On Spark, Hive, and Small Files: An In-Depth Look at Spark Partitioning Strategies

One of the most common ways to store results from a Spark job is by writing the results to a Hive table stored on HDFS. While in theory, managing the output file count from your jobs should be simple, in reality, it can be one of the more complex parts of your pipeline.

Z Zachary Ennenga Follow
Mar 3, 2020 · 17 min read

Author: Zachary Ennenga

Airbnb's new office building, 650 Townsend

Background

At Airbnb, our offline data processing ecosystem contains many mission-critical, time-sensitive jobs — it is essential for us to maximize the stability and efficiency of our data pipeline infrastructure.

So, when a few months back, we encountered a recurring issue that caused significant outages of our data warehouse, it quickly became imperative that we understand and solve the root cause. We traced the outage back to a single job, and how it, unintentionally and unexpectedly, wrote millions of files to HDFS.

Thus, we began to investigate the various strategies that can be used to manage our Spark file count in order to maximize the stability and efficiency of our Data Engineering ecosystem.

A quick note on vocabulary

Throughout this post I will need to use the term "partitions" quite a lot, and both Hive and Spark use this to mean different things. For this reason, I will use the term **sPartition** to refer to a Spark Partition, and **hPartition** to refer to a Hive partition.

I will use the term **partition key** to refer to the set of values that make up the partition identifier of any given hPartition.

How does this even happen?

As the term "ETL" implies, most Spark jobs can be described by 3 operations: Read input data, process with Spark, save output data. This means that while your actual data transformations are occurring largely in-memory, your jobs generally begin and end with a large amount of IO.

A common stack for Spark, one we use at Airbnb, is to use Hive tables stored on HDFS as your input and output datastore. Hive partitions are represented, effectively, as directories of files on a distributed file system. In theory, it might make sense to try to write as many files as possible. However, there is a *cost*.

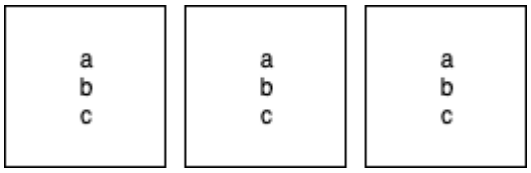
HDFS does not support large amounts of small files well. Each file has a 150 byte cost in NameNode memory, and HDFS has a limited number of overall IOPS. Spikes in file writes can absolutely take down, or otherwise render unusably slow, pieces of your HDFS infrastructure.

It may seem as though it would be hard to accidentally write a huge amount of files, but it really isn't. If your use cases only involve writing a single hPartition at a time, there are a number of solutions to this issue. But in a large data engineering organization, these cases are not the only ones you'll encounter.

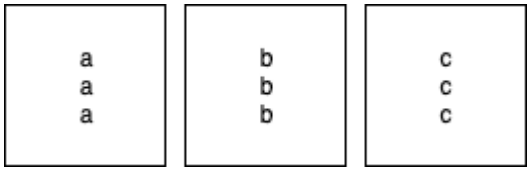
At Airbnb, we have a number of cases where we write to multiple hPartitions, most commonly, backfills. A backfill is a recomputation of a table from some historical date to the current date, often to fix a bug or data quality issue.

When handling a large dataset, say, 500GB-1TB, that contains 365 days' worth of data, you may break your data into a few thousand sPartitions for processing, perhaps, 2000–3000. While on the surface, this naive approach may seem reasonable, using dynamic partitioning, and writing your results to a Hive table partitioned by date will result in **up to 1.1M files**.

Why does this happen?



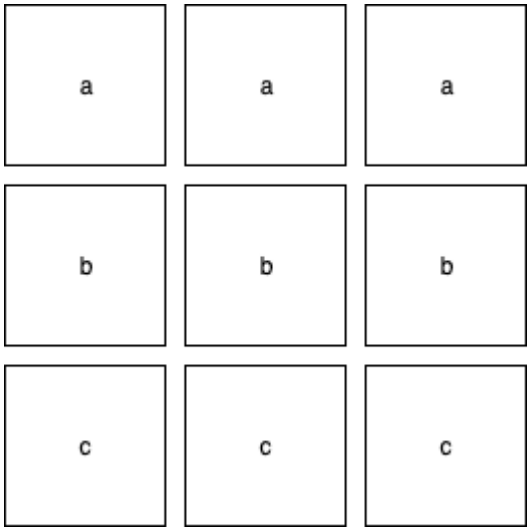
Data distributed randomly into 3 sPartitions



Our goal — Data written neatly into 3 files

Let's assume you have a job with 3 sPartitions, and you want to write to 3 hPartitions.

What you want to have happen in this situation is 3 files written to HDFS, with all records present in a single file per partition key.



Reality — Data written not-so-neatly to HDFS

What will actually happen is **you will generate 9 files, each with 1 record**.

