spark. sql. autoBroadcastJoinThreshold which is by **default 10MB.**

Spark RDD is an immutable collection of objects for the following reasons: Immutable data can be shared safely across various processes and threads. It allows you to easily recreate the RDD

Spark implements fault tolerance through a combination of RDD lineage and checkpointing.

- RDD **lineage** is a mechanism that tracks the lineage of each RDD, which is a directed acyclic graph (DAG) of operations that were used to create the RDD. This allows Spark to rebuild any RDD that is lost due to a failure, by re-executing the DAG of operations that created it.
- **Checkpointing** is a mechanism that periodically saves the state of an RDD to a persistent storage system, such as HDFS or Amazon S3. This allows Spark to recover any RDD that is lost due to a failure, by restoring the state of the RDD from the checkpoint.

1. Use RDD lineage to ensure that any lost RDDs can be rebuilt.
2. Use checkpointing to periodically save the state of your RDDs to a persistent storage system.
3. Use supervisors to automatically restart executors that fail.
4. Specify retry policies to specify how many times a Spark job should be retried if it fails.
5. Use speculative execution to execute the same Spark job on multiple executors in parallel.

There are many ways to tune Spark for performance. Here are some of the most important:

- Use the **right data structures**. Spark provides a variety of data structures, each with its own strengths and weaknesses. For example, RDDs are good for iterative processing, while DataFrames and Datasets are good for batch processing. Choosing the right data structure for your application can have a significant impact on performance.
- Use the **right partitioning**. Partitioning is the process of dividing data into smaller chunks that can be processed in parallel. The right partitioning scheme can help to improve data locality and reduce the amount of shuffling that needs to occur.
- Use the **right algorithms**. Spark provides a variety of algorithms for different types of problems. Some algorithms are more efficient than others, so it is important to choose the right algorithm for your application.
- Tune the **Spark configuration**. Spark has a number of configuration options that can be tuned to improve performance. For example, you can adjust the amount of memory that is allocated to each executor, or you can configure the number of cores that are used for each task.
- Use the **right hardware**. Spark can run on a variety of hardware platforms. However, some platforms are better suited for Spark than others. For example, Spark can take advantage of GPUs to improve performance.

Spark supports two main ways to distribute data:

- **Hash partitioning** divides the data into partitions based on a hash of the data keys. This is a good choice for data that is evenly distributed and where the order of the data does not matter.

- **Range partitioning** divides the data into partitions based on a range of values. This is a good choice for data that is not evenly distributed and where the order of the data does matter.

In addition to these two main methods, Spark also supports a number of other partitioning methods, such as:

- **Custom partitioning** allows you to define your own partitioning scheme.
- **Broadcast partitioning** distributes a small amount of data to all executors.
- **Shuffle partitioning** is a special type of partitioning that is used when shuffling data between stages of a Spark job.

The best way to distribute data in Spark depends on the specific application and the characteristics of the data. However, in general, **hash partitioning** is a good choice for most applications.

Debugging Spark applications can be challenging, but there are a number of tools and techniques that can help. Here are some of the most common debugging techniques for Spark applications:

- **Use the Spark web UI:** The Spark web UI provides a graphical interface that can be used to monitor Spark jobs and troubleshoot problems. The web UI allows you to see the DAG of a Spark job, as well as the status of each task.
- **Use the Spark logs:** The Spark logs provide detailed information about the execution of Spark jobs. The logs can be used to identify errors and track down the source of problems.
- **Use breakpoints:** Breakpoints can be used to pause the execution of a Spark job at a specific point. This can be helpful for debugging problems that are difficult to reproduce.

- **Use the debugger:** The debugger can be used to step through the execution of a Spark job line by line. This can be helpful for debugging problems that are difficult to understand.
- **Use print statements:** Print statements can be used to print the value of variables at a specific point in the code. This can be helpful for debugging problems that are difficult to track down.

