1 Links

1.1 joining

- https://databricks.com/session/optimizing-apache-spark-sql-joins
  30 minute video, good overview
- https://sujithjay.com/spark-sql/2018/02/17/Broadcast-Hash-Joins-in-Apache-Spark/
  part one of a series

1.2 catalyst

- https://databricks.com/session/deep-dive-into-catalyst-apache-spark-2-0s-optimizer
  deep dive into catalyst
- https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html databricks blog post
- http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

2 Questions

- **Tasks/cores/executors**
  - in databricks, can we specify only 2 (for instance) executors for a node with 8 cpus?
  - if so, are these executors/jvms aware that the node has more cpus available? Are they accessible Use case - want to run some non-spark multithreaded code, e.g. distribute 1 xgboost job to each node, with each job using c cores
  - Ali said that he turned off autoscaling and had one executor per node. (8 cores per node)
- **joining** We have (say) a small list of collector ids, and want to left join with 5-6 large tables (millions of collector ids, thousands of columns) What is the best (optimal) way to do this? More shuffle partitions are better? more data partitions?
- **joining also** - Build/stream? what are these? is this for hashing?

3 Joining

3.1 Shuffle Hash join

Pretty (most?) common type of join. Data from table 1 and table 2 are partitioned by key. Matching partitions are sent to the same executor, then the output is merged. This works best when

- data is evenly distributed by key
- good number of keys (unique values - monthid would be a bad key (12 unique values))

When these conditions can't be satisfied, then broadcast hash join might be suitable NOTE spark config option,`spark.sql.join.preferSortMergeJoin`, (set true by default), essentially means that Sort Merge join is always chosen over shuffle hash join

Diagnosing problems

- Tasks that take a long time - could be uneven sharding (keys not distributed evenly).
- Speculative tasks - if something takes a long time on one exdecutor, spark might send think theres a problem there and send the task to another to see if ti works there. If one particular task spawns speculative tasks, this can also be a sign that there is a sharding problem.
- Shuffle read/write memory - take a look to see if there is one task sending/receiving a lot of data (more than average)
- ls across the output files - see if one data partition is much larger than others

3.2 broadcast hash join

Small table sent to every executor, each partition is joined with the entire table, then output is returned. Bestest performance. Spark will automatically broadcast tables that are smaller than `spark.sql.autoBroadcastJoinThreshold`, which is 10 MB by default. Giving hints can override this behaviour, but if the broadcasted table is too big you may OOM your cluster. For right outer join, Spark can only broadcast the left side. For left outer, left semi, left anti and the internal join type ExistenceJoin, Spark can only broadcast the right side. I have had cases where I want to join a small list of collectors, tableA, (one column of collector keys) to a bigger table (millions of rows, thousands of columns). Normally I would do `c = tableA.join(tableB,['COLLECTOR_KEY'],how='left')`. To use broadcasting, the join has to be righted: `c = tableB.join(broadcast(tableA),['COLLECTOR_KEY'],h`

`df.explain()` on the output will tell whether a shuffle hash or broadcast hash join is being made (spark can sometimes have a hard time figuring out when to use a broadcast join with stuff in hive. giving a hint might be handy).

3.3 Cartesian join (cross join)

- create an rdd of uid by uid pairs
- broadcast that
- call a udf that retrieves data from the big tables given the uid,uid pairs

3.4 one to many join (one sided cartesian)

- number of rows can explode not a big deal with parquet - duplicate data encodes well (for output files)

3.5 theta join

join tableA with TableB on (keyA ¡ keyB +10) - Spark will treat this as a cartesian join - Much better to bucket A and B, and create an intial paritioning based on bucket equality

4 rough

5 parallelism

Spark runs on clusters. A cluster has one driver, and a bunch of nodes. Each node has one or more executors, and each executor has one or more slots. A slot is a "virtual" cpu core, when spark is running things, it will distribute the tasks to executors, and each executor will simultaneously run one task for each slot. Can think of these as cpu cores, but the number of slots is configurable (could have 6 physical cores and specify only 4 slots) (spark calls slots "cores", but databricks training referred to them as slots to avoid ambiguity).

Each executor is a process that runs on a worker node. Each slot is a seperate JVM managed by the executor process with some amount of resources allocated.

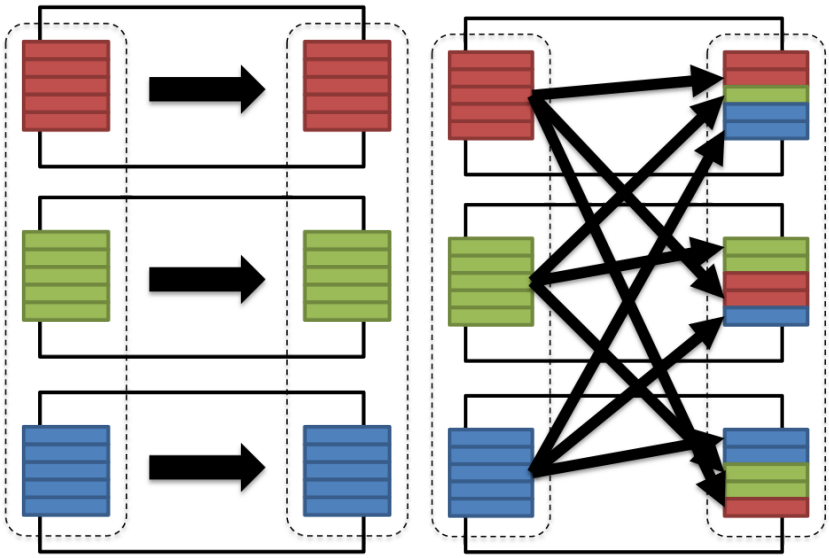
Data is parallelised also. Whenever a csv or parquet file is read in, the data is split into partitions and distributed amongst slots. If the input file was partitioned, then spark will use these partitions and distribute them amongst the slots. If the data is not partitioned (i.e. a single file, or multiple independant files), then spark will divide it into partitions and distribute them automatically (it will try to be smart about this).

