Pseudo-Key Bucketing and the Skewed Join Key Problem in Apache Spark

Optimizations and Implemented Solutions for Memory Management, Communication, & Resource Allocation in a Distributed Computing System

Yu Chen

December 3, 2018

Abstract

Apache Spark is an open-source distributed computing framework that has experienced significant industry adoption for big data processing. Spark utilizes a variety of core operating system design principles and techniques to solves issues of scheduling, coordination, and memory management. This analysis provides an overview of design architecture decisions(ie. tradeoffs) that the library's developers and maintainers have made in order to solve two core issues within modern data ETL (extract-transform-load) and machine learning workflows: **1.** intelligent memory management to handle data volumes that require distributed systems, and **2.** skewed distributions of data that nullify the benefits of parallelization. We highlight predicate pushdown and lazy evaluation as strategies to mitigate the problems associated with memory-intensive compute jobs. The latter portion of this analysis provides an overview of existing implemented solutions to counteract the common problem of skewed join key distributions, which presents a whole host of communication, task allocation, and memory management questions by effectively reducing parallel computing system into sequential workflows. In particular, the Iterative Broadcast Hash Join and the Partial Manual Broadcast Join are evaluated. Finally, the author's own proposed Pseudo-Key Bucketing algorithm is introduced as a potential hybrid solution that seeks to maintain the archetypal Spark `SortMergeJoin` operation while redistributing keys uniformly to maintain computational parallelism.

1 Overview

The Apache Spark project was started by Matei Zaharia in 2013 and is primarily maintained and developed by the Apache Software Foundation, the University of Berkeley's AMPLab, and Databricks, where Zaharia serves as Chief Technologist. By building upon the concept of a distributed file system / resource management infrastructure and iconic Map-Shuffle-Reduce algorithm of Hadoop, Spark has experienced widespread adoption been adapted for a variety of data-related tasks, including

- **machine learning**: Spark's `MLlib` libraries contain a variety of supervised and unsupervised algorithms to perform classification, regression, clustering, dimensionality reduction, etc.
- **data ETL (extract-transform-load**: Spark utilizes core Hadoop-based algorithms like the map-shuffle-reduce to intake data, distribute it across workers, perform some mapping or transformation of the data, and initial aggregations
- **streaming data management**: Spark's streaming library is frequently used in conjunction with Apache Kafka, AWS Kinesis, etc. to create an end-to-end pipeline for handling and processing real-time, high-velocity data.

1.1 Spark is a Higher-Level Operating System

We begin by making the claim that Spark is simply another abstraction layer above what we traditionally conceive of as an operating system. This is an important distinction to make because Spark's internal architecture mirrors the same fundamental design patterns of modern operating systems. Andrew Tannenbaum once famously described distributed systems as

> *A collection of independent computers that appear to its users as one computer.*[10]

Spark, at its core, coordinates commodity infrastructure, negotiating resources, tasks, and communication protocols to provides users with the semblance of a unified platform for computation, data extraction, and machine learning modelling. The concept of hierarchical abstraction is central to Spark's internal design - for instance, its core data abstraction is the **Resilient Distributed Dataset**, but users typically work with data using higher level abtractions that build upon each other, such as the **Dataset**, **Dataframe** and **Spark SQL** abstractions.

1.2 Hierarchical Abstraction

The concept of abstraction a familiar one within computer science. One particularly instance, **hierarchical abstraction** (otherwise known as the *Onion Model*), implies that layers of abstraction build upon each other through some sort of consistent interface, such that each layer is an independent kernel with its complexity hidden behind exposed API calls:

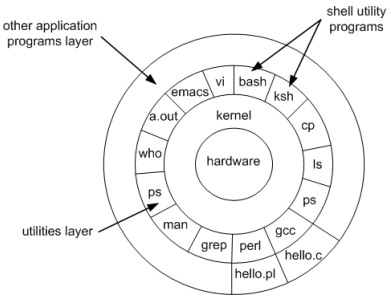


Figure 1: Operating System donut. From [17]

A modern operating system is designed deal with intricate, extremely complex human systems, and it is not a coincidence that this **onion architecture** is a popular choice for managing complexity and unifying

disparate devices and connected systems. At a high-level, Spark shares many of the same fundamental tasks of intricate systems[1]:

- **Communication**: what are the protocols and patterns that should be followed to exchange messages efficiently between various nodes within a Spark cluster, and various threads within an executor?
- **Scheduling and Synchronization**: how to ensure that each worker node's jobs are coordinated to reduce idle time and overall lead time?
- **Memory Management**: given the enormous size of datasets, how to ensure Spark efficiently allocates and frees memory to match data processing requirements?