When writing to a Hive table with dynamic partitioning, each sPartition is processed in parallel by your executors. When that sPartition is processed, each time an executor encounters a new partition key in a given sPartition, it opens a new file.

By default, Spark uses either a Hash or Round Robin partitioner on your data. Both of these, when applied to an arbitrary dataframe, can be assumed to distribute your rows relatively evenly, but randomly, throughout your sPartitions.

This means without taking any specific action, you can generally expect to write approximately 1 file per sPartition, per unique partition key, hence our 1.1M result above.

How do you decide on your target file count?

Before we dig into the various ways to convince Spark to distribute our data in a way that's amenable to efficient IO, we have to discuss what we're even aiming for.

Ideally, your target file size should be approximately a multiple of your HDFS block size, 128MB by default.

In pure Hive pipelines, there are configurations provided to automatically collect results into reasonably sized files, nearly transparently from the perspective of the developer, such as *hive.merge.smallfiles.avgsize*, or *hive.merge.size.per.task*.

However, no such functionality exists in Spark. Instead, we must use our own heuristics to try to determine, given a dataset, how many files should be written.

Size-based calculations

In theory, this is the most straightforward approach — set a target size, estimate the size of your dataframe, and then divide.

However, files are written to disk, in many cases, with compression, and in a format that is significantly different than the format of your records stored in the Java heap. This means it's far from trivial to estimate how large your records in memory will be when written to disk.

While you may be able to estimate via the size of your data in memory using the SizeEstimator utility, then apply some sort of estimated compression/file format factor, the SizeEstimator considers internal overhead of dataframes/datasets, in addition to the size of your data. Overall, this heuristic is *unlikely to be accurate for this purpose.*

Row count-based calculations

A second method is to set a target row count, count the size of your dataset, and then perform division to estimate your target.

Your target row count can be determined in a number of ways, either by picking a static number for all datasets, or by determining the size of a single record on disk, and performing the necessary calculations. Which way is best will depend on your number of datasets, and their complexity.

Counting is fairly cheap, but requires a cache before the count to avoid recomputing your dataset. We'll discuss the cost of caching later, so *while this is viable, it is not necessarily free*.

Static file counts

The simplest solution is to just require engineers to, on a per-insert basis, tell Spark how many files, in total, it should be writing. This "heuristic" will not work on its own, as we

need to give developers some other heuristic to get this number in the first place, but could be an optimization we can apply to skip an expensive calculation.

Evaluation

A hybrid is your best option here. Unknown datasets should be with a count-based heuristic to determine file count, but enable developers to take the result determined by the count heuristic, and encode it statically.

How do we get Spark to distribute our data in a reasonable way?

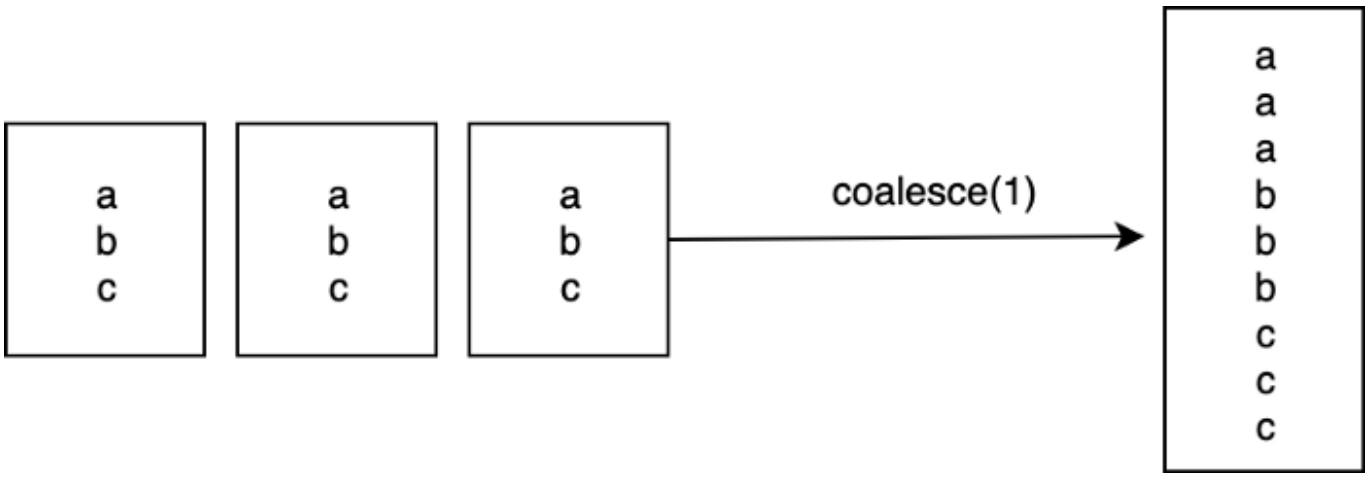
Even if we know how we want our files written to disk, we still have to get Spark to get our sPartitions structured in a way that is amenable to actually generating those files.

