



Optimizing Spark Job

Dano Lee

Apache Spark Optimization

- ❖ computation efficiency
- ❖ the communication between nodes.

Optimizing Spark Job

RDD over DataFrames?

- ❖ Because of their higher-level API and optimizations, DataFrames are typically easier to use and offer better performance;
- ❖ however, due to their lower-level nature, RDDs can still be useful for defining custom operations, as well as debugging complex data processing tasks.
- ❖ RDDs offer more granular control over partitioning and memory usage.
- ❖ **When dealing with raw, unstructured data, such as text streams, binary files, or custom formats, RDDs can be more flexible, allowing for custom parsing and manipulation in the absence of a predefined structure.**

Use an efficient file format and compression type

- ❖ The file format and compression type that you use can have a significant impact on the performance of shuffling.
- ❖ Using a file format that is optimized for Spark, such as **Apache Parquet** or **Apache ORC**, and a compression type that is efficient for shuffling, such as **Snappy** or **LZ4** helps to minimize shuffling and improve the performance of your Spark applications.

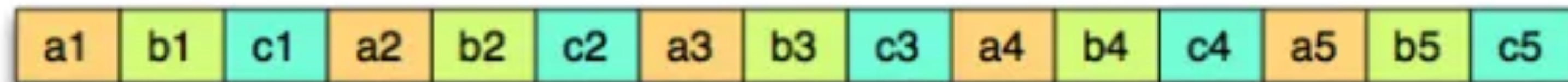
Row vs Column

Logical Table Representation

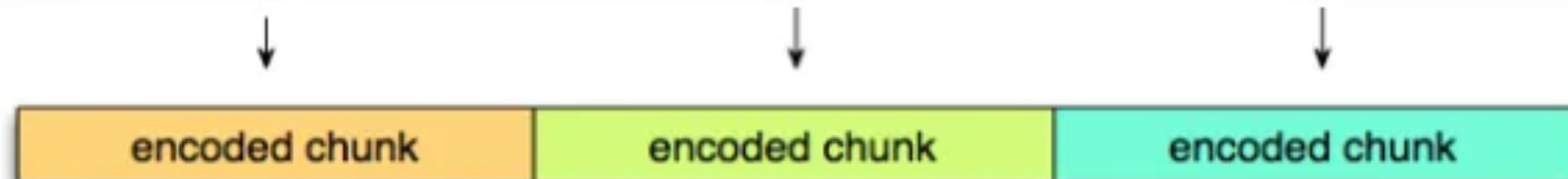
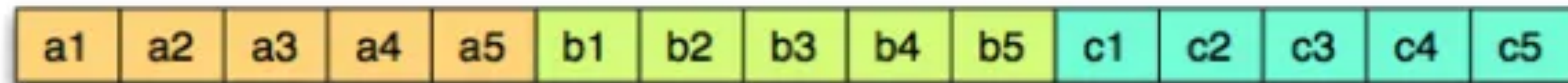
a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Physical Table Representation

Row layout



Column layout



Advantages of Columnar Storage Over Row-Based Storage:

- ❖ **Compression Efficiency:** Columnar storage allows for better compression techniques. Since column values are often of the same type, compression algorithms can be tailored to that type, resulting in higher compression ratios. This reduces storage requirements and speeds up data transfers.
- ❖ **Column Pruning:** When executing queries, columnar databases can skip irrelevant columns, reducing I/O and improving query performance. In row-based storage, entire rows must be read, even if only a few columns are needed.
- ❖ **Aggregation Performance:** Columnar storage is highly efficient for aggregate queries, as aggregations involve operations on single columns. This leads to faster query performance for analytics tasks.
- ❖ **Predicate Pushdown:** Columnar databases can apply filters early in the query execution process by analyzing metadata, minimizing the amount of data read from storage. This feature significantly speeds up query processing.
- ❖ **Analytics and Data Warehousing:** Columnar storage is well-suited for analytical workloads, reporting, and data warehousing. It allows for rapid analysis and reporting on large datasets.
- ❖ **Schema Evolution:** Columnar storage formats like Parquet support schema evolution, enabling the addition of new columns or changes to existing columns without disrupting existing data.

