Caching Data

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 via `spark.conf.set` 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

Tuning Partitions

Property Name	Default	Meaning	Since Version

spark.sql.files.maxPartitionBytes	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	2.0.0
spark.sql.files.openCostInBytes	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over-estimate, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	2.0.0
spark.sql.files.minPartitionNum	Default Parallelism	The suggested (not guaranteed) minimum number of split file partitions. If not set, the default value is `spark.sql.leafNodeDefaultParallelism`. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	3.1.0
spark.sql.files.maxPartitionNum	None	The suggested (not guaranteed) maximum number of split file partitions. If it is set, Spark will rescale each partition to make the number of partitions is close to this value if the initial number of partitions	3.5.0

		exceeds this value. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	
`spark.sql.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for joins or aggregations.	1.1.0
`spark.sql.sources.parallelPartitionDiscovery.threshold`	32	Configures the threshold to enable parallel listing for job input paths. If the number of input paths is larger than this threshold, Spark will list the files by using Spark distributed job. Otherwise, it will fallback to sequential listing. This configuration is only effective when using file-based data sources such as Parquet, ORC and JSON.	1.5.0
`spark.sql.sources.parallelPartitionDiscovery.parallelism`	10000	Configures the maximum listing parallelism for job input paths. In case the number of input paths is larger than this value, it will be throttled down to use this value. This configuration is only effective when using file-based data sources such as Parquet, ORC and JSON.	2.1.1

Coalesce Hints

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.

Leveraging Statistics

Apache Spark’s ability to choose the best execution plan among many possible options is determined in part by its estimates of how many rows will be output by every node in the execution plan (read, filter, join, etc.). Those estimates in turn are based on statistics that are made available to Spark in one of several ways:

- Data source: Statistics that Spark reads directly from the underlying data source, like the counts and min/max values in the metadata of Parquet files. These statistics are maintained by the underlying data source.
- Catalog: Statistics that Spark reads from the catalog, like the Hive Metastore. These statistics are collected or updated whenever you run `ANALYZE TABLE`.
- Runtime: Statistics that Spark computes itself as a query is running. This is part of the adaptive query execution framework.

Missing or inaccurate statistics will hinder Spark’s ability to select an optimal plan, and may lead to poor query performance. It’s helpful then to inspect the statistics available to Spark and the estimates it makes during query planning and execution.

- Data object statistics: You can inspect the statistics on a table or column with `DESCRIBE EXTENDED`.
- Query plan estimates: You can inspect Spark’s cost estimates in the optimized query plan via `EXPLAIN COST` or `DataFrame.explain(mode="cost")`.
- Runtime statistics: You can inspect these statistics in the SQL UI under the “Details” section as a query is running. Look for `Statistics(..., isRuntime=true)` in the plan.

Optimizing the Join Strategy

Automatically Broadcasting Joins

Property Name	Default	Meaning	Since Version
`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	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.	1.1.0
`spark.sql.broadcastTimeout`	300	Timeout in seconds for the broadcast wait time in broadcast joins	1.3.0

Join Strategy Hints

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.

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.

Property Name	Default	Meaning	Since Version
`spark.sql.adaptive.enabled`	true	When true, enable adaptive query execution, which re-optimizes the query plan in the middle of query execution, based on accurate runtime statistics.	1.6.0

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	Since

			Version
spark.sql.adaptive.coalescePartitions.enabled	true	When true and spark.sql.adaptive.enabled is true, Spark will coalesce contiguous shuffle partitions according to the target size (specified by spark.sql.adaptive.advisoryPartitionSizeInBytes), to avoid too many small tasks.	3.0.0
spark.sql.adaptive.coalescePartitions.parallelismFirst	true	When true, Spark ignores the target size specified by spark.sql.adaptive.advisoryPartitionSizeInBytes (default 64MB) when coalescing contiguous shuffle partitions, and only respect the minimum partition size specified by spark.sql.adaptive.coalescePartitions.minPartitionSize (default 1MB), to maximize the parallelism. This is to avoid performance regressions when enabling adaptive query execution. It's recommended to set this config to false on a busy cluster to make resource utilization more efficient (not many small tasks).	3.2.0
spark.sql.adaptive.coalescePartitions.minPartitionSize	1MB	The minimum size of shuffle partitions after coalescing. This is useful when the target size is ignored during partition coalescing, which is the default case.	3.2.0

spark.sql.adaptive.coalescePartitions.initialPartitionNum	(none)	The initial number of shuffle partitions before coalescing. If not set, it equals to spark.sql.shuffle.partitions. This configuration only has an effect when spark.sql.adaptive.enabled and spark.sql.adaptive.coalescePartitions.enabled are both enabled.	3.0.0
spark.sql.adaptive.advisoryPartitionSizeInBytes	64 MB	The advisory size in bytes of the shuffle partition during adaptive optimization (when spark.sql.adaptive.enabled is true). It takes effect when Spark coalesces small shuffle partitions or splits skewed shuffle partition.	3.0.0

Splitting skewed shuffle partitions

Property Name	Default	Meaning	Since Version
spark.sql.adaptive.optimizeSkewsInRebalancePartitions.enabled	true	When true and spark.sql.adaptive.enabled is true, Spark will optimize the skewed shuffle partitions in RebalancePartitions and split them to smaller ones according to the target size (specified by spark.sql.adaptive.advisoryPartitionSizeInBytes), to avoid data skew.	3.2.0

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.advisoryPartitionSizeInBytes.	3.3.0

Converting sort-merge join to broadcast join

AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side are 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 continuing the sort-merge join, as we can avoid sorting both join sides and read shuffle files locally to save network traffic (provided 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	3.0.0

		sort-merge join to broadcast-hash join.	

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 the threshold configured in `spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`.