Spark provides you a number of tools to determine how data is distributed throughout your sPartitions. However, there is a lot of hidden complexity in the various functions, and in some cases, they have implications that are not immediately obvious.

We will go through a number of these options that Spark provides, and various other techniques that we have leveraged at Airbnb to control Spark output file count.

Coalesce

Coalesce is a special version of repartition that only allows you to decrease the total sPartitions, but does not require a full shuffle, and is thus significantly *faster* than a repartition. It does this by, effectively, merging sPartitions.



Coalesce sounds useful in some cases, but has some problems.

First, coalesce has a behavior that makes it difficult for us to use. Take a pretty basic Spark application:

```
load().map(…).filter(…).save()
```

Let's say you had a parallelism of 1000, but you only wanted to write 10 files at the end. You might think you could do:

```
load().map(…).filter(…).coalesce(10).save()
```

However, Spark's will effectively push down the coalesce operation to as early a point as possible, so this will execute as:

```
load().coalesce(10).map(…).filter(…).save()
```

The only workaround is to force an action between your transformations and your coalesce, like:

```
val df = load().map(…).filter(…).cache()
df.count()
df.coalesce(10)
```

The cache is required because otherwise, you'll have to recompute your data, which can be very costly. However, *caching is not free*; if your dataset cannot fit into memory, or if you cannot spare the memory to store your data in memory effectively twice, then you must use disk caching, which has its own limitations and a significant performance penalty.

In addition, as you will see later on, performing a shuffle is often necessary to achieve the results we want for more complicated datasets.

Evaluation

Coalesce only works for a specific subset of cases:

1. You can guarantee you are only writing to 1 hPartition
2. The target number of files is less than the number of sPartitions you're using to process your data
3. You can afford to cache or recompute your data

Simple Repartition

A simple repartition is a repartition who's only parameter is target sPartition count — IE: df.repartition(100). In this case, a round-robin partitioner is used, meaning the only guarantee is that the output data has roughly equally sized sPartitions.

Evaluation

A simple repartition can fix skewed data, where the sPartitions are wildly different sizes.

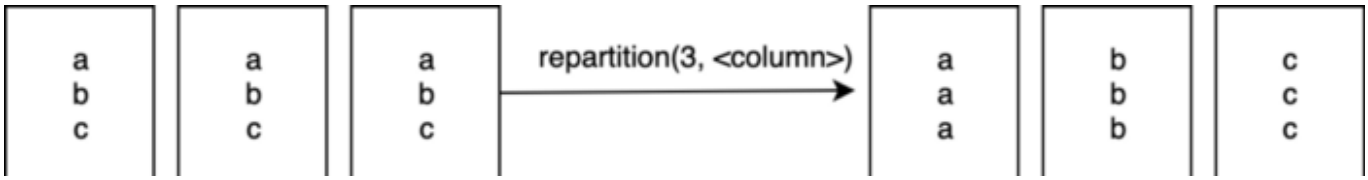
It is only useful for file count problems where:

1. You can guarantee you are only writing to 1 hPartition
2. The number of files you are writing is greater than your number of sPartitions and/or you cannot use coalesce for some other reason

Repartition by Columns

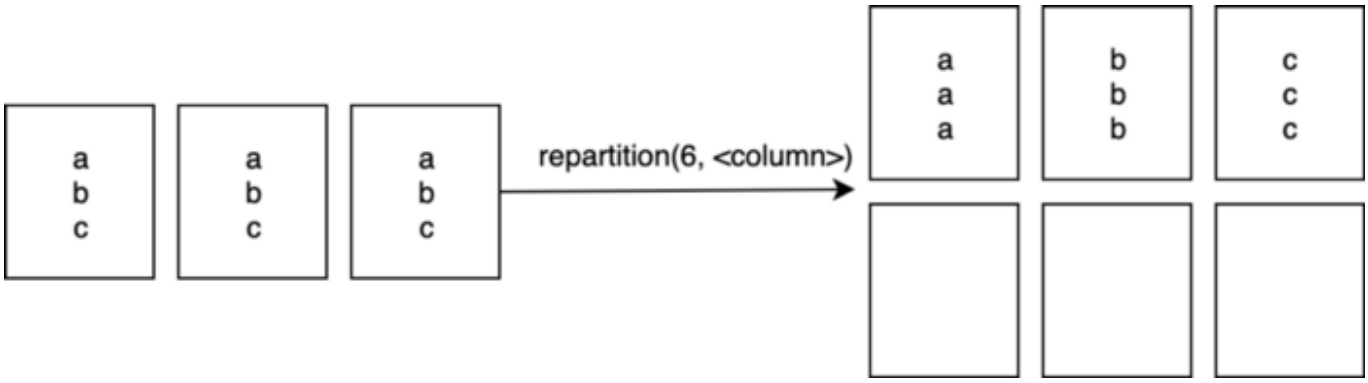
Repartition by columns takes in a target sPartition count, as well as a sequence of columns to repartition on — e.g., df.repartition(100, $"date"). This is useful for forcing Spark to distribute records with the same key to the same partition. In general, this is useful for a number of Spark operations, such as joins, but in theory, it could allow us to solve our problem as well.