Apache Parquet vs Other Formats

- ❖ Parquet is optimized for the Write Once Read Many (WORM) paradigm. It's slow to write, but incredibly fast to read, **especially when you're only accessing a subset of the total columns.**
- ❖ For use cases requiring operating on **entire rows of data**, a format like CSV, JSON or even AVRO should be used.

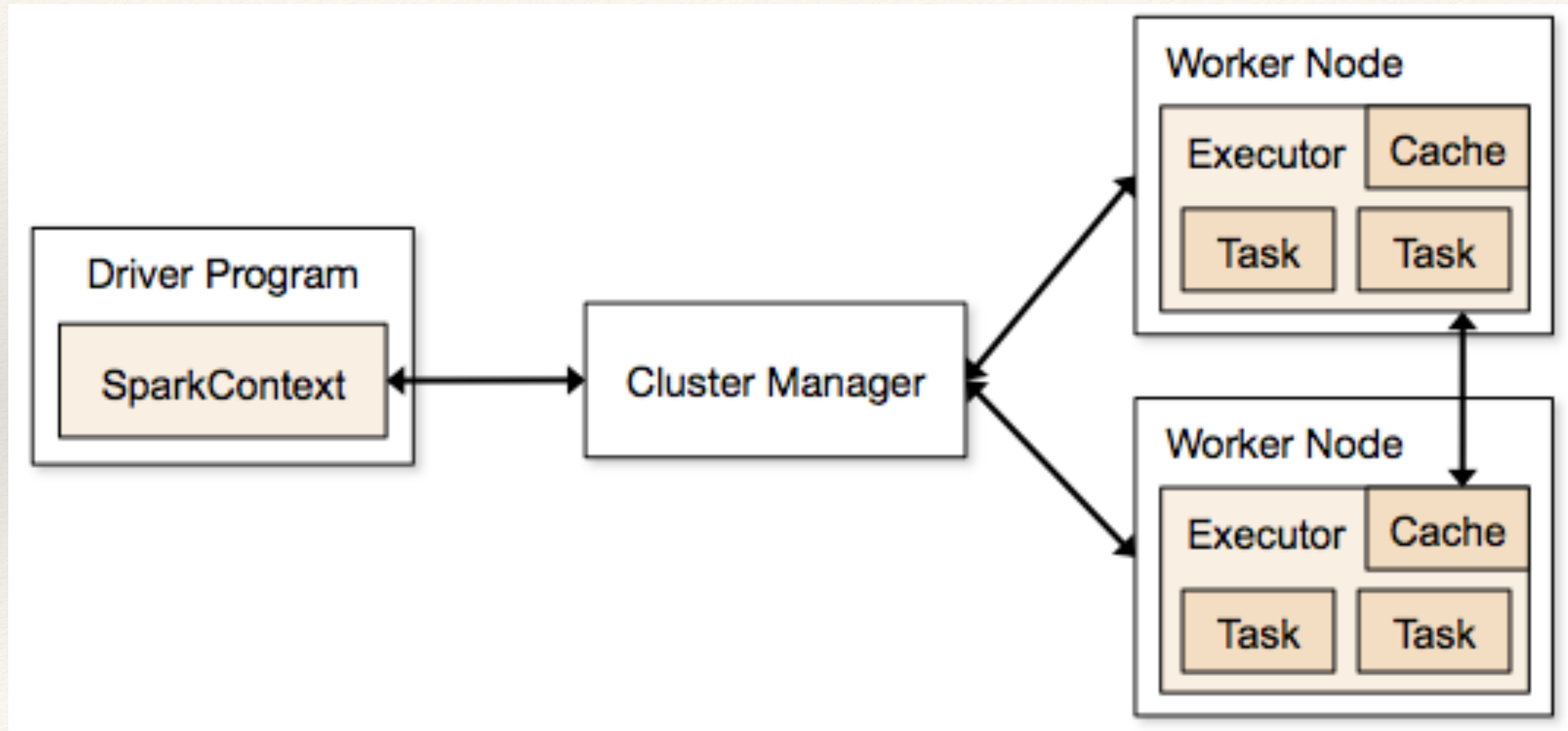
Apache Parquet vs Other Formats

Spark Format Showdown		File Format		
		<u>CSV</u>	<u>JSON</u>	<u>Parquet</u>
A t t r i b u t e	Columnar	No	No	Yes
	Compressable	Yes	Yes	Yes
	Splittable	Yes*	Yes**	Yes
	Human Readable	Yes	Yes	No
	Nestable	No	Yes	Yes
	Complex Data Structures	No	Yes	Yes
	Default Schema: Named columns	Manual	Automatic (full read)	Automatic (instant)
	Default Schema: Data Types	Manual (full read)	Automatic (full read)	Automatic (instant)

Apache Spark and JVM

- ❖ Spark is predominantly in Scala and runs on **Java virtual machines (JVMs)**

Spark is not really in-memory tool, it just utilizes the memory for its LRU cache

