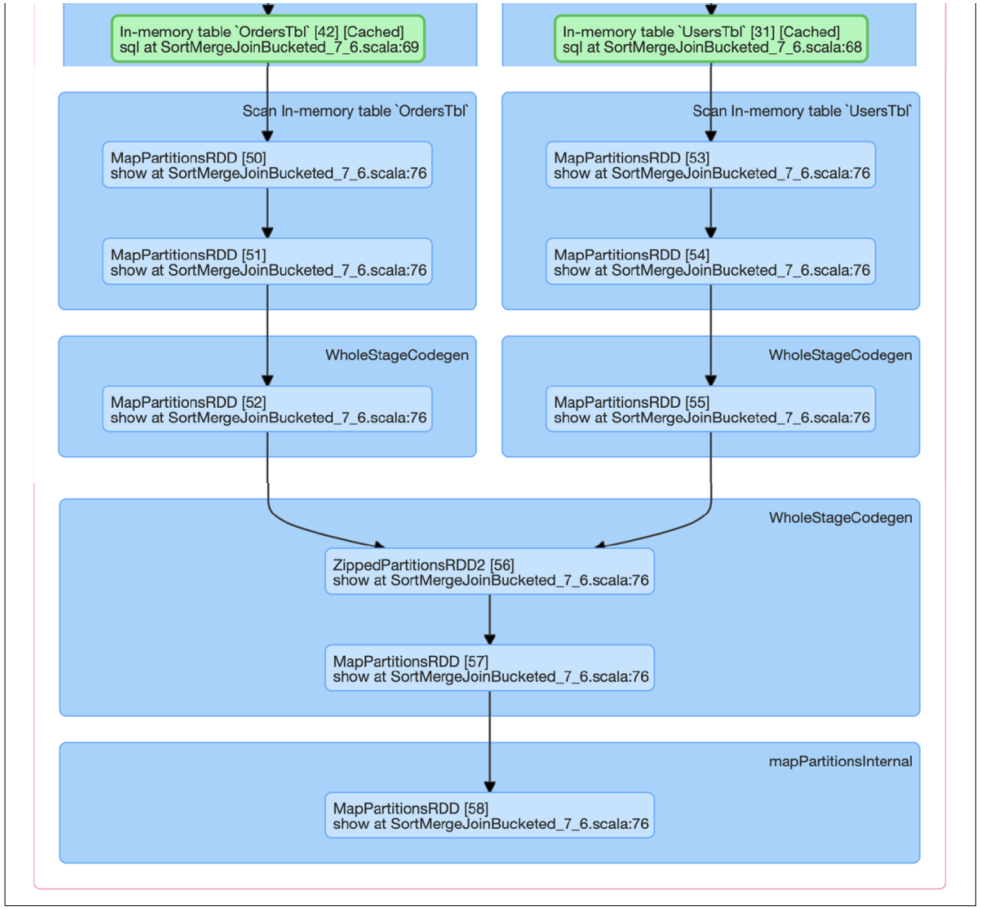


Figure 7-9. After bucketing: Exchange is not required

Family of Spark Joins - SMJ

When to use a shuffle sort merge join

- When each key within two large data sets can be sorted and hashed to the same partition by Spark
- When there is a need to perform only equi-joins to combine two data sets based on matching sorted keys
- When Exchange and Sort operations which cause large shuffles across the network need to be prevented.

Inspecting the Spark UI

https://spark.apache.org/docs/3.0.1/web-ui.html

Further Reading

- Tuning Apache Spark for Large Scale Workloads: https://www.youtube.com/watch?v=5dga0UT4RI8
- Hive Bucketing in Apache Spark: https://www.youtube.com/watch?v=6BD-Vv-ViBw&=645s
- Spark + Parquet In Depth: https://www.youtube.com/watch?v=_0Wpwj_gvzg
- Why You Should Care about Data Layout in the Filesystem: https://databricks.com/session/why-you-should-care-about-data-layout-in-the-filesystem
- Deep Dive: Apache Spark Memory Management: https://www.youtube.com/watch?v=dPHrykZL8Cg
- Decoding Memory in Spark: https://medium.com/walmartglobaltech/decoding-memory-in-spark-parameters-that-are-often-confused-c11be7488a24
- Bucketing in Spark SQL 2.3: https://databricks.com/session/bucketing-in-spark-sql-2-3
- Bucketing 2.0: Improve Spark SQL Performance by Removing Shuffle: https://databricks.com/session_na20/bucketing-2-0-improve-spark-sql-performance-by-removing-shuffle
- The Parquet Format and Performance Optimization Opportunities: https://www.youtube.com/watch?v=1j8SdS7s_NY

Lots of Links

- https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation
- https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation
- https://towardsdatascience.com/how-does-facebook-tune-apache-spark-for-large-scale-workloads-3238ddda0830
- https://docs.databricks.com/delta/data-transformation/index.html
- https://databricks.com/session/which-data-broke-my-code-inspecting-spark-transformations
- https://databricks.com/session/why-you-should-care-about-data-layout-in-the-filesystem
- https://mungingdata.com/apache-spark/partitionby/
- https://0x0fff.com/spark-memory-management/
- https://dzone.com/articles/accumulator-vs-broadcast-variables-in-spark
- https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-bucketing.html
- https://www.slideshare.net/databricks/bucketing-20-improve-spark-sql-performance-by-removing-shuffle
- https://www.youtube.com/watch?v=6BD-Vv-ViBw
- https://spark.apache.org/docs/latest/tuning.html
- https://spark.apache.org/docs/latest/configuration.html

Questions