Repartitioning by columns uses a HashPartitioner, which will assign records with the same value for the hash of their key to the same partition. In effect, it will do:



Beautiful!

Which, in theory, is exactly what we want!



Saying “in theory” is always inviting disaster.

However, this approach only works *if each partition key can safely be written to one file*. This is because no matter how many values have a certain hash value, they’ll end up in the same partition.

Evaluation

Repartitioning by columns only works when you are writing to one or more small hPartitions. In any other case it is not useful, because you will always end up with 1 file per hPartition, which only works for the smallest of datasets.

Repartition by Columns with a Random Factor

We can modify repartition by columns by adding a constrained random factor:

```
df
.withColumn("rand", rand() % filesPerPartitionKey)
.repartition(100, $”key”, $"rand")
```

In theory, this approach should lead to well sorted records, and files of fairly even size, as long as you meet the following conditions:

1. hPartitions are all roughly the same size
2. You know the target number of files per hPartition and can encode it at runtime

As we discussed earlier, determining the correct files-per-partition value is far from easy. However, the first condition is also far from trivial to meet:

In a backfill context, say, computing a year's worth of data, day-to-day data volume changes are low, whereas month-to-month and year-to-year changes are high. Assuming a 5% month-over-month growth rate of a data source, we expect the data volume to increase 80% over the course of the year. With a 10% month-over-month growth rate, 313%.

Given these factors, it seems we will suffer performance problems and skew over the course of any period larger than a month or so, and cannot meaningfully claim that all hPartitions will require roughly the same file count.

That said, even if we can guarantee all those conditions are met, there is one other problem: Hashing Collisions.

Collisions

Let's say you are processing 1 year's worth of data (365 unique dates), with date as your only partition key.

If you need 5 files per partition, you might do something like:

```
df.withColumn("rand", rand() % 5).repartition(5*365, $"date",
$"rand")
```

Under the hood, Scala will construct a key that contains both your date, and your random factor, something like (<date>, <0–4>). Then, if we look at the HashPartitioner code, it will do:

(See org.apache.spark.Partitioner.HashPartitioner for reference)

```
class HashPartitioner(partitions: Int) extends Partitioner {
 def getPartition(key: Any): Int = key match {
   case null => 0
   case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
```

```
    }
  }
```

Effectively, all that's being done is taking the hash of your key tuple, and then taking the (nonNegative) mod of it using the target number of sPartitions.

Let's analyze how our records will actually be distributed in this case. I have written some code to perform the analysis over here, also available as a gist here.

The above script calculates 3 quantities:

- **Efficiency**: The ratio of non-empty sPartitions (and thus, executors in use) to number of output files
- **Collision Rate**: The percentage of sPartitions where the hash of (date, rand) collided
- **Severe Collision Rate**: As above, but where the number of collisions on this key are 3 or greater

Collisions are significant because they mean our sPartitions contain multiple unique partition keys, whereas we only expected 1 per sPartition.

The results are pretty bad: We are using 63% of the executors we could be, and there is likely to be severe skew; close to half of our executors are processing 2, 3, or in some cases up to 8 times more data than we expect.

