For some workloads, it is possible to improve performance by either caching data in memory, or by turning on some experimental options.

Caching Data In Memory

Spark SQL can cache tables using an in-memory columnar format by calling `spark.catalog.cacheTable("tableName")` or `dataFrame.cache()`. Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure. You can call `spark.catalog.uncacheTable("tableName")` or `dataFrame.unpersist()` to remove the table from memory.

Configuration of in-memory caching can be done using the `setConf` method on `SparkSession` or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning	Since Version
`spark.sql.inMemoryColumnarStorage.compressed`	true	When set to true, Spark SQL will automatically select a compression codec for each column based on statistics of the data.	1.0.1
`spark.sql.inMemoryColumnarStorage.batchSize`	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.	1.1.1

Other Configuration Options

The following options can also be used to tune the performance of query execution. It is possible that these options will be deprecated in future release as more optimizations are performed automatically.

Property Name	Default	Meaning
`spark.sql.files.maxPartitionBytes`	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when files. This configuration is effective only when using file-based sour as Parquet, JSON and ORC.
`spark.sql.files.openCostInBytes`	4194304 (4 MB)	The estimated cost to open a file, measured by the number of byte could be scanned in the same time. This is used when putting mult into a partition. It is better to over-estimate, then the partitions with will be faster than partitions with bigger files (which is scheduled fir configuration is effective only when using file-based sources such Parquet, JSON and ORC.
`spark.sql.files.minPartitionNum`	Default Parallelism	The suggested (not guaranteed) minimum number of split file parti set, the default value is `spark.sql.leafNodeDefaultParallelism`. Th configuration is effective only when using file-based sources such Parquet, JSON and ORC.
`spark.sql.files.maxPartitionNum`	None	The suggested (not guaranteed) maximum number of split file part is set, Spark will rescale each partition to make the number of part close to this value if the initial number of partitions exceeds this va configuration is effective only when using file-based sources such Parquet, JSON and ORC.
`spark.sql.broadcastTimeout`	300	Timeout in seconds for the broadcast wait time in broadcast joins
`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broad worker nodes when performing a join. By setting this value to -1, broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command `ANALYZE TABLE <tableName> COMPUTE STATISTICS` been run.
`spark.sql.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for aggregations.
`spark.sql.sources.parallelPartitionDiscovery.threshold`	32	Configures the threshold to enable parallel listing for job input path number of input paths is larger than this threshold, Spark will list th using Spark distributed job. Otherwise, it will fallback to sequential This configuration is only effective when using file-based data sour as Parquet, ORC and JSON.
`spark.sql.sources.parallelPartitionDiscovery.parallelism`	10000	Configures the maximum listing parallelism for job input paths. In c number of input paths is larger than this value, it will be throttled d this value. This configuration is only effective when using file-base sources such as Parquet, ORC and JSON.

Join Strategy Hints for SQL Queries

The join strategy hints, namely `BROADCAST`, `MERGE`, `SHUFFLE_HASH` and `SHUFFLE_REPLICATE_NL`, instruct Spark to use the hinted strategy on each specified relation when joining them with another relation. For example, when the `BROADCAST` hint is used on table 't1', broadcast join (either broadcast hash join or broadcast nested loop join depending on whether there is any equi-join key) with 't1' as the build side will be prioritized by Spark even if the size of table 't1' suggested by the statistics is above the configuration `spark.sql.autoBroadcastJoinThreshold`.

When different join strategy hints are specified on both sides of a join, Spark prioritizes the `BROADCAST` hint over the `MERGE` hint over the `SHUFFLE_HASH` hint over the `SHUFFLE_REPLICATE_NL` hint. When both sides are specified with the `BROADCAST` hint or the `SHUFFLE_HASH` hint, Spark will pick the build side based on the join type and the sizes of the relations.

Note that there is no guarantee that Spark will choose the join strategy specified in the hint since a specific strategy may not support all join types.

