Spark Join Optimizations

Shabbir Hussain, Manthan Thakar, Tirthraj Parmar

December 5, 2017

Objective

Optimizing Spark joins on Resilient Distributed Datasets (RDDs) by employing column-pruning and / or broadcast join wherever appropriate.

Join Optimization

Join operation on RDDs can be expensive. We suspect that one of the biggest factors that affects join performance is the amount of data shuffled in the process. In order to alleviate excessive shuffling of data, we propose and implement two types of optimizations, namely, **Column Pruning** and **Broadcast Join**. In the following sections, we discuss both of these approaches and present our findings.

Optimization 1: Column Pruning

Hypothesis: Pruning unused columns of an RDD before performing a join reduces the amount of data shuffled for join and improves join performance

Although there is no concept of columns in RDDs, by columns we mean values in a PairRDD.

Approach

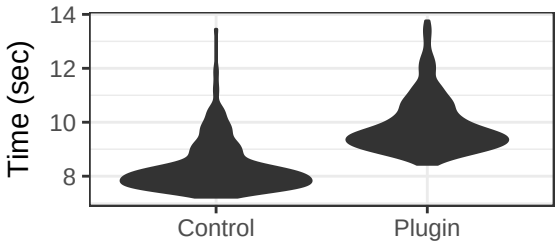
At a higher-level, we need following steps to perform column pruning on a spark join.

- **Identify target RDDs** Identify RDDs on which join is being performed.
- **Obtain column Usage** The columns from each RDD that are used after join operation
- **Transform target RDDs** Using the column usage information, transform the target RDDs before join is performed.

In order to capture information required to perform steps above, we build a compiler-plugin for scala compiler. Scala compiler plugin gives us the facility to access the abstract syntaxt tree (AST) of the source code after different phases of the compiler. This information would be either completely missing or hard to obtain inside Spark's DAG scheduler. Hence, we believe that a compiler plugin is a better choice for our purposes.

	config	value
1	Processor	m3.xlarge
2	RAM	8GB
3	vCores	4
4	Storage	256GB
5	Spark	2.2.0
6	Scala	2.11.11

Fig 2: Compilation time for program with and withou



Scala Compiler Plugin

Scala compiler can be modified by building compiler plugins. Scala compiler has 25 phases including phases liek parser, typer, erasure, etc. The compiler plugin can be placed before or after any of these phases and the modified AST obtained from the previous phase can be analyzed and rewritten inside the plugin. We use this capability and place our compiler plugin `JoinOptimizer` after scala compiler's parser phase.

When a scala program is compiled using `JoinOptimizer`, the plugin obtains the AST constructed by compiler's parser, at which point `JoinOptimizer` performs following steps:

- Identify join operations in a scala program using scala's quasiquotes
- If join operation isn't performed in the program, pass the AST to the next compiler phase without any modifications
- If a join operation is encountered, capture the `JoinContext`. A `JoinContext` stores the trees for target RDDs as well as `nextFunction` which is the function (e.g. mapValues, filter, map, etc.) that directly follows `join`
- Analyze the lambda used inside `nextFunction` and obtain column usage of target RDDs
- Using the column usage information transform the target RDDs and rewrite the tree

Fig 1 shows the source code for optimized spark join that is yielded by following steps outlined above.

Fig 1: Unoptimized code vs plugin optimized code

```
1  val rdd1: RDD[(Long, (Long, Long, Long, Long, Long))] = ... // Init RDD1
2  val rdd2: RDD[[(Long, (Long, Long, Long, Long, Long))]] = ... // Init RDD2
3
4  // Unoptimized Job
5  rdd1.join(rdd2).mapValues(x => x._1._1 + x._2._2)
6
7  // Optimized code by JoinOptimizer
8  rdd1.mapValues(_._1)
9      .join(rdd2.mapValues((null, _._2)))
10     .mapValues(x => x._1._1 + x._2._2)
```

Two PairRDDs `rdd1` and `rdd2` are defined in line 1 and 2, both with key data type of `Long` and value data type of `Tuple5[Long, ...]`. Line 5 contains code that performs `join` on `rdd1` and `rdd2` and then uses the first column from `rdd1` and second column from `rdd2` inside the lambda enclosed in `mapValues`. It is quite obvious that 4 columns in both RDDs are unused after `join`. Hence, these unused columns can be pruned.