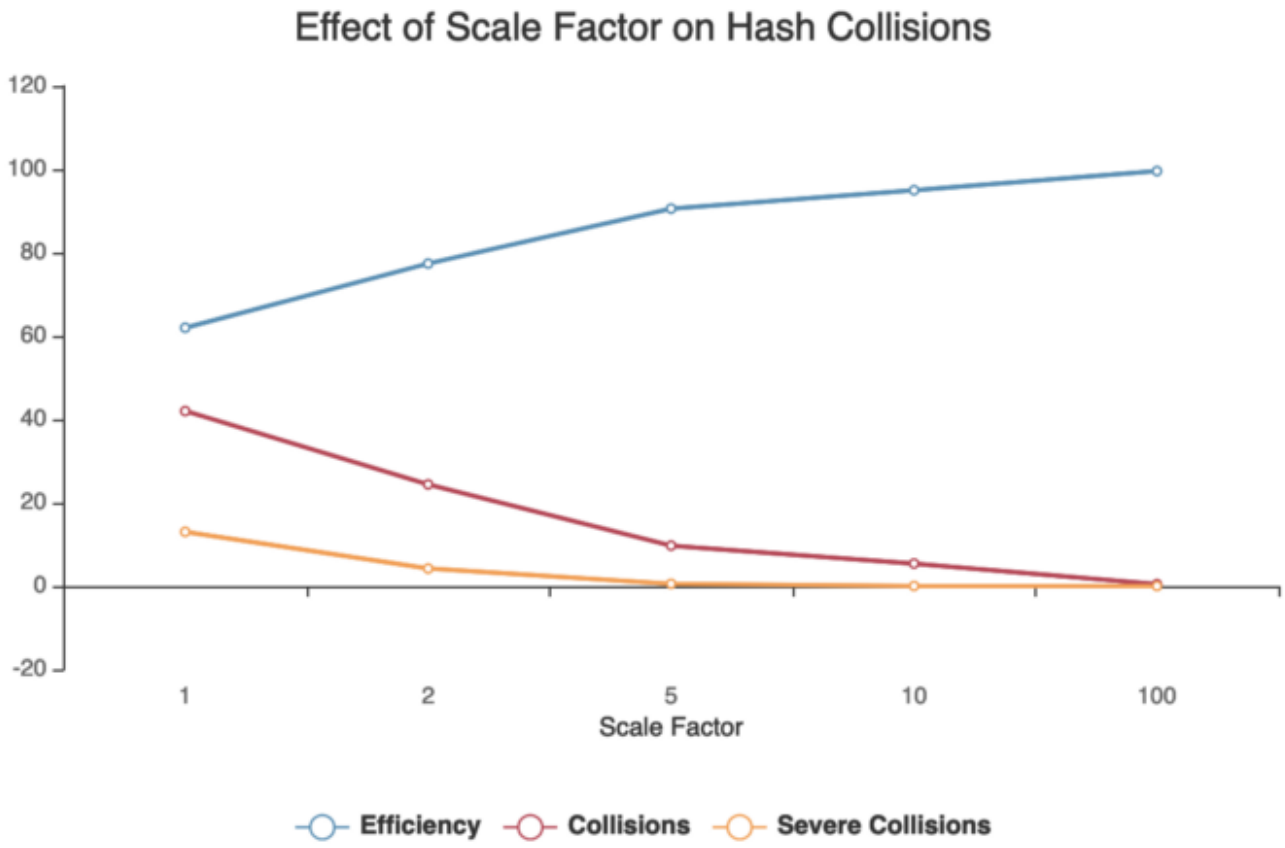
Now, there is a workaround — **partition scaling**.

In our previous examples, our number of output sPartitions is equal to our intended total file count. This causes hash collisions because of the same principles that surround the Birthday Problem— that is, if you're randomly assigning n objects to n slots, you can expect that there will be several slots with more than one object, and several empty slots. Thus, to fix this, you must decrease the ratio of objects to slots.

We do this by scaling our output partition count, by multiplying our output sPartition count by a large factor, something akin to:

```
df
.withColumn("rand", rand() % 5)
.repartition(5*365*SCALING_FACTOR, $"date", $"rand")
```

See here for updated analysis code, however, to summarize:



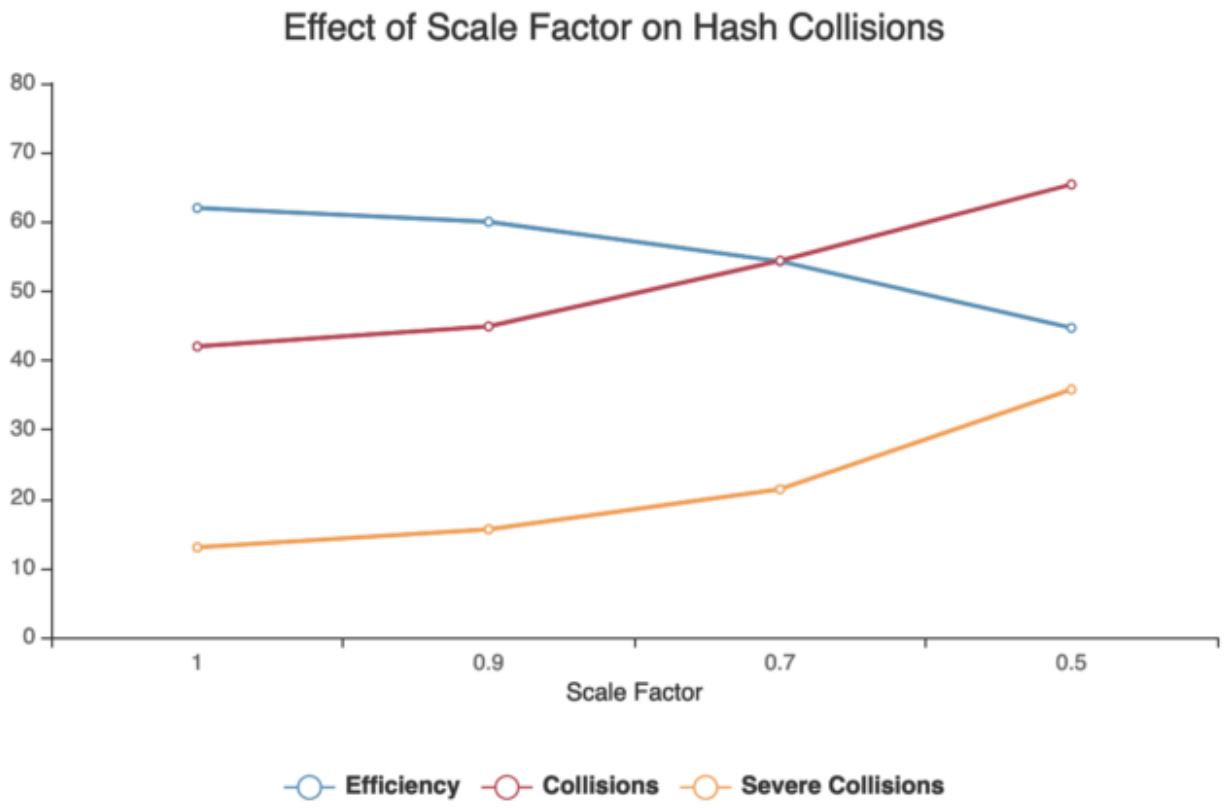
As our scale factor approaches infinity, collisions fairly quickly approach 0, and efficiency gets closer to 100%.