6 Transformations and Actions

There are two types of operations in spark, transformations and actions. Transformations are lazy, which means when they are invoked, nothing actually happens. Spark figures out what you want to do and makes a note of it. Actions are eager, when an action is invoked, spark realizes that it needs to touch the data, so it backtracks through the lineage of the dataframe (back to the input), computes all the transformations and returns a result. Theres a bunch of optimisation (see catalyst below) steps that happen first, but these aren't executed untill you invoke the action.

6.1 Narrow and Wide transformations

Data in spark is partitioned. That is, each executor slot holds some pieces of the data. Transformations can be either narrow or wide. A narrow transformation is one in which there is a one to one mapping between input and output partitions of data. That is, the transformation can be applied to one input partition to create one partition of the output dataset.



(a) narrow transformation (b) wide transformation

Figure 1: narrow (a) and wide (b) transformations. A narrow transformation has a one to one mapping from input to output data partitions. Output partitions from a wide transformation depend on multiple input partitions (shuffle)

Narrow transformations are great. Spark (catalyst) will pipeline things, optimising sequences of narrow transformations as much as possible. Narrow transformations are extremely parallelisable, as they are performed independantly across each partition. If there is at least one data partition for each available slot, then these will scale very well.

In order to carry out a wide transformation, data needs to be shared amongst partitions. This involves a shuffle, which is an expensive operation. For example, consider

```
df.groupBy('COLLECTOR_KEY').agg(F.sum(F.col('AMRM_EARNED'))
```

You can get an idea of the execution plan by using the df.explain() function. The aggregation above requires spark to first create partial sums (by collector key) over eah partition, then distribute the data so that all of the partial sums

for each collector key end up in the same shuffle partition. Each shuffle partition then sums by collector key again to get the final totals.

Shuffling involves two exhange operations. First, data on a partition is broken up into different pieces, one for each of the shuffle partitions that will be used. Each of these pieces is written to disk. The second part involves executors (slots) pulling the data that they need from other slots. These are then merged into a single partition, and data processing continues.

When shuffling, spark needs to figure out where a given piece of data will go (partial sum by collector key in the above example). This is a mapping from a key to a shuffle partition, which is done by a hash function. Spark will use rangepartitioning (hashpartitioning) to figure out how to map data to shuffle partitions **if** the number of shuffle partitions is less than 200. In these cases hashpartitioning is optimal and works best. If there are more than 200 shuffle partitions then sortmerge partitioning is used. Ideally you want less than 200 shuffle partitions.

6.2 actions

For Dataframes, there are three types of actions

- actions to view data in the console (.show(), display in databricks)
- actions to collect results at driver (collect())
- actions to write data (write)

With dataframes, any function that acts on a dataframe and returns a dataframe is a transformation (filter, withColumn, select, etc). Any function that returns something other than a dataframe is an action (collect, write, show, count). Transformations tell spark how to change data, actions tell spark what to do with the data.

6.3 lazy Evaluation

6.4 catalyst

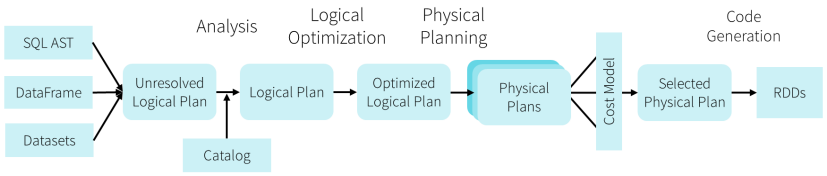


Figure 2: schematic illustrating the steps taken by catalyst in order to determine execution plan

6.5 DAG - job/stage/task

Spark jobs are initiated when actions are invoked. Each job is broken down into one or more stages, and each stage is comprised of multiple tasks. A task is a piece of work - operations that run within a single partition of data. A stage is a set of tasks that can be run without exchanging any data (i.e. narrow transformations). When data needs to be shuffled, then the stage will end in an exchange operation, where each partition divides the data it has processed into shuffle partitions and writes these to disk. The first step of the next stage involves each shuffle partition pulling the data that it needs from other slots, and merging these together.

The Directed Acyclic Graph illustrates the flow of data. Stages are seperated horizontally in the DAG, with a red box around them. Wavy arrows going from one box to another indicate shuffles.

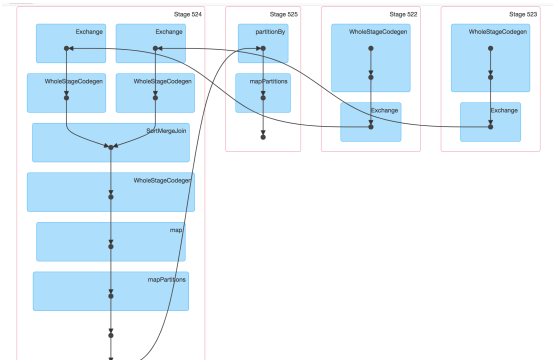


Figure 3: Directed Acyclic Graph (DAG) from a spark job. Stages are bounded by red boxes, wavy arrows between red boxes indicate shuffles (these arrows begin and end on exchange operations).