Lines 8-10 show the code that is generated by our compiler plugin `JoinOptimizer`. It prunes the columns from both `rdd1` and `rdd2` before `join` by adding an extra `mapValues` stage. Note that, for `rdd2` it uses a null value for first column since it is not used. Ideally, we would like to only emit the second column of `rdd2` in the `mapValues` stage but the information about the size of the tuple is absent in the tree generated by `parser`.

Benchmarks

To measure the impact of `JoinOptimizer` plugin we compile spark programs with and without the plugin. The spark program used for benchmarking performs joins between two RDDs, `rdd1` and `rdd2` for different configurations of total columns and columns used after join. Note that, these configurations are only varied for `rdd1` and for simplicity, we keep `rdd2` constant for all benchmarks. As mentioned above, ideally, we'd like `JoinOptimizer` to prune unused columns.

Fig 3.1 shows execution times for both sparks programs compiled with (depicted as control) and without (depicted as plugin) `JoinOptimizer`. The blocks in heatmaps show execution times for different configurations of columns. The X-axis shows the total number of columns in `rdd1` and Y-axis shows the used number of columns right after join.

Optimization 2: Broadcast Join

Hypothesis: Out of two RDDs on which join is to be performed, if one RDD is much smaller than the other and it can fit in memory, then broadcasting the small RDD to all the nodes and then performing join reduces the amount of data being shuffled for join and improves join performance

Approach

Spark's join function by default performs a reduce side join. Meaning, it shuffles records from both target RDDs having same keys to one node and then combines their data. But in situations where one of the RDDs is much smaller, this smaller RDD can be broadcast to all the nodes. Once the smaller RDD is transferred to each node, both RDDs can be joined by adding map stages. This eliminates an extra reduce step that is required in reduce-side joins. Although, broadcast join includes sending data to all the nodes upfront, it saves the bigger RDD from being shuffled. This is equivalent to performing a map-side join instead of spark's reduce-side join.

The key components in this proposed approach are:

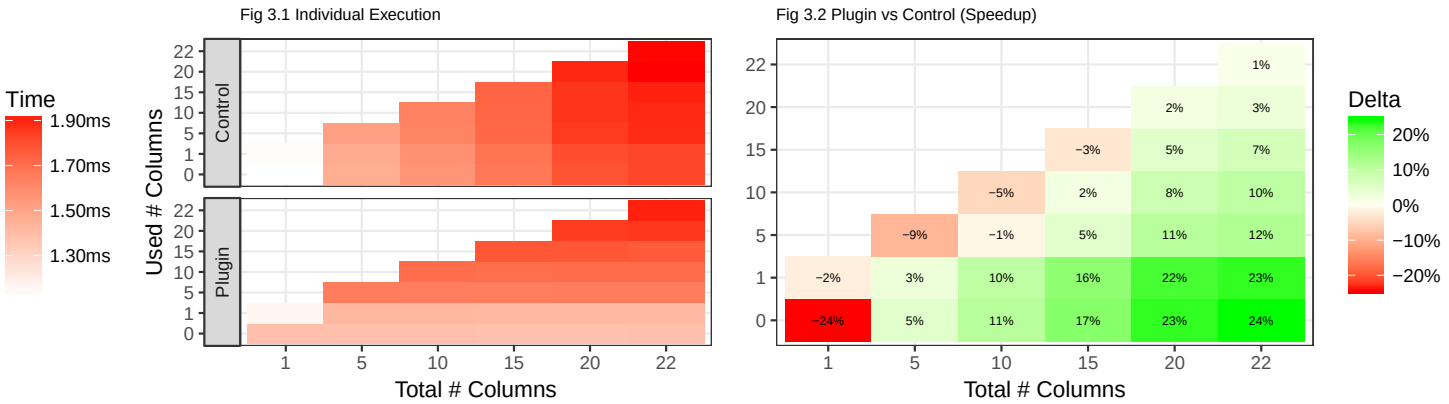


Figure 3: Execution Time

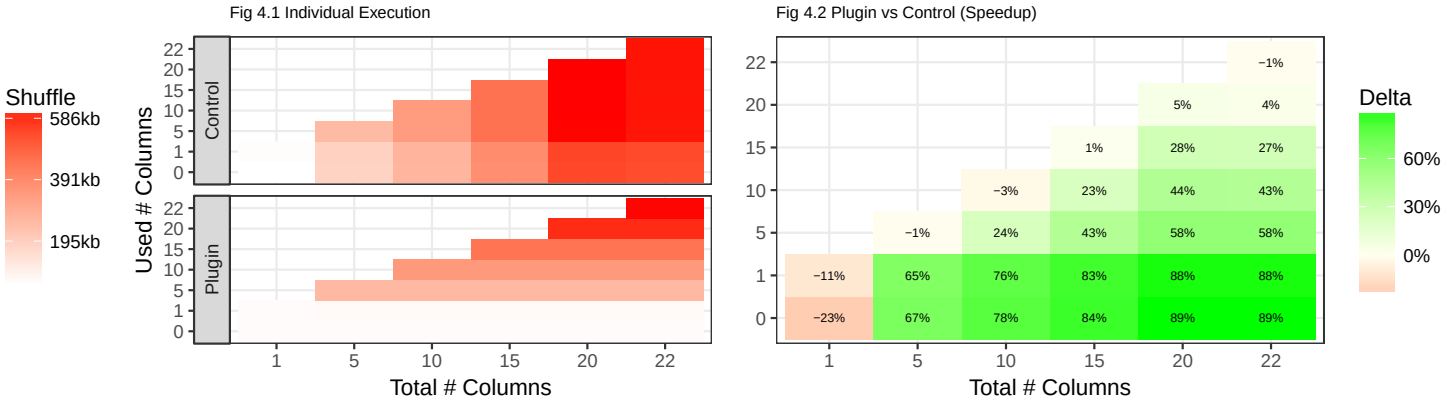


Figure 4: Shuffled Data

RDD Size Estimation

Estimating the size of target RDDs is instrumental in performing broadcast join. We have implemented a size estimator for RDDs that uses the following formula:

```
estimated size = size(N) * R, where R is total number of rows in the RDD,
                              and size(N) gives size of N sample rows fetched from each partition
```

Since this approach can be inaccurate when rows contain variable number of columns, broadcasting an RDD might result in ungraceful job failure. Therefore, we keep the estimator configurable for the user, so that user can provide a custom size estimator according to data.

Moreover, broadcast join requires that the small RDD can fit in memory on all the nodes. It can prove cumbersome to estimate memory on all the nodes in the cluster. Therefore, we keep memory threshold configurable as well. If size of an RDD obtained using the given estimator, is smaller than the memory threshold, we perform broadcast join, otherwise spark's default shuffle-join is performed.

Performing map-side join

As previously mentioned, if one of the target RDDs are smaller than memory threshold, that RDD is broadcast to all nodes. Since this smaller RDD is now cached on all nodes, we only need to map through the bigger RDD and lookup keys from the

bigger RDD in the cached smaller RDD to join records. This in turn eliminates shuffling of bigger RDD and a reduce stage.

Benchmarks

Limitations

- Broadcast join is only useful when one of the RDDs can fit in memory
- If both the RDDs are bigger than memory threshold, broadcast join incurs extra cost of performing size estimation on both the RDDs.