Python **Scala** **Java** **R** **SQL**

```
spark.table("src").join(spark.table("records").hint("broadcast"), "key").show()
```

For more details please refer to the documentation of Join Hints.

Coalesce Hints for SQL Queries

Coalesce hints allow Spark SQL users to control the number of output files just like `coalesce`, `repartition` and `repartitionByRange` in the Dataset API, they can be used for performance tuning and reducing the number of output files. The “COALESCE” hint only has a partition number as a parameter. The “REPARTITION” hint has a partition number, columns, or both/neither of them as parameters. The “REPARTITION_BY_RANGE” hint must have column names and a partition number is optional. The “REBALANCE” hint has an initial partition number, columns, or both/neither of them as parameters.

```
SELECT /*+ COALESCE(3) */ * FROM t;
SELECT /*+ REPARTITION(3) */ * FROM t;
SELECT /*+ REPARTITION(c) */ * FROM t;
SELECT /*+ REPARTITION(3, c) */ * FROM t;
SELECT /*+ REPARTITION */ * FROM t;
SELECT /*+ REPARTITION_BY_RANGE(c) */ * FROM t;
SELECT /*+ REPARTITION_BY_RANGE(3, c) */ * FROM t;
SELECT /*+ REBALANCE */ * FROM t;
SELECT /*+ REBALANCE(3) */ * FROM t;
SELECT /*+ REBALANCE(c) */ * FROM t;
SELECT /*+ REBALANCE(3, c) */ * FROM t;
```

For more details please refer to the documentation of Partitioning Hints.

Adaptive Query Execution

Adaptive Query Execution (AQE) is an optimization technique in Spark SQL that makes use of the runtime statistics to choose the most efficient query execution plan, which is enabled by default since Apache Spark 3.2.0. Spark SQL can turn on and off AQE by `spark.sql.adaptive.enabled` as an umbrella configuration. As of Spark 3.0, there are three major features in AQE: including coalescing post-shuffle partitions, converting sort-merge join to broadcast join, and skew join optimization.

Coalescing Post Shuffle Partitions

This feature coalesces the post shuffle partitions based on the map output statistics when both `spark.sql.adaptive.enabled` and `spark.sql.adaptive.coalescePartitions.enabled` configurations are true. This feature simplifies the tuning of shuffle partition number when running queries. You do not need to set a proper shuffle partition number to fit your dataset. Spark can pick the proper shuffle partition number at runtime once you set a large enough initial number of shuffle partitions via `spark.sql.adaptive.coalescePartitions.initialPartitionNum` configuration.

Property Name	**Default**	**Meaning**
`spark.sql.adaptive.coalescePartitions.enabled`	true	When true and `spark.sql.adaptive.enabled` is true, Spark will coa partitions according to the target size (specified by `spark.sql.adaptive.advisoryPartitionSizeInBytes`), to avo
`spark.sql.adaptive.coalescePartitions.parallelismFirst`	true	When true, Spark ignores the target size specified by `spark.sql.adaptive.advisoryPartitionSizeInBytes` (default contiguous shuffle partitions, and only respect the minimum partition by `spark.sql.adaptive.coalescePartitions.minPartitionSize` parallelism. This is to avoid performance regression when enabling a recommended to set this config to false and respect the target size s by `spark.sql.adaptive.advisoryPartitionSizeInBytes`.
`spark.sql.adaptive.coalescePartitions.minPartitionSize`	1MB	The minimum size of shuffle partitions after coalescing. This is useful during partition coalescing, which is the default case.
`spark.sql.adaptive.coalescePartitions.initialPartitionNum`	(none)	The initial number of shuffle partitions before coalescing. If not set, it to `spark.sql.shuffle.partitions`. This configuration only has an when `spark.sql.adaptive.enabled` and `spark.sql.adaptive.co` both enabled.
`spark.sql.adaptive.advisoryPartitionSizeInBytes`	64 MB	The advisory size in bytes of the shuffle partition during adaptive opti (when `spark.sql.adaptive.enabled` is true). It takes effect when S partitions or splits skewed shuffle partition.