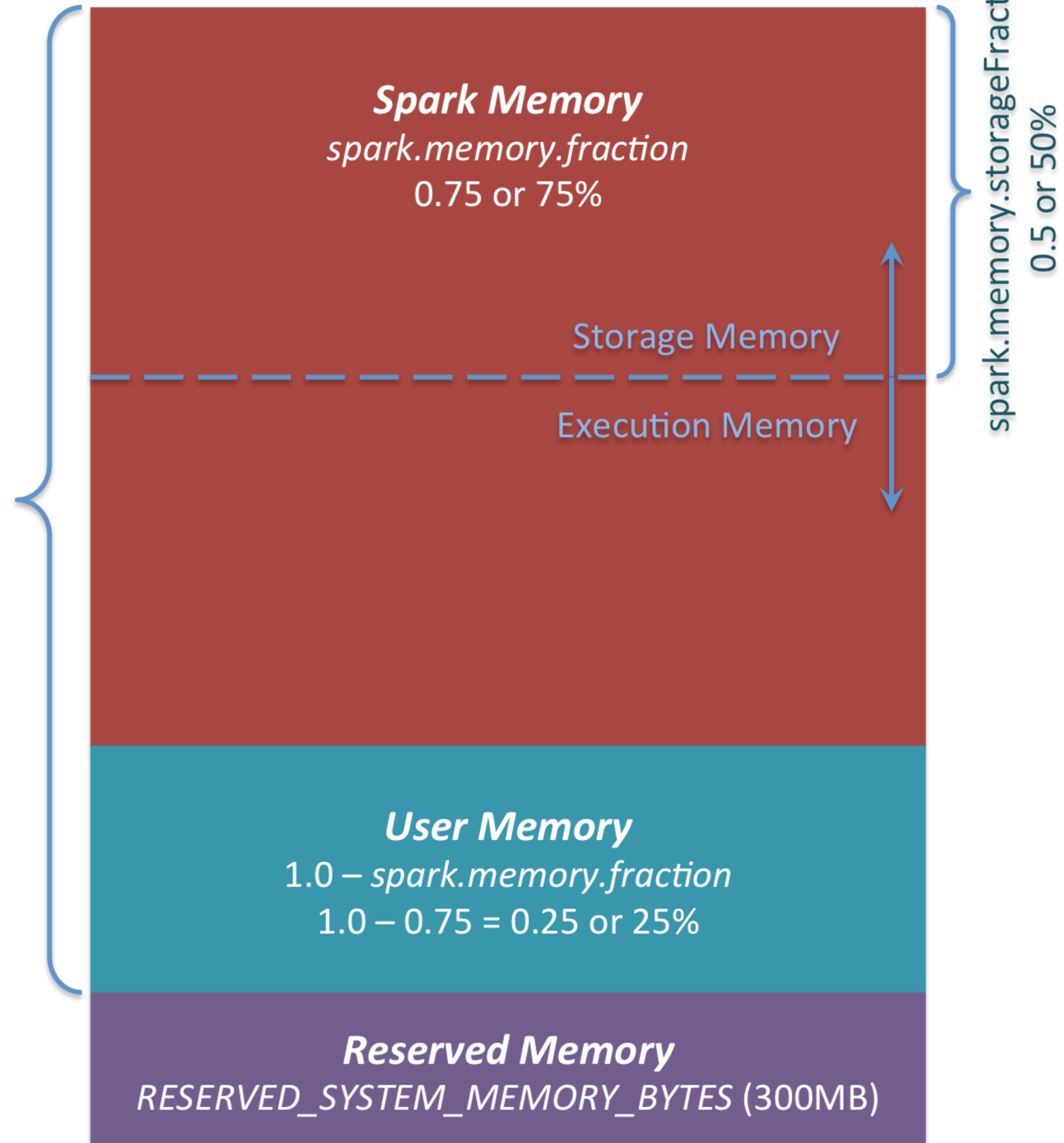


LRU Cache

- ❖ **Least Recently Used Cache**

- ❖ Picks the data that is least recently used and removes it in order to make space for the new data.
- ❖ The priority of the data in the cache changes according to the need of that data

Java Heap – Reserved Memory



Use SparkSQL

- ❖ Use SparkSQL to express your queries more effectively if you need to filter, group, or aggregate data. Most of the time, SparkSQL can reduce shuffling and improve the execution of these queries.
- ❖ Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data.

Shuffle in Spark

- ❖ **In order that spark ensures that all the records with the same key are on the same node**, Spark needs to shuffle the data if performing operations like groupBy and joins on a large dataset.
- ❖ This makes it possible to process all the records at once and combine the results.
- ❖ The shuffle operation must be finished before the next stage of processing can start, which can also delay the processing of the data.

Data shuffling is a performance killer

- ❖ costly process as it leads to network data transfer by data travel between nodes, High disk I/O and Rearranging files operations.
- ❖ The performance of Spark applications can then be enhanced by being aware of these elements and avoiding shuffling whenever possible.

Causes of Shuffle

❖ **Data skew**

- ❖ Spark shuffling may take place when some keys in a dataset are significantly more heavily populated with data than others. Data skew is the term used to describe this.
- ❖ Operations like groupByKey, reduceByKey or same sort of joins which call for the grouping or aggregation of data by key, can also cause data shuffling. But not only, the shuffling can be caused by different other factors like :

❖ **Partitioning:**

- ❖ Spark divides data among nodes through a procedure known as partitioning. The possibility of shuffling exists if the data is not distributed among partitions equally.

❖ **Spark operations:**