Indeed Spark parallels the same onion architecture:

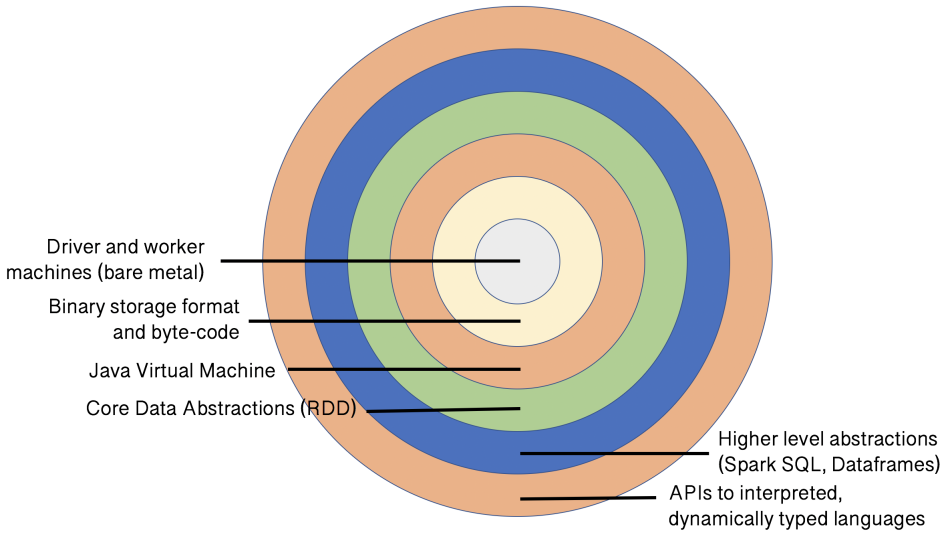


Figure 2: Spark's unified data processing engine as a sequence of hierarchical abstractions, with each layer providing more granular control over execution, memory management, etc.

Spark ultimately runs on the Java Virtual Machine (JVM), but exposes a variety of API libraries for interoperability with other languages. Since much of data science and machine learning within industry is conducted in either Python or R, Spark offers PySpark and SparkR, respectively, as higher level API frontends. Typically, these frontend APIs incur additional performance costs, since PySpark establishes a JVM process and accompanying socket to communicate with the JVM. Moreover, dynamically-typed, interpreted languages like Python and R offer less opportunity for low-level optimizations that a compiler for C can make (for instance, loop permutations, pipelining, etc. to minimize cache misses). Thus, Spark shares many of the same design patterns that a modern operating system features because it likewise shares many of the same fundamental problems- in terms of managing complexity, offering abstractions, optimizing the common cases, and unifying disparate infrastructure.

1.3 The Resilient Distributed Dataset

At the core of the Spark onion is its main data abstraction- the **resilient distributed dataset (RDD)**. An RDD is an iterable collection of elements that is partitioned worker cluster nodes. Two types of primary operations can be applied to an RDD:

1.3.1 Transformations

The core operator within Spark are transformations, function that accepts an RDD as input and returns a newly created RDD or RDDs. Common Spark transformations include `map`, `join`, `filter`, `reduce`. For example,

the following code accepts text from a log file, splits each text line in the file by a delimiter ::, performs a `filter` transformation, and returns a newly created RDD of only log entries emitted from a certain IP address:

```
 # read from disk a text file called server_logs.txt, load into memory and distribute across worker
     nodes
all_logs = sparkContext.textFile("server_logs.txt")

# tokenize each line of the input (assume that the IP address of the log entry comes before the ::
    and the log entry body comes afterwards)
tokenized_logs = all_logs.map(lambda x: x.split("::").map(lambda x: (x[0], x[1:])

# filter the RDD so that only entries coming from 10.0.0.20 remain.
filtered_logs = tokenized_logs.filter(lambda x: x[0] == "10.0.0.20")
```

We perform two mappings - once to split the text line into chunks based on the `::` delimiter, and another time to convert the RDD into a tuple, with the first element as the **key** and the second element as the **value**. This is a common data structure pattern within RDD, and many transformations and actions depend upon having RDDs in a key-value format.

1.3.2 Actions

Actions are operations that actually evaluate and trigger the transformations. Examples of these are `count()`, `collect()`, `sum()`. Usually, this has the effect of collecting the output of the action onto the driver node (the entrypoint to the Spark application). Thus, it is critical than any call to an action returns a result that can fit into memory on a single local machine. Due to Spark's lazy evaluation model (discussed in later sections), ordering an action correctly can be the difference between inefficient data transfer to and from the driver node, and a fully parallelized computing cluster, with tasks ideally spread uniformly across multiple partitions.

1.3.3 Partitions

All RDDs are partitioned across various different worker nodes, so that each transformation applied to them can be operated on in parallel. For instance, we can see that `tokenized_logs` is by default split into 9 partitions on my MacBook Pro:

```
scala> tokenized_logs.partitions
res1: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.HadoopPartition@3c1, org.apache.spark.rdd.Hado
opPartition@3c2, org.apache.spark.rdd.HadoopPartition@3c3, org.apache.spark.rdd.HadoopPartition@3c4, org.apache.spa
rk.rdd.HadoopPartition@3c5, org.apache.spark.rdd.HadoopPartition@3c6, org.apache.spark.rdd.HadoopPartition@3c7, org
.apache.spark.rdd.HadoopPartition@3c8, org.apache.spark.rdd.HadoopPartition@3c9)

scala> tokenized_logs.partitions.size
res2: Int = 9
```

Figure 3: Each RDD contains references in a Scala Array to `Partition` instances. Spark will automatically create one partition for each HDFS (Hadoop Distributed File System) block [4]

A popular technique for optimizing utilization and memory consumption across a Spark cluster is to increase the number of partitions so that a dataset is evenly divided up amongst all workers. For instance, we can repartition `tokenized_logs` to be split across 64 partitions:

```
scala> val repartitioned_logs = tokenized_logs.repartition(64)
repartitioned_logs: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[9] at repartition at <console>:25

scala> repartitioned_logs.partitions.size
res6: Int = 64
```

Figure 4: Repartitioned RDD from 9 partitions to 64 partitions. Increasing partition size provides more granular control for the Spark Scheduler to decide where (which worker node) and when (within a stage) to run a task. However, more partitions also implies more costs associated with communication and coordination.

