Techniques for efficient ETL Jobs using Apache Spark

Mr. M.G. Thiruvalluvan
VP of Engineering, Aqfer, Inc.
thiru_mg@yahoo.com

Introduction

Apache Spark offers a rich programming platform for big data processing. The platform is highly scalable and resilient against failures. The programming interface is available as an extension of high level languages – Scala, Python and Java. The programming is modeled after traditional idioms of those languages plus on Structured Query Language (SQL).

One of the important class of applications for this platform is what is known in Data warehousing Industry as Extract, Transform and Load or ETL. In this article we present three unconventional techniques that enhance the traditional methods of ETL, under certain circumstances. These techniques individually improve the efficiency of ETL, but since they are complementary they offer even more benefits when applied together.

Traditional techniques and their limitations

Traditional ETL methods require the programmer to specify as a code snippet what should happen on each data row. Occasionally one would want to drop the row because it does not meet certain criteria. Otherwise it is a one-to-one mapping of input row to an output row. This works fairly well in most situations. But problems arise when one or more of the following situations are encountered:

- The input files are not partitioned in the right size or partitioned in a non-uniform fashion. When this happens the output is also organized in the same pattern as input leaving the output also in a suboptimal way. Big data systems work well when the files are organized as a small to medium number of fairly large files (256 MB is typical). A “large number of small files” kills the system performance.
- It is true that Spark allows to coalesce any data into desired number of partitions. This operation is relatively cheap, but it does not allow the programmer to choose the size of the partition but rather the number of them. Unless the input size is fairly predictable, it is difficult to achieve correct partition size across job runs. Typical workloads generate data that varies with the time of the day and/or day of the week and, in general, not predictable.
- The other method to “repartition” using “partitionByKey” is even less performant and less predictable in terms of resource needs.
- Sometimes, one needs to produce ETL outputs in more than one format. For example, in a data lake one would like to keep data sets in both Apache Avro and Apache Parquet format. Apache Avro is a row-oriented storage and optimized for bulk processing of data in the data lake. On the other hand Apache Parquet is a columnar format optimized for ad-hoc querying using SQL using products like Impala or Presto. In traditional ETL methods, if more than on format needs to be produced, then one has to scan the entire data sets as many times as the number of output formats. If there is not enough memory to hold the entire data set, they the data is serialized into temporary persistent store and deserialized when needed.
- Spark has excellent support for writing different file formats such as Apache Avro and Apache Parquet. But the code is optimized for generality rather than performance. So there are several layers of data transformation between the in-memory representations and on-disk representation. This in turn consumes more memory and CPU.
- Spark, like many other big data platforms, requires the user to specify ahead of time the number and size of executors. Getting these two numbers is not straightforward. One has to essentially guess something that would work. If the guess is too small, the job may fail to execute stating “OutOfMemoryException”. If the guess is too large, the resources will go unused. In today’s cloud computing environments, unused compute resources directly translate to money left on the table.
- The above problem is even more challenging if the size of data changes from one job execution to another.

Our techniques

As mentioned earlier, our techniques consist of three ideas. They are:

- Use custom low-level serialization routines
- Produce output as side effect rather than as a global operation
- Use block-level file merge for file size optimization

We describe these ideas in detail in the following sections. These ideas are fairly independent of each other and hence can be employed either individually or collectively depending on your taste, your team’s expertise and nature of your workload.

Low-level serialization

Spark offers excellent support for serializing data into several popular formats. However, it involves some form of "generalization" before serialization. By generalization, we mean conversion of data objects into a single generic form so that the serialization libraries can handle them. (An alternative to this approach is to use reflection which is worse in terms of performance). Since the primary language of Spark is Scala and the underlying runtime environment is JVM, this generalization step typically mean converting into some combination of maps and lists and conversion of primitive types into their corresponding wrapper classes. Both these operations increase memory footprint and processing time. It also puts pressure on JVM's garbage collector.

All these problems can be solved by writing one's own custom serialization code which converts Scala objects directly into serialized byte streams. (All modern file formats offer pretty good support for low-level operations in their libraries). Such code is extremely efficient. But here we are trading programmer time execution time, which is a controversial decision. Often, critics point out that the cost of maintenance of this custom code is prohibitive. In our case, it did not prove to be so. The output schema for the ETL jobs has been robustly stable and we didn't need to spend a lot of effort in maintaining this code. But that is a valid concern for many organization.

One option to overcome this criticism is to use code generators which generate the custom code based on the schema. But then, the skills required to write code generators is not commonplace.

Output as side-effect

This is perhaps the most important idea of the three. Here, first, instead of handling each input row at a time, we look at one partition at a time. Typically a single input partition corresponds to either a single input file or (if the file is block structured) a block. We then use the low-level IO routines mentioned above (or generic routines provided by serialization libraries) to save the records in the partitions into a output file. If multiple output formats are required, all those formats can be written at once. By this we ensure that the same record need not ever be seen more than once.