- ❖ Grouping, aggregating the data by key or when joining two datasets, some operations, such as groupByKey and reduceByKey, will cause shuffling.

❖ **Caching :**

- ❖ Shuffling may occur if a dataset is cached in memory and the amount of data in the cache exceeds the amount of memory on a single node.

❖ **Data locality:**

- ❖ Spark tries to minimize shuffling by putting data on the same node as the computation that will be run on it. It must be moved to the node where the computation is being done if the data is not already stored there.

Transformation Optimization (1)

- ❖ Using efficient transformations and avoiding unnecessary shuffles can improve performance.
 - ❖ Use the **sortWithinPartitions** transformation: If you need to sort the data within each partition, you can **use the sortWithinPartitions transformation instead of sort**. This will minimize shuffling by sorting the data within each partition rather than across all the partitions.
 - ❖ Use the **repartitionByRange** transformation: If you need to sort the data and you are using a range-based partitioner, such as the RangePartitioner, you can use the **repartitionByRange transformation instead of sort**. This will minimize shuffling by sorting the data within each partition and ensuring that data with the same key is placed in the same partition.

Transformation Optimization (2)

- ❖ Using efficient transformations and avoiding unnecessary shuffles can improve performance.
 - ❖ Use the **window** function: If you need to **perform windowed aggregations**, using window functions can be helpful to specify the window and the aggregation function. The window function does not trigger shuffling, as it operates on the data within each partition.
 - ❖ Use **filtering** and **aggregation** instead of `groupBy`: If you only need to filter the data or perform simple aggregations, you can use the `filter` and `agg` transformations instead of `groupBy`. These transformations do not trigger shuffling.