Therefore, when we actually perform the transformation `tokenized_logs.filter(lambda x: x[0] =="10.0.0.20")`, Spark is actually performing 64 concurrent filter operations. The significance of correct partitioning can be highlighted with a toy use case where our cluster has three worker nodes, each with one core, and our data has been divided up into 4 partitions:

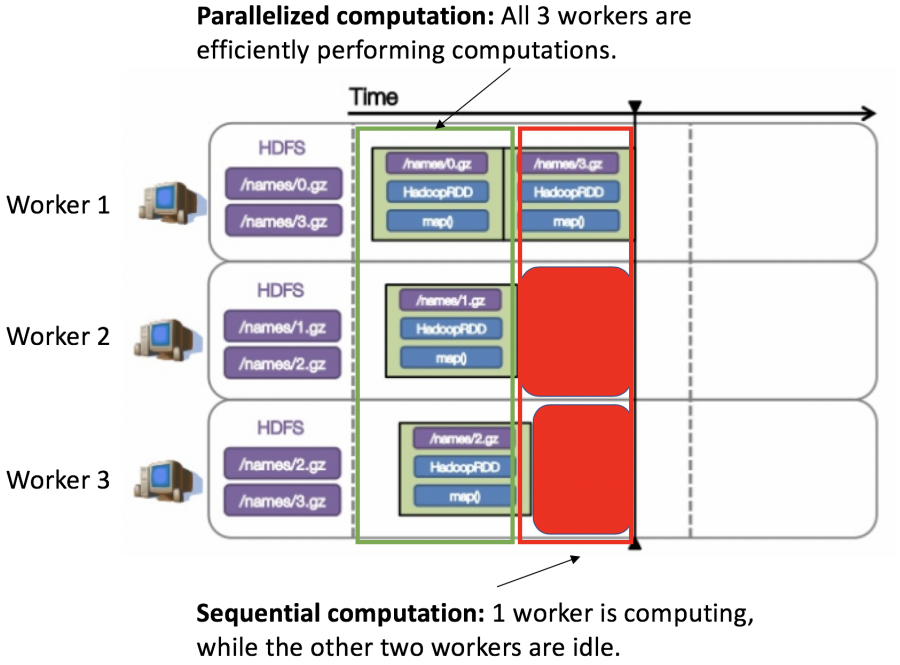


Figure 5: An inefficient partitioning scheme that essentially nullifies Spark's distributed computing benefits. Adapted from slides by Aaron Davidson of Databricks.[5]

In this case, while at first computation proceeds in a parallelized fashion, Worker 2 and Worker 3 nodes finish their assigned tasks prior to Worker 1, which has been assigned two tasks because of the uneven partitioning. Thus, for the second half of this stage, computation is sequential. Worse, if this stage is only one of several over stages of computation, work must wait until Worker 1 completes before the next stage can be commenced.

1.3.4 Lazy Evaluation

The other core principle behind Spark's memory management optimization strategy centers around RDDs being lazily evaluated. If you actually attempt to access `filtered_logs`, you will not receive any results:

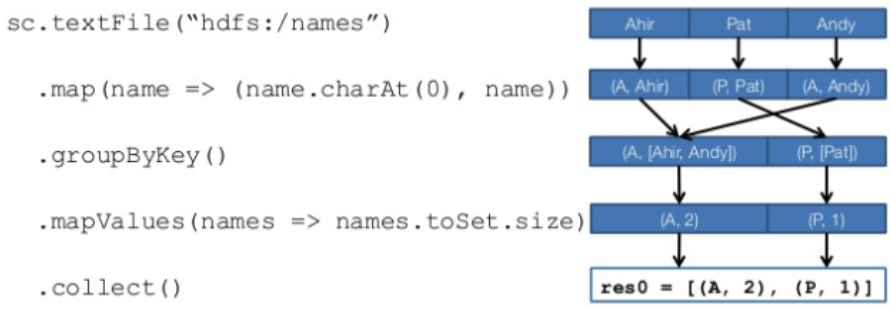
```
scala> filtered_logs
res4: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[4] at map at <console>:25
```

Figure 6: The result of accessing the data structure called `filtered_logs`

What is returned is not the actual filtered and mapped data, but rather the in-memory reference to a `MapPartitionsRDD` object, with an ID of 4. At a high level, RDDs construct a directed acyclic graph (DAG) of its lineage, and the calls to `filter()`, `map()`, and even `textFile()` simply append additional transformations or pointers to the graph, rather than actually evaluating any result. Note also the ID of RDD (4) is a globally incremented value to uniquely identify the RDD. Therefore, this particular RDD was the 4th RDD created within the `SparkContext` object. In fact, notice in the below Scala script for loading in Amazon food reviews and filtering for only reviews that contain the word "good" that the ID increments with each new transformation:

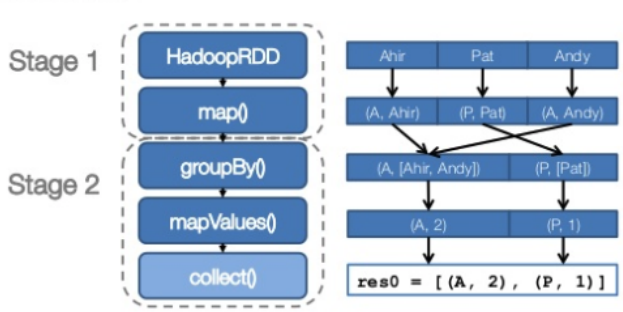
```
scala> val raw_reviews = sc.textFile("Reviews.csv")
raw_reviews: org.apache.spark.rdd.RDD[String] = Reviews.csv MapPartitionsRDD[1] at textFile at <console>:24

scala> val mapped_reviews = raw_reviews.map(line => line.split(","))
mapped_reviews: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:25

scala> mapped_reviews.filter(line => line(8).toLowerCase contains "good")
res0: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[3] at filter at <console>:26
```

It will become apparent at runtime that regardless of whether or not there are 10 or 10 billion records, the above script will completely in almost roughly the same amount of time. In fact, Spark will accept any string in its `sparkContext.textFile()` call, as it does not actually fetch and load the text data until an action is called. We can see, for instance, `filteredReviews` is essentially the aggregation of previous RDDs created from transformations. Consider, for instance, the case where we attempt to find the **number of distinct numbers per letter in the English alphabet** for a given text file of employee names:



Step 1. Source code mapped to visualized data flow for distinct name job.

Transformations within Spark can be either **narrow** or **wide**. Narrow transformations can be chained together so that they are essentially fused into the same function call (and thus share the same stack, memory, and local variables), in a process called **pipelining**. In this way, grouping together a series of narrow transformations allows for pipelining optimizations that help to minimize the number of operations performed and percentage of cache misses. Note that some transformations and operations create intermediate RDDs - for instance, the `sc.textFile()` call generates a `HadoopRDD` than is then itself transformed into a `MappedRDD`. Image from Davidson talk.[5]



Step 2. Mapping from source code to RDDs.
Note that some transformations and operations make create intermediate RDDs - for instance, the `sc.textFile()` call generates a `HadoopRDD` than is then itself transformed into a `MappedRDD`. Here, the task is broken up into two separate stages because it is impossible to pipeline across a `groupBy()` - data is shuffled and rearranged by key across the clusters, so that all data points with a key of `A` (ie., names that start with A) will be moved onto the same machine. Image from Davidson talk.[5]

1.3.5 Immutability and Stages

Since we have seen that new RDDs are created upon each transformation, RDDs are *immutable*. They do not change state once created. `raw_reviews` is a different RDD in reference (memory address) than `mapped_reviews`. Why is lazy evaluation in Spark justified, when traditionally within operating system design eager evaluation is typically more efficient (ie., prefetching)? Recall the **cost of operation** C_o is defined as

$$C_o = C_s + C_i \tag{1}$$

Where C_s is the startup cost and C_i is the per-item cost.[2]

	Spark Unified Compute Model	**Traditional Operating Systems**
Startup Cost (C_s)	Reading/writing to disk is always relatively slow, as well as the initial partitioning of RDDs into worker nodes. Also includes scheduling and health checking of nodes, and communication with individual partitions (which increases as the number of partitions grows).	**Moderate** - prefetching data and loading from disk / memory into registers or L1-L3 caches is costly, but not as costly as I/O.
Per-Item Cost (C_i)	**Varies, but typically quite high** - include all costs from traditional operating systems, plus relatively efficient pipelined mathematical operations to expensive broadcast, map, shuffle, reduce operations across distributed cluster over a network (which may also involve deserialization/serialization and garbage collection)	**Generally low** - typically processes operate on small memory sizes, and may incur costs associated with interrupts (context switching) and page faults

Spark jobs are by nature extremely expensive - as a general rule of thumb, it is used for handling data volumes that cannot fit within memory of a single machine. Thus, **it incurs all the traditional per-item costs of an operating system, plus the associated communication and synchronization costs of distributed systems**. A typically job, however, will only ever need to operate on a small percentage of the whole dataset (often, data engineers and scientists may immediately filter down a large text file by a specific set of keys, or time interval). Thus, we can derive a heuristic for when lazy evaluation is beneficial within Spark by comparing eager evaluation with lazy evaluation cost of operation for N items. Our lazy cost is defined our cost of startup plus the fraction of per unit cost we incur:

$$C_o = \sum_{i=1}^{N} (C_s + C_i(p)) \tag{2}$$

Setting this equal to our eager evaluation cost, we can approximate a decision boundary for p, the percentage of the dataset that we plan to actually operate on:

$$C_s + \sum_{i=1}^{N} C_i = \sum_{i=1}^{N} (C_s + C_i(p)) \tag{3}$$

$$C_s + N(C_i) = N(C_s) + N(p(C_i))) \tag{4}$$

$$N(C_i) - N(p(C_i)) = N(C_s) - C_s \tag{5}$$

$$N(C_i)(1-p) = (N-1)C_s \tag{6}$$

$$1 - p = \frac{(N-1)C_s}{N(C_i)} \tag{7}$$

$$p = 1 - \frac{C_s}{C_i} \tag{8}$$

The N is eliminated in step 7 with the assumption that N will be a reasonably large number such as $\frac{N-1}{N} \approx 1$. As a rule of thumb, the percentage of the dataset that we operate on p must decrease (we do less work) as the ratio of cost between C_s and C_i increases - if startup time is high relative to C_i, then we need to work on an increasingly smaller fraction of the dataset to justify the costs of lazy evaluation. Obviously note that p would evaluate to a nonsensical negative value if C_s is greater than C_i, in which case clearly a eager evaluation strategy would make more sense.

1.3.6 Predicate Pushdown and Early Binding

The concept of lazy evaluation is extended as part of Spark more recent Project Tungsten optimizations. While RDDs are essentially the atomic data unit of Spark, more recent versions of the framework build upon higher level abstractions, including Spark Datasets and Dataframes:

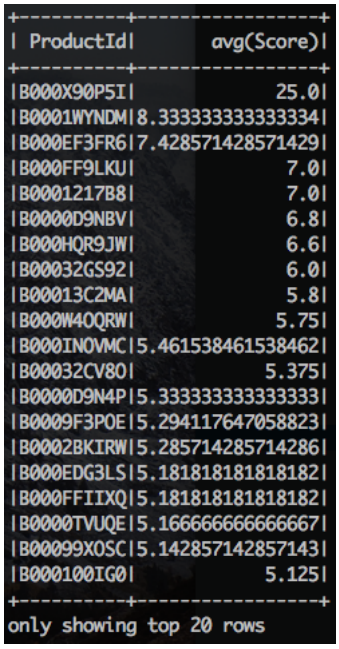
- **Dataset:** a data structure that includes strongly typed data that exposes a variety of high-level APIs for users to reason about their data with. For instance, the RDD's `map()`, `reduceByKey()`, and other transformation functions are replaced with higher-level function calls.
- **Dataframe:** a Spark Dataset, but of type `Row`. The below code reads in a text file of about 600,000 Amazon food product reviews as a Dataframe, with Spark automatically inferring schema and data types, and provides the top rated products that begin with `B000`:

```
spark.read.option("header", "true"). /* read in a file with schema defined in header */
      csv("Reviews.csv").      /* define text file with file path and delimiter */
      groupBy("ProductId").   /* map RDD into key value pairs where ProductID is the key and
          rest of columns are value*/
      agg(avg("Score")).  /* for each key, compute the average of the Score column */
      orderBy(desc("avg(Score)")). /* sort by this computed column */
      filter(col("ProductId").startsWith("B000")). /* filter to show only products with B000...
          IDs */
      show()
```

There is an even higher level of abstraction available, in the form of Spark SQL, where instead of declaratively stating transforms for Spark to perform, the user writes expressive queries and Spark's internal Catalyst engine creates the DAGs and compute strategy:

```
/* create in memory representation of data as a view */
spark.read.option("header", "true").csv("Reviews.csv").createOrReplaceTempView("raw_reviews")

/* query this data */
spark.sql("SELECT ProductID, AVG(Score)
      FROM raw_reviews
      GROUP BY ProductID
      HAVING ProductID LIKE 'B000%' ORDER BY 2 DESC").show()
```

In either case (through declarative transformations using Dataframes, or through expressive Spark SQL queries), the end result is



```
+----------+------------------+
| ProductId|       avg(Score)|
+----------+------------------+
|B000X90P5I|              25.0|
|B0001WYNDM|8.333333333333334|
|B000EF3FR6|7.428571428571429|
|B000FF9LKU|               7.0|
|B0001217B8|               7.0|
|B0000D9NBV|               6.8|
|B000HQR9JW|               6.6|
|B00032GS92|               6.0|
|B00013C2MA|               5.8|
|B000W4OQRW|              5.75|
|B000INOVMC|5.461538461538462|
|B00032CV80|             5.375|
|B0000D9N4P|5.333333333333333|
|B0009F3POE|5.294117647058823|
|B0002BKIRW|5.285714285714286|
|B000EDG3LS|5.181818181818182|
|B000FFIIXQ|5.181818181818182|
|B0000TVUQE|5.166666666666667|
|B00099XOSC|5.142857142857143|
|B000100IG0|             5.125|
+----------+------------------+
only showing top 20 rows
```

Calling `results.explain(extended=true)`, where `results` is the Dataframe *prior* to the `collect()` action, provides an explanation of how Spark constructed the DAG:

```
== Analyzed Logical Plan ==
ProductID: string, avg(CAST(Score AS DOUBLE)): double
5 Sort [avg(CAST(Score AS DOUBLE))#895 DESC NULLS LAST], true
4 +- Filter cast(ProductID#744 LIKE B000% as boolean)
   3 +- Aggregate [ProductID#744], [ProductID#744, avg(CAST(Score AS DOUBLE))#895]
      2 +- SubqueryAlias `raw_reviews`
         1 +- Relation[...column names omitted for brevity...] csv
```

Figure 9: The parsed logical plan from the query provided.