Spliting skewed shuffle partitions

Property Name	**Default**	**Meaning**
`spark.sql.adaptive.optimizeSkewsInRebalancePartitions.enabled`	true	When true and `spark.sql.adaptive.enabled` is true, Spark wi optimize the skewed shuffle partitions in RebalancePartitions an split them to smaller ones according to the target size (specified by `spark.sql.adaptive.advisoryPartitionSizeInBytes`), t avoid data skew.
`spark.sql.adaptive.rebalancePartitionsSmallPartitionFactor`	0.2	A partition will be merged during splitting if its size is small than this factor multiply `spark.sql.adaptive.advisoryPartitionSizeInByt`

Converting sort-merge join to broadcast join

AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side is smaller than the adaptive broadcast hash join threshold. This is not as efficient as planning a broadcast hash join in the first place, but it’s better than keep doing the sort-merge join, as we can save the sorting of both the join sides, and read shuffle files locally to save network traffic(if `spark.sql.adaptive.localShuffleReader.enabled` is true)

Property Name	Default	Meaning	Since Version
`spark.sql.adaptive.autoBroadcastJoinThreshold`	(none)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1, broadcasting can be disabled. The default value is the same as `spark.sql.autoBroadcastJoinThreshold`. Note that, this config is used only in adaptive framework.	3.2.0
`spark.sql.adaptive.localShuffleReader.enabled`	true	When true and `spark.sql.adaptive.enabled` is true, Spark tries to use local shuffle reader to read the shuffle data when the shuffle partitioning is not needed, for example, after converting sort-merge join to broadcast-hash join.	3.0.0

Converting sort-merge join to shuffled hash join

AQE converts sort-merge join to shuffled hash join when all post shuffle partitions are smaller than a threshold, the max threshold can see the config `spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`.

Property Name	Default	Meaning	
`spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`	0	Configures the maximum size in bytes per partition that can be allowed to build local hash map. If this value is not smaller than `spark.sql.adaptive.advisoryPartitionSizeInBytes` and all the partition sizes are not larger than this config, join selection prefers to use shuffled hash join instead of sort merge join regardless of the value of `spark.sql.join.preferSortMergeJoin`.	

Optimizing Skew Join

Data skew can severely downgrade the performance of join queries. This feature dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed tasks into roughly evenly sized tasks. It takes effect when both `spark.sql.adaptive.enabled` and `spark.sql.adaptive.skewJoin.enabled` configurations are enabled.

Property Name	Default	Meaning
`spark.sql.adaptive.skewJoin.enabled`	true	When true and `spark.sql.adaptive.enabled` is true, Spark dyna handles skew in sort-merge join by splitting (and replicating if need skewed partitions.
`spark.sql.adaptive.skewJoin.skewedPartitionFactor`	5.0	A partition is considered as skewed if its size is larger than this fac multiplying the median partition size and also larger than `spark.sql.adaptive.skewJoin.skewedPartitionThresho`
`spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`	256MB	A partition is considered as skewed if its size in bytes is larger thar threshold and also larger than `spark.sql.adaptive.skewJoin.skewedPartitionFactor` the median partition size. Ideally, this config should be set larger than `spark.sql.adaptive.advisoryPartitionSizeInBytes`.
`spark.sql.adaptive.forceOptimizeSkewedJoin`	false	When true, force enable OptimizeSkewedJoin, which is an adaptiv optimize skewed joins to avoid straggler tasks, even if it introduces shuffle.

Misc

Property Name	Default	Meaning	Since Version
`spark.sql.adaptive.optimizer.excludedRules`	(none)	Configures a list of rules to be disabled in the adaptive optimizer, in which the rules are specified by their rule names and separated by comma. The optimizer will log the rules that have indeed been excluded.	3.1.0
`spark.sql.adaptive.customCostEvaluatorClass`	(none)	The custom cost evaluator class to be used for adaptive execution. If not being set, Spark will use its own `SimpleCostEvaluator` by default.	3.2.0