So when a single input partition has been processed, we end up writing one file per output format and nothing is left in memory. That is why we call this technique "producing output as side effect". Thus we can process a very large amount of data without using a large amount of memory – at most one partition is kept in memory at a time. A larger amount of input data means more time to process it – there is no nasty OutOfMemoryException.
There could be two objections to this method:

- We end up having one file per input partition per output format. There are a lot of them and Big Data hates a "large number of small files". So this is an anti-pattern. The answer to this objection is the next idea described below.
- Side effects are harmful in functional and/or distributed programming because, when a portion of work needs to be rerun because of runtime errors, side effects cease to be atomic. So if a portion of task is rerun, then you end up having more than one output file per input partition per output format. The output thus may not even be correct. A simple way to overcome this problem is to ensure that the file produced for each input partition and output format is deterministically and uniquely chosen so that if work is repeated for the same input partition, the output files are also the same. When writing such output files, we must ensure that we overwrite any existing file with the same name. This ensures that exactly one file is produced per input partition per output format.

File merge

Spark's default way to combine large number of small files into small number of large files is to load the data into a data frame or data set and repartition (or coalesce) into the desired number of partitions and save it back. Obviously this is resource intensive as each row is deserialized to its fullest detail and reserialized. As mentioned earlier, there is no simple way to decide the number of partitions even though we know approximate size of output partition.

Fortunately for us all popular file formats – Apache Avro, Apache Parquet and Apache ORC are organized as blocks internally. CSV can be split into block cheaply. Compressed CSV could be troublesome. Fortunately for us ETLs do not produce CSVs as often as other file formats. Even otherwise, CSV (compressed or otherwise) entire files can be treated as blocks, provided there are no header rows. Compressed CSV files with header rows is hard-luck.
Typical block sizes are in the order of a few megabytes to tens of megabytes. For example, for Parquet typical block size is 2 to 3 MB and for Avro it is 64 MB. Also in all these file formats, one can deconstruct files into blocks and reconstruct files again from the blocks without ever looking into the blocks themselves. The algorithm to pack such files is the standard distributed bin-packing algorithms which are well-known. This method is three to four orders of magnitude faster than naïve Spark technique. Since the bin-packing algorithm can take the size of the "bin" instead of the number of bins, we end up having relatively uniform sized output files. All these file formats compress the contents inside the blocks and never

across the blocks. So the files size extremely predictable given the sizes of blocks inside them. As a bonus we don't have to un-compress or recompress the blocks.

All these files are generated in a temporary location instead of their final location in order to prevent potential consumers of these "work in progress" files. When the final version is ready they are moved into the final output location.

Performance improvements

With successive implementation of the above techniques, we could achieve up to six-fold improvement in performance when saving data into two format. Of the three techniques, low-level serialization provided up to 2x performance improvement and saving output as side effect improved performance by about 3x and hence 6x improvement overall. We have been using these methods over 3 years in production with no significant problems.

Pitfalls

No set of techniques will work in all situations. Here are some potential problems of these techniques and possible ways to work around them:

- Fragmentation. If the input partitions are too small or if the output is partitioned on keys whose cardinality is big, we end up having output files each with a single small block. Because we do not merge more than one block into a larger block, we may end up with output files with a large number of small blocks. Even though the situation is not as bad as large "number of small files", the effect is not zero. The of fragmentation is varied across file formats:
    - Almost all file formats do poor compression when block size is small and hence will need more storage and more processing
    - Performance improvement seen when using columnar storage is diminished when the blocks are too small. Pruning is a technique used by query systems to skip blocks altogether when it can be proven that the blocks are not going to contribute to the query. Pruning is less effective during query execution if the blocks are small or numerous.
- One way to avoid fragmentation is to detect when fragmentation happens and then use the standard spark load-and-repartition method to remove it. If we do that, we may discover that certain jobs are not suitable for the improvements described in this article and simply abandon them.
- Not easy to implement. These ideas are neat and produce excellent results, but most Spark teams may not have the required sill set to write low-level serialization code or block level file merge operations. Even if the teams produce such code, maintaining them may be expensive. Of course this drawback can be alleviated by publishing open source code for file merge and code generators for custom serialization. If we do that, the users of these need not spend a lot of time developing or maintaining it. We never found time to open source our work.

Note: We presented the technique in Strata Data Conference in New York in Sep 2017:
https://conferences.oreilly.com/strata/strata-ny-2017/public/schedule/detail/60729

About the author

Thiru has been building software systems for three decades. Apart from his day job, he is presently the PMC Chair of Apache Avro Project, which he helped found in 2009. His current interests include large scale distributed systems for batch and low-latency web services. He has built some innovative performant and resilient products on top of the current and past crop of Big Data platforms. In the past he worked on desktop search, web search and advertising platforms, cloud platforms and deployment automation in Yahoo! He has been developing and using cloud product for the past 10 years and has been in the forefront of server-less computing. In his current job, among other things, he has put Content Delivery Network into innovative use for beaconing solutions.

US FDA has helped develop a free-to-play horror video game in a bid to curb smoking among youth aged 12 to 17. Part of FDA's "The Real Cost" youth tobacco prevention campaign, only one out of four players can escape the game. 'One Leaves' is inspired by statistics depicting three out of four high school smokers continue on to adulthood.

Swedish luxury automaker Volvo announced vehicles starting 2021 will come with a "Care Key" that lets users set speed limits on vehicles when powered on with it. This follows the automaker's recent announcement that it will limit top-speed of all vehicles starting 2020 to 180 kmph. Volvo had also announced a camera monitoring system to detect drunk or distracted driving.