Caching in Spark

```
df.cache()
```

Caching in Spark

- ❖ It is important to keep in mind that caching requires careful planning, because it utilizes the memory resources of Spark's worker nodes, which perform such tasks as executing computations and storing data.
- ❖ If the data set is significantly larger than the available memory, or you're caching RDDs or DataFrames without reusing them in subsequent steps, the potential overflow and other memory management issues could introduce bottlenecks in performance.

Data Partitioning in Spark

- ❖ While Spark provides a default partitioning strategy typically based on the number of available CPU cores, it also provides options for custom partitioning. Users might instead specify a custom partitioning function, such as dividing data on a certain key.
- ❖ One of the most important factors affecting the efficiency of parallel processing is the number of partitions.
 - ❖ If there aren't enough partitions, the available memory and resources may be underutilized.
 - ❖ On the other hand, too many partitions can lead to increased performance overhead due to task scheduling and coordination.
 - ❖ The optimal number of partitions is **usually set as a factor of the total number of cores available in the cluster.**

repartition() and coalesce()

```
df = df.repartition(200)      # repartition method
```

```
df = df.coalesce(200)        # coalesce method
```

repartition() and coalesce()

- ❖ The repartition() method increases or decreases the number of partitions in an RDD or DataFrame and **performs a full shuffle of the data across the cluster**, which can be costly in terms of processing and network latency.
- ❖ The coalesce() method decreases the number of partitions in an RDD or DataFrame and, unlike repartition(), **does not perform a full shuffle**, instead combining adjacent partitions to reduce the overall number.

Handling Data Skews

- ❖ Partitioning
 - ❖ Properly partitioning data can help distribute the workload evenly across nodes.
- ❖ Sampling
 - ❖ Using sampling techniques to identify skewed keys and apply custom partitioning or filtering strategies.
- ❖ Aggregation
 - ❖ Using alternative aggregation strategies, such as pre-aggregation or partial aggregation, to reduce the impact of data skew.

Splitting

```
from pyspark.sql.functions import rand

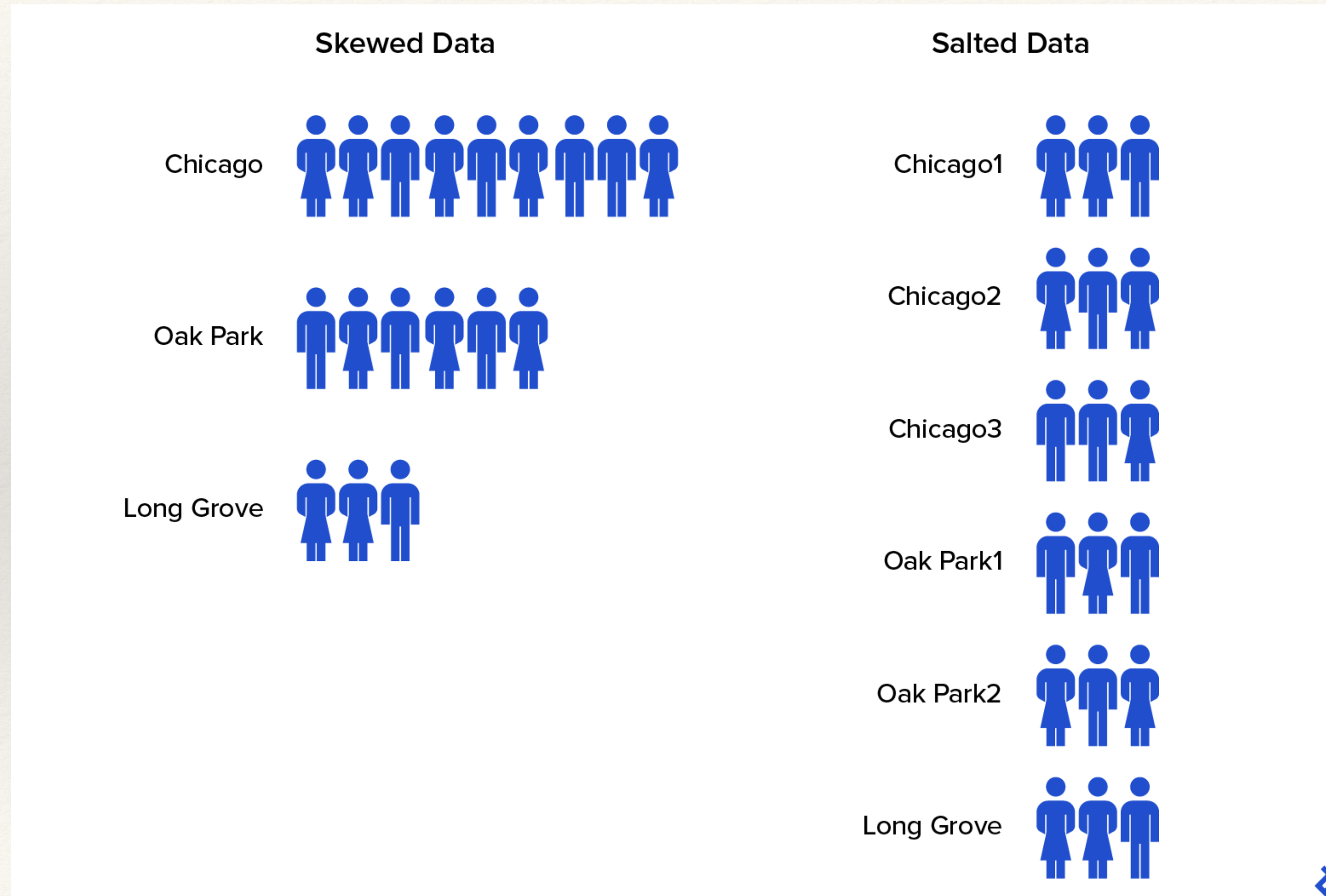
# Split the DataFrame into two DataFrames based on the skewed key.
df_skew = df.filter(df['id'] == 12345) # contains all rows where id = 12345
df_non_skew = df.filter(df['id'] != 12345) # contains all other rows

# Repartition the skewed DataFrame into more partitions.
df_skew = df_skew.repartition(10)

# Now operations can be performed on both DataFrames separately.
df_result_skew = df_skew.groupBy('id').count() # just an example operation
df_result_non_skew = df_non_skew.groupBy('id').count()

# Combine the results of the operations together using union().
df_result = df_result_skew.union(df_result_non_skew)
```