There are a number of ways to optimize a Spark application for performance. Here are some of the most common optimization techniques:

- **Use the right data structure:** The choice of data structure can have a significant impact on the performance of a Spark application. For example, if you are working with structured data, then you should use a DataFrame or a Dataset. If you are working with unstructured data, then you should use an RDD.
- **Use the right partitioning:** The partitioning of data can also have a significant impact on the performance of a Spark application. For example, if you are working with a large dataset, then you should use a large number of partitions. This will allow Spark to distribute the data across a larger number of worker nodes, which can improve performance.
- **Use the right Spark configuration:** The Spark configuration can also be tuned to improve performance. For example, you can increase the number of cores that are used by Spark, or you can increase the amount of memory that is allocated to Spark.
- **Use the right Spark libraries:** There are a number of Spark libraries that can be used to improve performance. For example, the MLlib library provides a number of machine learning algorithms that can be used to improve the performance of Spark applications.
- **Use the right programming techniques:** There are a number of programming techniques that can be used to improve the performance of Spark applications.

For example, you can use the `cache()` and `persist()` functions to cache data in memory, or you can use the `broadcast()` function to broadcast data across a Spark cluster.

There are a number of ways to **scale a Spark application to handle large amounts** of data. Here are some of the most common scaling techniques:

- **Increase the number of worker nodes:** The most common way to scale a Spark application is to increase the number of worker nodes in the cluster. This will allow Spark to distribute the data across a larger number of nodes, which can improve performance.
- **Increase the amount of memory:** Another way to scale a Spark application is to increase the amount of memory that is allocated to each worker node. This will allow Spark to cache more data in memory, which can improve performance.
- **Use a distributed file system:** Spark can use a distributed file system, such as Hadoop's Distributed File System (HDFS), to store data. This can improve performance by allowing Spark to access data from multiple nodes in parallel.
- (Nah) Use Spark Streaming: Spark Streaming can be used to process streaming data. This can be helpful for scaling Spark applications to handle large amounts of data that is being generated in real time.
- **Use Spark SQL:** Spark SQL can be used to work with structured data. This can be helpful for scaling Spark applications to handle large amounts of data that is stored in a relational database.

Shuffle operations are a common operation in Spark, and they can be a bottleneck for performance. Spark uses a number of techniques to optimize shuffle operations, including:

- **Map side aggregation**: Map side aggregation is a technique for performing aggregations on the map side of a shuffle operation. This can reduce the amount of data that needs to be shuffled by avoiding the need to shuffle intermediate results.
- **Shuffle file consolidation**: Shuffle file consolidation is a technique for merging shuffle files after a shuffle operation has completed. This can reduce the number of shuffle files that need to be stored on disk, which can improve the performance of subsequent shuffle operations.
- **Broadcasting**: Broadcasting is a technique for sending a copy of a large dataset to all of the worker nodes in a Spark cluster. This can be helpful for shuffle operations that involve small datasets, as it can avoid the need to copy the entire dataset across the network.
- **Coalescing**: Coalescing is a technique for reducing the number of shuffle partitions. This can be helpful for shuffle operations that involve large datasets, as it can reduce the amount of data that needs to be transferred across the network.
- **Compression**: Compression is a technique for reducing the size of data that needs to be transferred across the network. This can be helpful for shuffle operations that involve large datasets, as it can reduce the amount of time it takes to transfer the data.
- **Shuffle persist:** Shuffle persist is a technique for storing shuffle data in memory after the shuffle operation has completed. This can be helpful for shuffle operations that need to be executed multiple times, as it can avoid the need to read the data from disk each time the operation is executed.

Spark transformations are operations that are performed on RDDs (Resilient Distributed Datasets). Transformations can be either narrow or wide.

- **Narrow transformations** are operations that are performed on a single partition of an RDD. For example, the `map()` transformation is a narrow transformation because it is applied to each element in a partition independently.
- **Wide transformations** are operations that require shuffling data between partitions. For example, the `reduceByKey()` transformation is a wide transformation because it requires grouping the elements in a partition by key and then reducing the elements in each group.
- Narrow transformations:
    - `map()`
    - `filter()`
    - `flatMap()`
- Wide transformations:
    - `reduceByKey()`
    - `groupByKey()`
    - `join()`

Lineage in Spark's RDDs (Resilient Distributed Datasets) is a mechanism that tracks the lineage of each RDD. This means that Spark can track the source of each RDD, as well as the transformations that have been applied to it.