However, this creates another problem, where a huge amount of the output sPartitions will be empty. While these empty sPartitions aren't necessarily a deal breaker, they do carry some overhead, increase driver memory requirements, and make us more vulnerable to issues where, due to bugs, or unexpected complexity, our partition key space is unexpectedly large, and we end up writing millions of files again.

Default Parallelism and Scaling

A common approach here is to not set the partition count explicitly when using this approach, and rely on the fact that Spark defaults to your spark.default.parallelism value (and similar configurations) if you do not provide a partition count.

While, often, parallelism is naturally higher than total output file count (thus, implicitly providing a scaling factor greater than 1), this is not always true — I have observed many cases where developers do not tune parallelism correctly, and result in cases where the desired output file count is actually greater than their default parallelism. The penalties for this are high:



Scale factors below 1 get bad, fast.

Evaluation

This is an efficient approach if you can meet a few guarantees:

- hPartitions will have roughly equal file counts
- We can determine what the average partition file count should be

- We know, roughly, the total number of unique partition keys, so we can correctly scale our dataset.

In the examples, we assumed many of these things could easily be known; primarily, total number of output hPartitions, and number of files desired per hPartition. However, I think it's rare we can ask developers in broad to be able to provide these numbers, and keep them up to date.

This approach is not a bad one by any means, and will likely work for many use cases. That said, if you are not aware of its' pitfalls, you can encounter difficult-to-diagnose performance problems. Because of this, and because of the requirements to maintain file count related constants, I feel it is not a suitable default.

For a true default, we need an approach that requires minimal information from developers, and works with any sort of input.

Naive Repartition by Range

Repartition by range is a special case of repartition. Rather than applying RoundRobin or Hash Partitioners, it uses a special one, called a Range Partitioner.

A range partitioner splits rows across sPartitions based on the ordering of some given key, however, it is not performing a global sort. The guarantees it makes are:

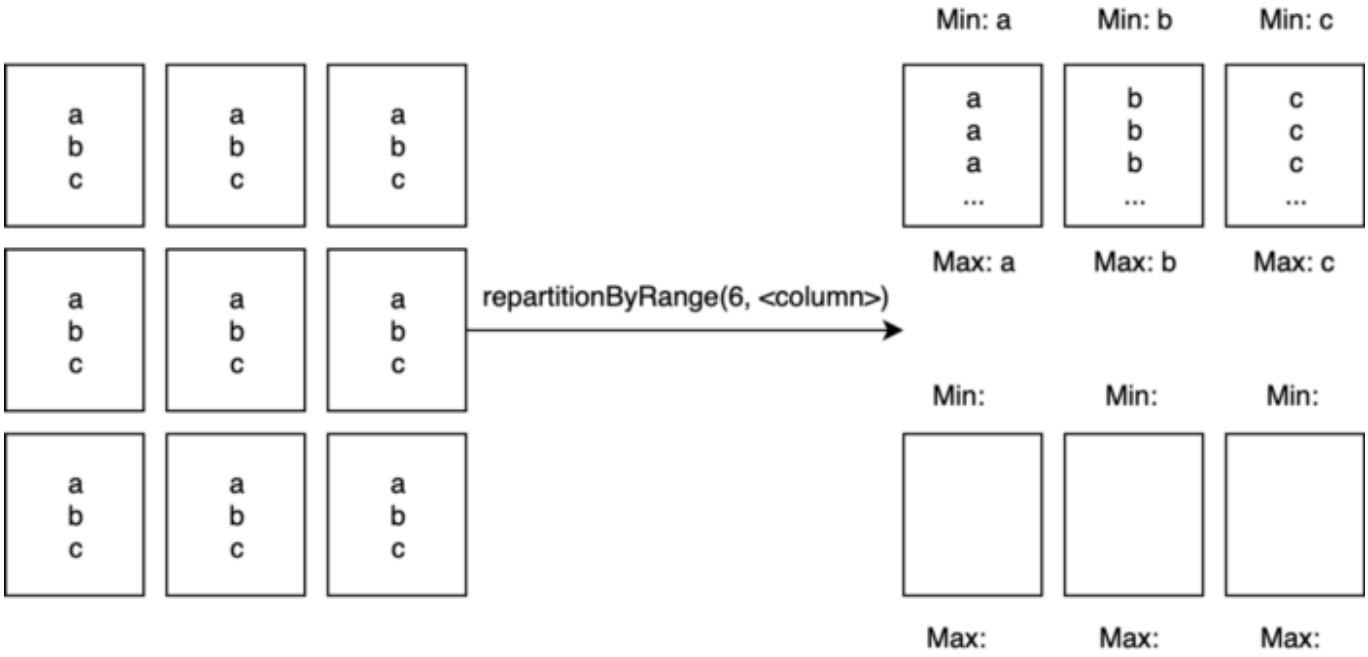
- All records with the same hash will end up in the same partition
- All sPartitions will have a "min" and "max" value associated with them, and all values between the "min" and "max" will be in that partition
- The "min" and "max" values will be determined by using sampling to detect key frequency and range, and partition bounds will be initially set based on these estimates.
- Partitions are not guaranteed to be totally equal in size, their equality is based on the accuracy of the sample, and thus, the predicted per-sPartition min and max values. Partitions will grow or shrink as necessary to guarantee the first two conditions.

To summarize, a range partitioning will cause Spark to create a number of "buckets" equal to the number of requested sPartitions. It will then map these buckets to "ranges"

of the specified partition key. For example, if your partition key is date, a range could could be (Min: “2018-01-01”, Max: “2019–01–01”). Then, for each record, compare the record’s partition key value to the bucket min/max values, and distribute them accordingly.

While this is, overall, fairly efficient, the sampling required to determine bounds is not free. To sample, Spark has to compute your whole dataset, so caching your dataset may be necessary, or at least, beneficial. In addition, sample results are stored on the driver, so driver memory must be increased — roughly 4–6G at most in our tests — but this will depend on your record and dataset size.

Evaluation



This has the same problem as df.repartition(6, <column>)

Repartition by range seems to deliver what we need, in theory. However, the first guarantee — all records with the same hash will end up in the same partition — is a sticking point. For our purposes, this makes it the same as a simple repartition, but more expensive, and thus unusable as it stands.

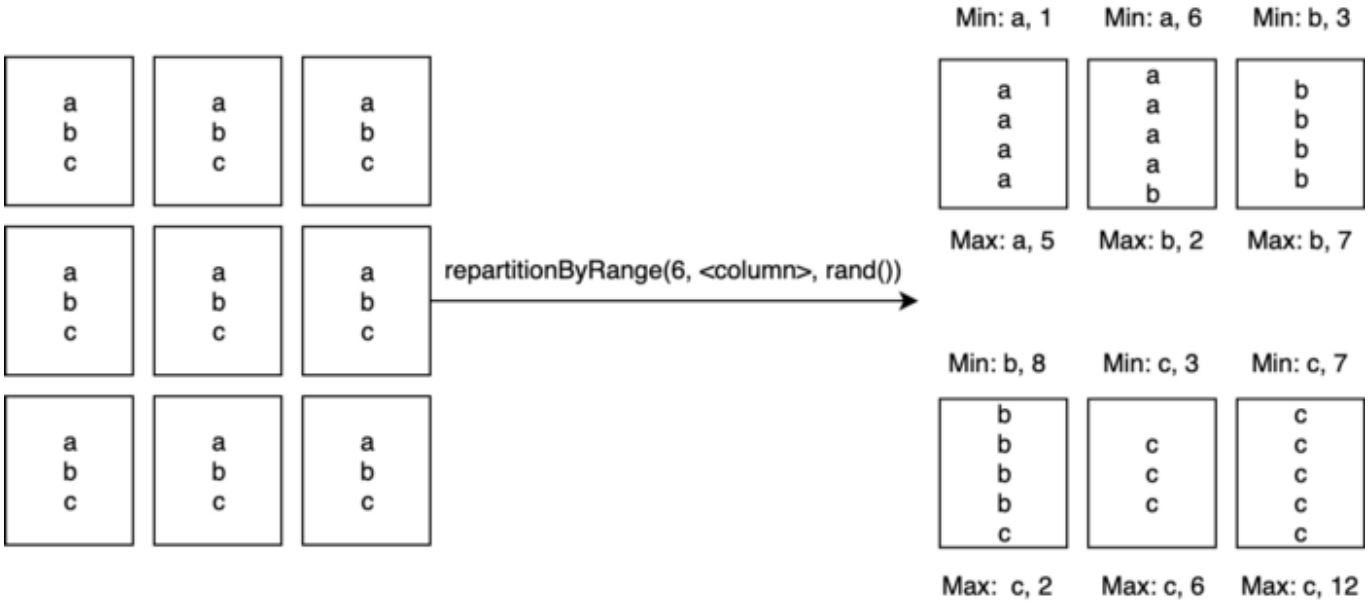
Repartition by Range with Additional Columns