Property Name	Default	Meaning	Since Version
`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`.	3.2.0

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	Since

			Version
spark.sql.adaptive.skewJoin.enabled	true	When true and spark.sql.adaptive.enabled is true, Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions.	3.0.0
spark.sql.adaptive.skewJoin.skewedPartitionFactor	5.0	A partition is considered as skewed if its size is larger than this factor multiplying the median partition size and also larger than spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes.	3.0.0
spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes	256MB	A partition is considered as skewed if its size in bytes is larger than this threshold and also larger than spark.sql.adaptive.skewJoin.skewedPartitionFactor multiplying the median partition size. Ideally, this config should be set larger than spark.sql.adaptive.advisoryPartitionSizeInBytes.	3.0.0
spark.sql.adaptive.forceOptimizeSkewedJoin	false	When true, force enable OptimizeSkewedJoin, which is an adaptive rule to optimize skewed joins to avoid straggler tasks, even if it introduces extra shuffle.	3.3.0

Advanced Customization

You can control the details of how AQE works by providing your own cost evaluator class or by excluding AQE optimizer rules.

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

Storage Partition Join

Storage Partition Join (SPJ) is an optimization technique in Spark SQL that makes use the existing storage layout to avoid the shuffle phase.

This is a generalization of the concept of Bucket Joins, which is only applicable for bucketed tables, to tables partitioned by functions registered in FunctionCatalog. Storage Partition Joins are currently supported for compatible V2 DataSources.

The following SQL properties enable Storage Partition Join in different join queries with various optimizations.

Property Name	Default	Meaning	Since Version

`spark.sql.sources.v2.bucketing.enabled`	true	When true, try to eliminate shuffle by using the partitioning reported by a compatible V2 data source.	3.3.0
`spark.sql.sources.v2.bucketing.pushPartValues.enabled`	true	When enabled, try to eliminate shuffle if one side of the join has missing partition values from the other side. This config requires `spark.sql.sources.v2.bucketing.enabled` to be true.	3.4.0
`spark.sql.requireAllClusterKeysForCoPartition`	true	When true, require the join or MERGE keys to be same and in the same order as the partition keys to eliminate shuffle. Hence, set to false in this situation to eliminate shuffle.	3.4.0
`spark.sql.sources.v2.bucketing.partiallyClusteredDistribution.enabled`	false	When true, and when the join is not a full outer join, enable skew optimizations to handle partitions with large amounts of data when avoiding shuffle. One side will be chosen as the big table based on table statistics, and the splits on this side will be partially-clustered. The splits of the other side will be grouped and replicated to match. This config requires both `spark.sql.sources.v2.bucketing.enabled` and `spark.sql.sources.v2.bucketing.pushPartValues.enabled` to be true.	3.4.0

`spark.sql.sources.v2.bucketing.allowJoinKeysSubsetOfPartitionKeys.enabled`	false	When enabled, try to avoid shuffle if join or MERGE condition does not include all partition columns. This config requires both `spark.sql.sources.v2.bucketing.enabled` and `spark.sql.sources.v2.bucketing.pushPartValues.enabled` to be true, and `spark.sql.requireAllClusterKeysForCoPartition` to be false.	4.0.0
`spark.sql.sources.v2.bucketing.allowCompatibleTransforms.enabled`	false	When enabled, try to avoid shuffle if partition transforms are compatible but not identical. This config requires both `spark.sql.sources.v2.bucketing.enabled` and `spark.sql.sources.v2.bucketing.pushPartValues.enabled` to be true.	4.0.0
`spark.sql.sources.v2.bucketing.shuffle.enabled`	false	When enabled, try to avoid shuffle on one side of the join, by recognizing the partitioning reported by a V2 data source on the other side.	4.0.0

If Storage Partition Join is performed, the query plan will not contain Exchange nodes prior to the join.

The following example uses Iceberg (https://iceberg.apache.org/docs/latest/spark-getting-started/), a Spark V2 DataSource that supports Storage Partition Join.

```
CREATE TABLE prod.db.target (id INT, salary INT, dep STRING)
USING iceberg
PARTITIONED BY (dep, bucket(8, id))

CREATE TABLE prod.db.source (id INT, salary INT, dep STRING)
USING iceberg
```

```
PARTITIONED BY (dep, bucket(8, id))

EXPLAIN SELECT * FROM target t INNER JOIN source s
ON t.dep = s.dep AND t.id = s.id

-- Plan without Storage Partition Join
== Physical Plan ==
* Project (12)
+- * SortMergeJoin Inner (11)
   :- * Sort (5)
   :  +- Exchange (4) // DATA SHUFFLE
   :     +- * Filter (3)
   :        +- * ColumnarToRow (2)
   :           +- BatchScan (1)
   +- * Sort (10)
      +- Exchange (9) // DATA SHUFFLE
         +- * Filter (8)
            +- * ColumnarToRow (7)
               +- BatchScan (6)


SET 'spark.sql.sources.v2.bucketing.enabled' 'true'
SET 'spark.sql.iceberg.planning.preserve-data-grouping' 'true'
SET 'spark.sql.sources.v2.bucketing.pushPartValues.enabled' 'true'
SET 'spark.sql.requireAllClusterKeysForCoPartition' 'false'
SET 'spark.sql.sources.v2.bucketing.partiallyClusteredDistribution.enabled'
'true'

-- Plan with Storage Partition Join
== Physical Plan ==
* Project (10)
+- * SortMergeJoin Inner (9)
   :- * Sort (4)
   :  +- * Filter (3)
   :     +- * ColumnarToRow (2)
   :        +- BatchScan (1)
   +- * Sort (8)
      +- * Filter (7)
         +- * ColumnarToRow (6)
            +- BatchScan (5)
```