Lineage is important for fault tolerance because it allows Spark to reconstruct lost RDDs. If an RDD is lost, Spark can use the lineage information to rebuild the RDD from the original sources.

Lineage also has implications for performance and caching. Because Spark tracks the lineage of each RDD, it can avoid re-computing RDDs that have already been computed. This can improve performance by reducing the amount of computation that needs to be performed.

However, lineage also has some overhead. The lineage information needs to be stored, which can consume memory. Additionally, the lineage information needs to be processed when an RDD is rebuilt, which can slow down the reconstruction process.

Overall, lineage is a powerful tool for fault tolerance in Spark. However, it is important to weigh the benefits and drawbacks of lineage before deciding whether to use it for a particular application.

Here are some of the benefits of using lineage in Spark:

- Improved fault tolerance: Lineage allows Spark to reconstruct lost RDDs, which can improve the fault tolerance of Spark applications.
- Reduced recomputation: Lineage can avoid re-computing RDDs that have already been computed, which can improve the performance of Spark applications.
- Simplified debugging: Lineage can simplify the debugging of Spark applications by providing information about the source of each RDD.

Spark's **Catalyst optimizer** is a compiler that analyzes and optimizes Spark SQL and DataFrame queries. It uses a variety of techniques to improve the performance of queries, including:

- **Rule-based optimization:** The Catalyst optimizer uses a set of rules to rewrite queries into more efficient forms. For example, the optimizer can rewrite a join query into a more efficient form by using a hash join instead of a nested loop join.
- **Cost-based optimization:** The Catalyst optimizer also uses a cost model to estimate the cost of different query plans. This allows the optimizer to choose the most efficient query plan for a given query.
- **Iterative optimization:** The Catalyst optimizer uses an iterative optimization process to find the most efficient query plan. This process starts with a simple query plan and then iteratively refines the plan until the most efficient plan is found.

Apache Spark can run in two different deployment modes: standalone cluster mode and YARN cluster mode.

Standalone cluster mode is a simple and easy-to-use deployment mode. In standalone cluster mode, Spark creates its own cluster manager and manages all of the worker nodes in the cluster. This makes it easy to get started with Spark, but it can be less scalable and flexible than YARN cluster mode.

YARN cluster mode is a more complex deployment mode, but it is also more scalable and flexible. In YARN cluster mode, Spark uses YARN as the cluster manager. YARN is a general-purpose cluster manager that can be used to run a variety of applications, including Spark. This makes it possible to run Spark alongside other applications on the same cluster.

Here is a table that summarizes the differences between standalone cluster mode and YARN cluster mode:

Here are some of the trade-offs when choosing one deployment mode over the other for production deployments:

- Standalone cluster mode is easier to use, but it is less scalable and flexible.
- YARN cluster mode is more scalable and flexible, but it is more complex to use.
- If you are running Spark on a small cluster, then standalone cluster mode may be a good option.
- If you are running Spark on a large cluster, then YARN cluster mode may be a better option.
- If you need to run Spark alongside other applications on the same cluster, then YARN cluster mode is the only option.

How Does Salting Work in Spark?

- **Preprocess the Data**: Before performing a join or aggregation, you can preprocess the datasets by adding a random or unique prefix to each record's key. This can be done using a hash function, appending a random number, or any other suitable method.
- **Perform the Join** or Aggregation: After adding the salt, perform the join or aggregation operation on the datasets. The salting should have distributed the data more evenly across partitions, reducing the likelihood of data skew.
- **Remove the Salt**: After the join or aggregation, you can remove the salt from the keys if necessary to obtain the final results.