However, we should not give up on repartition by range without a fight.

We will make two improvements:

First, we will hash the columns that make up our partition key. We don't actually care about the relative sorting of our keys, or the raw values of our partition keys at all, only that partition keys are differentiated, and a hash guarantees that. This reduces the cost of sampling (as all samples are collected in driver memory) as well as the cost of comparing partition keys to to min/max partition bounds.

Second, we will add a random key in addition to our hash. This will mean that, due to the hierarchal, key-based sort, we will effectively have all records for a given partition key present in multiple, sequential sPartitions.



Partitions won't be perfectly even because of the random factors, but you get pretty close.

An implementation of this would look something like:

```
val randDataframe = dataFrame.withColumn(
     "hash",
     hash(partitionColumns.map(new ColumnName(_)): _*)
 ).withColumn(
     "rand",
     rand()
)
randDataframe.repartitionByRange(
   fileCount,
   $"hash",
   $"rand"
).drop("hash", "rand")
```

On the surface, this looks similar to the repartition by columns + rand approach earlier, and you might suspect it has the same collision problems.

However, the hash calculation in the previous approach amounts to:

```
(date, rand() % files per hPartition).hashCode % sPartitions
```

Which ends up having a total number of unique hashes equal to your sPartition count.

Here, the hash calculation is simply:

```
(date, rand()).hashCode
```

Which has, effectively, infinite possible hashes.

The reason for this is the hash done here is only done to determine uniqueness of our keys, whereas the hash function used in the previous example is a two-tier system designed to assign records to specific, limited, buckets of records.

Evaluation

This solution works for all cases where we have multiple hPartition outputs, regardless of the count of output hPartitions/sPartitions, or the relative size.

Earlier, we discussed that determining file count at the dataset level, rather than the key level is the cheapest and easiest approach to perform generically, and thus, this approach requires no information to be provided by developers.

As long as the total file count provided to the function is reasonable, we can expect, at most, fileCount + count(distinct hash) files written to disk.

Bringing it all Together

So, to summarize, what should you be doing?

Determine your ideal file count

I recommend using count-based heuristics, and applying them to the entire dataset, rather than on a per-key basis. Encode these statically if you wish, to increase performance by skipping the count.

Apply a repartitioning scheme

Based on my above evaluations, I recommend using the following to decide what repartitioning scheme to use:

Use coalesce if:

- You're writing fewer files than your sPartition count
- You can bear to perform a cache and count operation before your coalesce
- You're writing exactly 1 hPartition

Use simple repartition if:

- You're writing exactly 1 hPartition
- You can't use coalesce

Use a simple repartition by columns if:

- You're writing multiple hPartitions, but each hPartition needs exactly 1 file
- Your hPartitions are roughly equally sized

Use a repartition by columns with a random factor if:

- Your hPartitions are roughly equally sized
- You feel comfortable maintaining a files-per-hPartition variable for this dataset
- You can estimate the number of output hPartitions at runtime or, you can guarantee your default parallelism will always be much larger (~3x) your output file count for any dataset you're writing.

Use a repartition by range (with the hash/rand columns) in every other case.

For many use cases, caching is an acceptable cost, so I suspect this will boil down to:

- **Use coalesce if you're writing to one hPartition.**
- **Use repartition by columns with a random factor if you can provide the necessary file constants.**
- **Use repartition by range in every other case.**

Looking Forward

Repartition by range works fairly well. However, it can be improved for this use case, by removing some of the guarantees and constraints it provides. We are experimenting with custom, more efficient versions of that repartition strategy specifically for managing your Spark file count. Stay tuned!

Interested in becoming an expert on repartitioning schemes? Want to solve the biggest of big data problems? Airbnb is hiring! Check out our open positions and apply!

Thanks to Egor Pakhomov.

Spark Hive Hadoop Data Engineering Data