There is a fair bit of inefficiency in how we execute our query:

1. We only care about two columns: `ProductID` and `Score`. However, in our first Dataset declarative transformation, we perform a `groupBy("ProductId")` operation on all 10 columns of the dataset, include the extremely large `CommentText` column.

2. Our dataset has nearly 600,000 user reviews, but we are interested in only about 180,000 of them (reviews pertaining to products that start with ID of `B000`. However, in both the declarative and expressive approaches, we perform an expensive `groupBy` operation on all reviews, and then filter for only those IDs beginning with `B000`.

The example dataset used here is extremely small (330MB) and can easily fit entirely into memory on a local machine. However, when working with production-level Big Data, the memory ramifications of these inefficiencies are enormous. While a Spark worker will attempt to **spill** to disk if it cannot fit all of its data and meta-data into memory, holding two columns versus 10 columns in memory will yield vastly different performance in terms of the JVM's **garbage collection** behavior. It is a well-known Spark issue to experience "GC storms", where the garbage collection execution time of a worker process completely overshadows actual compute and execution time. [6]

Spark's physical plan describes the actual implemented execution strategy for our underlying transformations:

```
== Physical Plan ==
*(3) Sort [avg(CAST(Score AS DOUBLE))#895 DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(avg(CAST(Score AS DOUBLE))#895 DESC NULLS LAST, 200)
   +- *(2) HashAggregate(keys=[ProductID#744], functions=[avg(cast(Score#749 as double))])
      +- Exchange hashpartitioning(ProductID#744, 200)
         +- *(1) HashAggregate(keys=[ProductID#744], functions=[partial_avg(cast(Score#749 as double))],
            +- *(1) Project [ProductId#744, Score#749]
               +- *(1) Filter (isnotnull(ProductID#744) && StartsWith(ProductID#744, B000))
                  +- *(1) FileScan csv [ProductId#744,Score#749] Batched: false,
                  DataFilters: [isnotnull(ProductId#744), StartsWith(ProductId#744, B000)]...
                  PushedFilters: [IsNotNull(ProductId), StringStartsWith(ProductId,B000)]
```

Figure 10: The physical plan (query execution plan) generated and executed by Spark using its `WholeStageCodeGen` optimizer.

Notice that the filter for product IDs has been "pushed down" several steps, immediately as part of the scan operator. There's several optimizations here to note:

- **Predicate Pushdown**: the filter is applied immediately at the scan level, resulting in significantly fewer rows (from about 600 to 180 thousand rows) held in memory. This also directly impacts the `groupBy` operation, which requires a shuffling/redistribution of data across worker nodes and is typically an inefficient bottleneck in a Spark task.
- **Partial Aggregations:** given that exchanges across nodes is expensive, Spark performs a `partial_avg` for each partition, and then sends only this result, rather than the entire list of ratings across the network. The below line in the Physical Plan

  ```
  HashAggregate(keys=[ProductID#744], functions=[partial_avg(cast(Score#749 as double))],
  output=[ProductID#744, sum#901, count#902L])
  ```

  indicates that for each partition, Spark summed up the total score, and also the count. Notice that the intermediate outputs of this `HashAggregate` operation were a `sum` and `count` scalar value per `ProductID`. These values were exchanged across cluster to other partitions that contained the same key in a future step:

  ```
  HashAggregate(keys=[ProductID#744], functions=[avg(cast(Score#749 as double))],
  output=[ProductID#744, avg(CAST(Score AS DOUBLE))#895])
  ```

  Here, the actual average was calculated using the `sum` and `count` intermediate outputs from the prior `partial_avg`.
- **Repartitioning**: Remember that originally our reviews data was divided up into 9 partitions. Spark automatically executed `rangepartitioning()` and `haspartitioning()` on the average scores and `Product IDs`, respectively, to distribute them evenly amongst the cluster.

1.3.7 Encoded Binary and Columnar Storage Formats

2 Skewed Key Distributions: A Bottleneck for Distributed Systems

The **Pareto principle** has been applied to a variety of domains, and states very generally that for a given phenomenon, about 80% of the observed effects can be attributed to 20% of the causes. Indeed, many observed phenomena follow this pattern:

- During a Waterfall software development lifecycle, with proper project management, the cost of labor can be reduced by 80% while ultimate software output and quality is reduced by only 20%.[7]
- As of 1989, the richest 20% of the world controlled about 82.7% of the world's gross domestic product (GDP).[8]
- Microsoft CEO Steve Ballmer famously claimed in 2002 that about 80% of all errors reported in Windows and the Office product suite were due to 20% of bugs.[9]

This phenomenon is frequently found within underlying data distributions themselves, and will nullify any benefit from parallelization offered by Spark when joining across datasets. In fact, a co-sponsored study by Oracle and the University of Arizona found that even on a small dataset (7.8MB) of job assignments for University of Arizona employees that contained only 3.6% duplicate key values (in other words, over **96%** of the records had unique keys and therefore new skew), the number of physical reads to the database file system was **3** times more frequent and **5x** times longer runtime that of a similar-sized join with no skewed keys.[14].