AQE (Adaptive Query Execution) was a feature introduced in Apache Spark 3.0. It aimed to enhance the query optimization process, making Spark SQL queries more efficient and adaptive to varying workloads. AQE improves Spark's performance by dynamically adjusting execution plans during query runtime based on data statistics and feedback from previous stages. Here are the key features and benefits of AQE:

- **Dynamic Optimization:** AQE enables Spark to adaptively optimize query plans during execution based on the actual data and statistics available at runtime. This ensures that Spark can make better decisions on how to execute a query, leading to potentially significant performance improvements.
- **Stage Reordering:** AQE introduces the ability to reorder stages of a query dynamically based on data skew, size, and computation cost. This helps in avoiding bottleneck stages and distributing the workload more evenly across the cluster.

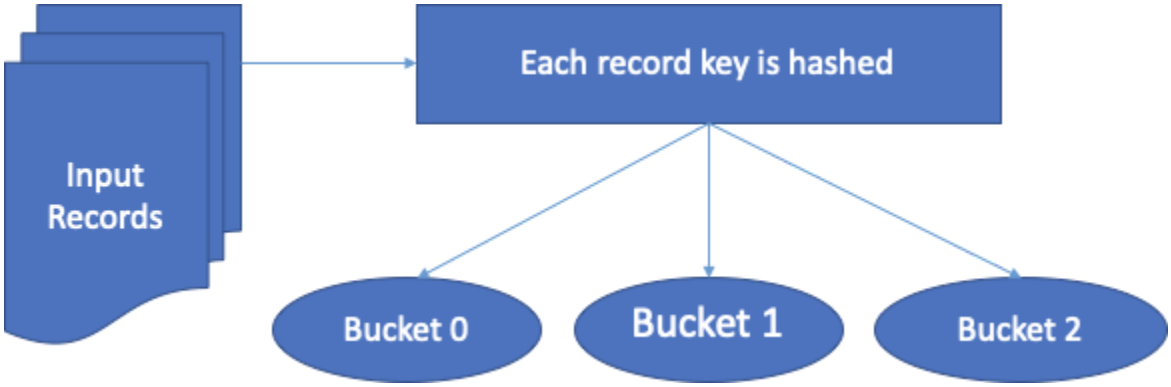
- **Runtime Coalescing:** AQE allows Spark to combine smaller tasks into larger tasks at runtime, which reduces the overhead of task scheduling and execution, resulting in better resource utilization.
- **Broadcast Hash Join Optimization:** AQE improves the performance of broadcast hash joins by making more informed decisions on whether to use the broadcast join strategy based on the size of the tables involved and available memory.
- **Cost-Based Optimization:** AQE enhances Spark's cost-based query optimization by using runtime statistics to make better decisions on join strategies, filters, and other query operations.
- **Skew Join Handling:** AQE addresses data skew issues in join operations by automatically redistributing skewed data, which helps in avoiding out-of-memory errors and improving overall query performance.
- **Adaptive Shuffle Fetch:** AQE monitors the data size being fetched during shuffle operations and dynamically adjusts the number of reducers to optimize network I/O and prevent data spills.
- **Improved Caching:** AQE optimizes caching decisions by considering the size of the data to be cached and the available memory in the cluster.

  https://spark.apache.org/docs/latest/sql-performance-tuning.html#spliting-skewed-shuffle-partitions

Overall, AQE in Apache Spark 3.0 brings adaptive query optimization capabilities that lead to more efficient query processing and improved performance, especially in scenarios with varying workloads and data distributions. As with any new feature, it's essential to keep track of the latest developments and best practices in the Spark community to leverage AQE effectively.

What is bucketing?

Bucketing is a technique in both Spark and Hive used to optimize the performance of the task. In bucketing buckets (clustering columns) determine data partitioning and prevent data shuffle. Based on the value of one or more bucketing columns, the data is allocated to a predefined number of buckets.



When we start using a bucket, we first need to specify the number of the buckets for the bucketing column (column name). At the time of loading, the data processing engine will calculate the hash value for that column, based on which it will reside in one of the buckets. Although not mandatory, using a partitioned table to do the bucketing will give the best results.