Salting



Salting

```
# In this example, the DataFrame 'df' has a skewed column 'city'.
skewed_column = 'city'

# Create a new column 'salted_city'.
# 'salted_id' consists of the original 'id' with a random integer
between 0-10 added behind it
df = df.withColumn('salted_city', (df[skewed_column].cast("string") +
(rand()*10).cast("int").cast("string")))

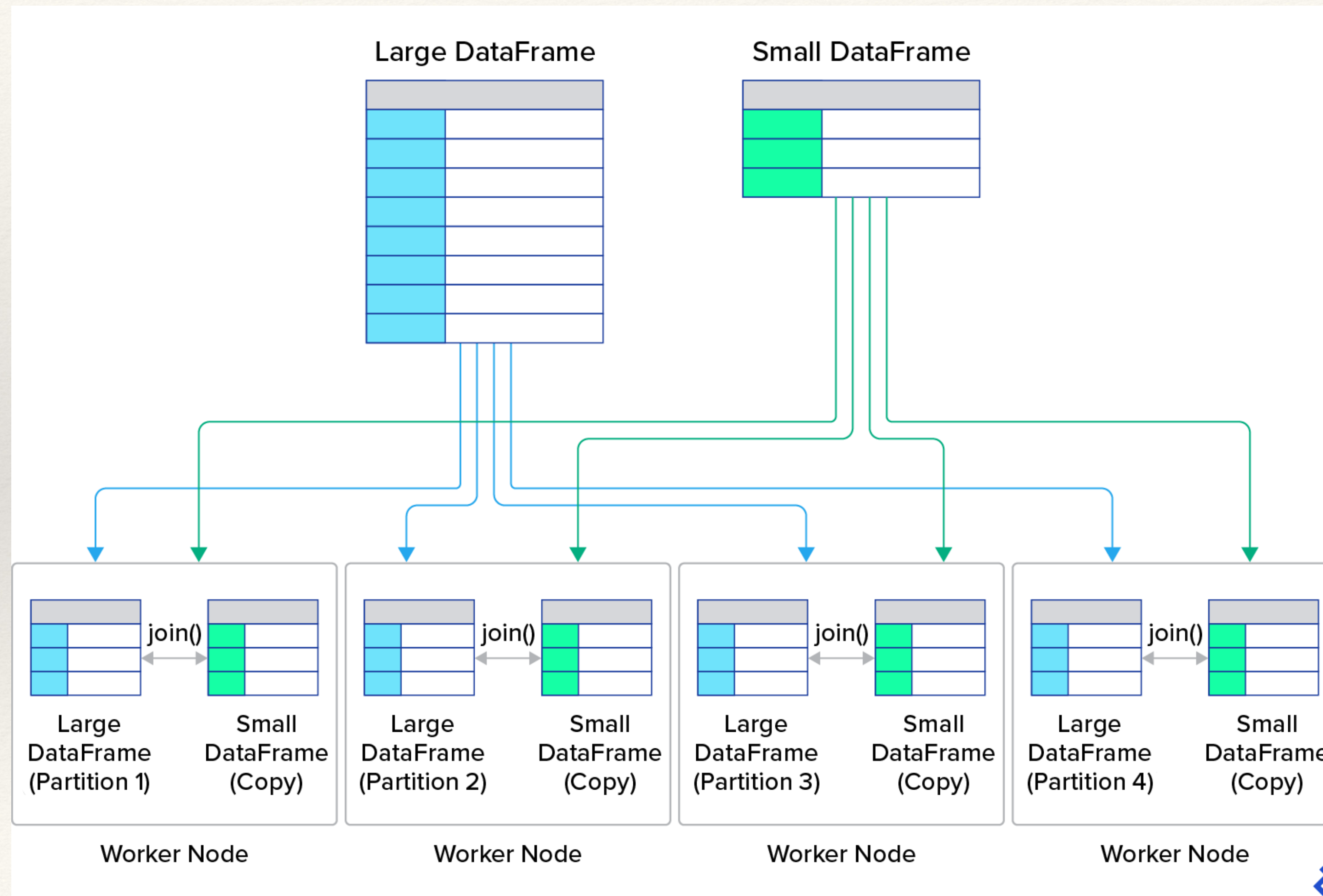
# Now operations can be performed on 'salted_city' instead of 'city'.
# Let's say we are doing a groupBy operation.
df_grouped = df.groupby('salted_city').count()

# After the transformation, the salt can be removed.
df_grouped = df_grouped.withColumn('original_city',
df_grouped['salted_city'].substr(0, len(df_grouped['salted_city'])-1))
```

Broadcasting

- ❖ A `join()` is a common operation in which two data sets are combined based on one or more common keys. Rows from two different data sets can be merged into a single data set by matching values in the specified columns. Because data shuffling across multiple nodes is required, a `join()` can be a costly operation in terms of network latency.
- ❖ In scenarios in which **a small data set is being joined with a larger data set**, Spark offers an optimization technique called broadcasting.
- ❖ **If one of the data sets is small enough to fit into the memory of each worker node, it can be sent to all nodes, reducing the need for costly shuffle operations.** The `join()` operation simply happens locally on each node.

Broadcasting



Broadcasting

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), 'id')
# must be small enough to fit into the memory of each worker node;
# a DataFrame that is too large will cause out-of-memory errors.
```

Filtering Unused Data

- ❖ When working with high-dimensional data, minimizing computational overhead is essential. Any rows or columns that are not absolutely required should be removed. Two key techniques that reduce computational complexity and memory usage are early filtering and column pruning:
 - ❖ **Early filtering:** Filtering operations should be applied as early as possible in the data processing pipeline. This **cuts down on the number of rows** that need to be processed in subsequent transformations, reducing the overall computational load and memory resources.
 - ❖ **Column pruning:** Many computations involve only a subset of columns in a data set. **Columns that are not necessary for data processing should be removed.** Column pruning can significantly decrease the amount of data that needs to be processed and stored.

Filtering

```
df = df.select('name', 'age').filter(df['age'] > 21)
```