2.1 Shuffle Hash and Sort Merge Joins

Joins are typically quite expensive operations within distributed systems largely because each rows with identical keys of the relevant RDDs must be shuffled across network to colocate in the same data partition (the exception to this is the **Broadcast Join**, where one of the to-be-joined tables is small enough to fit into memory on each partition of the larger table, and therefore can simply be broadcast to all partitions in order to avoid a costly cross-network shuffle). However, with the proliferation of normalized tables in RDMS (relational database management systems), joins are increasingly vital for data processing tasks, business intelligence visualization analytics, and preparing the feature space for machine learning models.

In an ideal case, the distribution of keys within the join key of two datasets is more or less uniform, with a probability mass function (pmf) of $p(k) = \frac{1}{|K|}$, where k is any key, and $|K|$ is the length of the set of all keys in the join column. In a *perfect* case, there would be a direct one-to-one mapping between the join keys of both tables. In any case, uniformity in terms of key partitions allows linear (or log-linear) algorithms to be used during the merge step. During the shuffle step, Spark will uniformly distribute the data such that all rows with the same join key find themselves on the same executor. It will then locally sort the data per executor so that all rows with the same join key are colocated together in memory. This allows for a relatively efficient local merge task. See Appendix C for a visualization of this join annotated using Keevil and Driesprong's Spark Summit visualizations. However, datasets in the real world often tend to follow Pareto's Principle, with a significant portion of rows belonging to only a handful of keys.

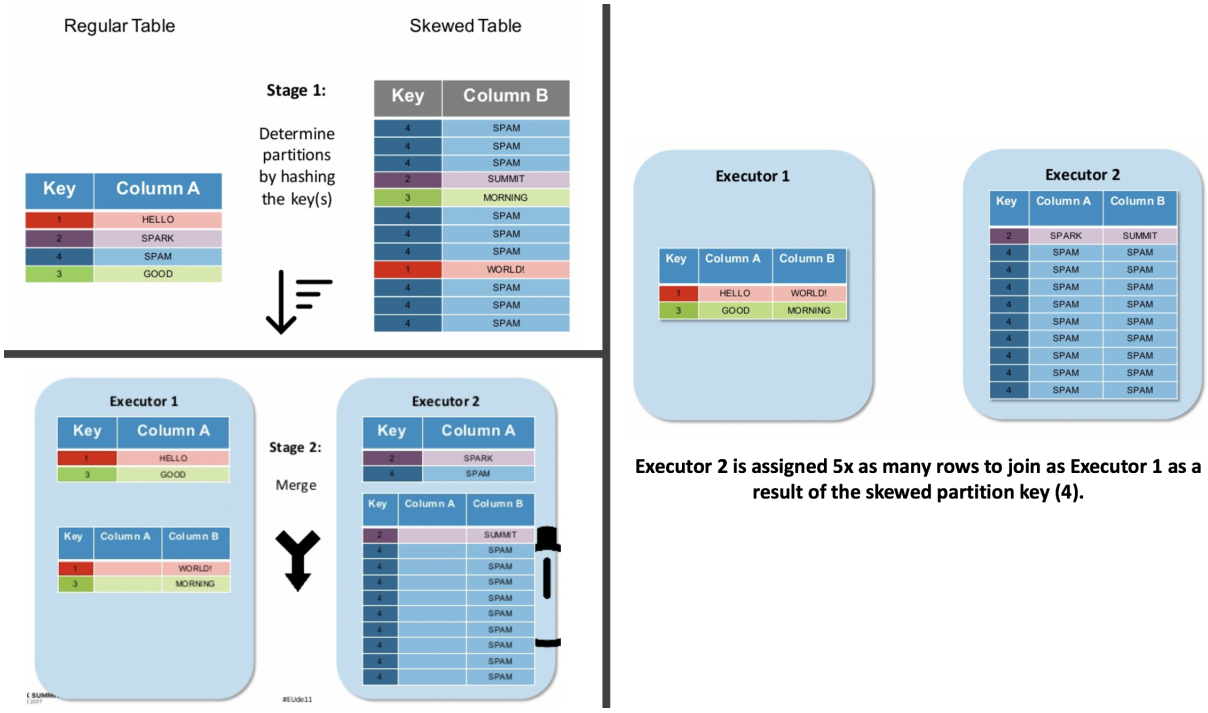


Figure 11: A sort merged join performed on a skewed partition key, from Keevil and Driesprong's Spark Summit talk.[11]

The negative effects of this result are fourfold:

- **The skewed partition key becomes the bottleneck (critical path) of the compute workflow**. Since Spark typically chunks tasks into stages, individual executors can be at different points within a stage, but must wait for all executors to complete their assigned tasks this stage prior to advancing onto the next stage's tasks.
- textbfWe are essentially performing a costly shuffle step for limited benefit to parallelization. We're moving a large chunk of data from one partition of an executor to another partition, and incurring all the costs of serialization, network communication, and deserialization with very little parallelization benefit.
- **The skewed partition key may not actually fit into memory**. Imagine we have 40 million rows with a key of `SPAM` and only 2 million rows for every other key. The unfortunate executor assigned to merging `SPAM` will end up either frequently spilling to disk or throwing an `OutOfMemoryError`.
- In addition, **the level of parallelism obtained by the `SortMergeJoin` is bounded by the number unique keys there are**- a skewed distribution of keys inherently reduces the number of distinct keys. Since each key is uniqued partitioned during the shuffle step to a worker node, fewer unique keys means fewer opportunities for scaling out in terms of compute worker instances to parallelize a given computation.