Bucketing has two key benefits:

Improved query performance: At the time of joins, we can specify the number of buckets explicitly on the same bucketed columns. Since each bucket contains an equal size of data, map-side joins perform better than a non-bucketed table on a bucketed table. In a map-side join, the left-hand side table bucket will know exactly the dataset contained by the right-hand side bucket to perform a table join in a well-structured format.

Improved sampling: The data is already split up into smaller chunks so sampling is improved.

Avoiding data spills in a Spark executor is crucial for maintaining optimal performance and avoiding out-of-memory errors. Data spills occur when the amount of data being processed or shuffled in a task exceeds the available memory in the executor, causing excess data to be written to disk. This can lead to significant performance degradation due to increased disk I/O.

Here are some strategies to avoid data spills in a Spark executor:

Increase Executor Memory: Allocate sufficient memory to each executor to handle the data processing and shuffling tasks efficiently. This can be done by adjusting the spark.executor.memory configuration property.

Tune Spark Memory Overhead: Spark uses off-heap memory for certain internal data structures, known as memory overhead. Properly configuring spark.executor.memoryOverhead is crucial to ensure that enough memory is available for Spark's internal operations.

Adjust Memory Fraction: Spark allows you to control the memory allocation for storage and execution by setting the spark.storage.memoryFraction and spark.shuffle.memoryFraction properties. Properly tuning these fractions helps prevent excessive memory allocation for caching and shuffling.

Optimize Data Serialization: Efficient data serialization, such as using Apache Parquet or Apache Arrow formats, can significantly reduce the memory footprint of data during shuffle operations.

Repartition Data: Ensuring proper data partitioning using repartition() or coalesce() can help avoid unnecessary data shuffling during transformations.

Use Memory-optimized Data Structures: Leveraging Spark's optimized data structures like DataFrames and Datasets can reduce memory overhead and improve memory management.

https://medium.com/@ghoshsiddharth25/apache-spark-join-strategies-deep-dive-26bf7e85db28

Spark join types

1. Broadcast Hash Join
2. Shuffle Hash Join
3. **Shuffle Sort Merge Join (default)**
4. Cartesian Join
5. Broadcast Nested Loop Join

Hash Join

In the case of a Hash Join, a hash table is created based on the join key of the smaller dataset and then looping over the larger dataset to match the hashed join key fields. It only supports the equivalence join condition. And this strategy is applied at the per node level(all partitions on the nodes where the dataset is available). The creation of the hash table improves the searching. Once the hash

table is created for the smaller dataset, loop over the larger dataset and based on the join key attribute, search the hash value in the smaller dataset(O(1) operation).

Broadcast Hash Join

Broadcast Hash join employs a simple strategy of broadcasting the smaller dataset to the worker nodes thus saving the Shuffle cost. This type of join is quite useful when one of the datasets is huge and the other is small usually less than 10 MB(default) but can be configurable. This join is also called Map End Join.

1. The property is configurable and has a max limit of 8GB.
2. The table is cached on the driver node as well on the executor nodes and if a large table is broadcasted then it will be a network intensive operation leading to performance degradation.

Shuffle Hash Join

Shuffle Hash Join involves a two-phase process, the shuffle and hash join phase. Datasets with the same join key are moved to the same executor node and then on the executor node, create a hash table for the smaller table and apply Hash Join.

Shuffle Sort Merge Join

Shuffle Sort Merge Join as the name suggest involves a shuffle and sort-merge phase. Datasets with the same join key are moved to the same executor node and then on the executor node, the dataset partitions on the node are sorted by the join keys and then merged based on the join keys.

pyspark.sql.DataFrame.explain

DataFrame.explain(extended=None, mode=None)

specifies the expected output format of plans.

simple: Print only a physical plan.

extended: Print both logical and physical plans.

codegen: Print a physical plan and generated codes if they are available.

cost: Print a logical plan and statistics if they are available.

formatted: Split explain output into two sections: a physical plan outline and node details.