2.1.1 Iterative Broadcast

Rob Keevil and Fokko Driesprong proposed an iterative broadcast solution at Spark Summit EU 2017 as a potential solution to the skewed partition key problem. Although Spark doesn't have a higher-level API call to trigger a broadcast hash join, it can be manually implemented in the algorithm described below by Warren and Karau:

```
def manualBroadCastHashJoin[K : Ordering : ClassTag, V1 : ClassTag,
V2 : ClassTag](bigRDD : RDD[(K, V1)],
```

```
  smallRDD : RDD[(K, V2)])= {
  val smallRDDLocal: Map[K, V2] = smallRDD.collectAsMap() /* converts the smaller RDD into a HashMap
      */
  val smallRDDLocalBcast = bigRDD.sparkContext.broadcast(smallRDDLocal) /* broadcast this HashMap to
      each partition */
  bigRDD.mapPartitions(iter => {
   iter.flatMap{
    case (k,v1 ) =>
     smallRDDLocalBcast.value.get(k) match { /* iterate through the partition rows of the bigger
         RDD, checking if the key is found in the HasMap */
      case None => Seq.empty[(K, (V1, V2))]
      case Some(v2) => Seq((k, (v1, v2)))
     }
   }
  }, preservesPartitioning = true) */ this is an important parameter to prevent Spark from attempt a
      shuffle
 }
 //end:coreBroadCast[]
}
```

Instead of using `map()`, we use `mapPartitions()`. The core difference is that `mapPartitions()` is called only once per partition, where all the elements of the RDD in a given partition are converted into a sequential stream available using an `Iterator[T]`. Because `mapPartitions()` is called only once per partition whereas `map()` is called once per element, it is typically faster since it avoids reinitialization of the function stack frame each time. `mapPartitions()` is particularly useful a function contains some expensive task (like connecting to a database)[15].

The Broadcast Hash Join is widely used when the smaller to-be-joined RDD can fit into memory on each of the worker nodes. If this is the case, Spark can simply broadcast the smaller RDD to each worker, and then perform the merge join locally. This avoids the expensive shuffle step, and also maintains parallelism across workers (since the data is not shuffled by the skewed key, so each partition will have more or less an equal number of merge tasks to perform). However, in practice, sometimes datasets are so large that even the smaller RDD cannot fit into memory on worker nodes. While Spark has evolved a wide class of optimizations (spilling to disk, a `UnifiedMemoryManager` that dynamically switches between storages and execution memory), ultimately, the Broadcast Hash Join's bottleneck becomes the memory size of the smaller RDD. Keevil and Driesprong, two data engineers from GoDataDrive, have proposed an algorithm that addresses this issue, consisting of the following steps:

1. Split up the smaller RDD into chunks. Decide upon how many chunks to split the smaller RDD into by setting the `Config.numberOfBroadcastPasses`, which is a hyperparameter specifying how many iterations (or passes) to make.
2. For each iteration, perform the following actions:
3. Add an extra column to the RDD called `pass`, uniformly distributing the rows so that each pass has an equal number of rows. Then select only that specific chunk:

```
val out = result.join(
  broadcast.filter(col("pass") === lit(iteration)), // filter for only 1/Nth of the data,
      where N is the number of passes
  Seq("key"),
  "left_outer"
).select(
  result("key"),
  coalesce(
    result("label"),
    broadcast("label")
  ).as("label")
)
```

4. Write the filtered dataframe out to disk (Spark will automatically attempt to pipeline tasks, and will therefore attempt to join the filtered RDDs together. This defeats the purpose of the iterative broadcast and we will again run out of memory attempting to broadcast the smaller RDD to each partition. Thus, a `collect()` and write to disk operation is performed to force Spark to place a stage barrier within each iteration:

```
SparkUtil.dfWrite(out, tableName) /* a simple helper function created by Keevil and
    Driesprong to write to Parquet file *
```

5. Recursively call `iteratieBroadcastJoin()` function again, simply incrementing the `iteration` number by 1. If `iteration ==Config.numberOfBroadcastPasses`, then simply return the resulting dataframe

The algorithm yielded incremental improvements over Spark's `SortMergeJoin` for datasets over approximately 14 million rows. The figures below show the execution time of three different algorithms: **1.** An Iterative Broadcast Join with 2 chunks, **2.** An Iterative Broadcast Join with 3 chunks, and **3.** A traditional `SortMergeJoin`. The smaller the number of chunks (iterations), the faster the execution time - with each iteration there is a significant cost of startup, including the collect actions and write to Parquet.

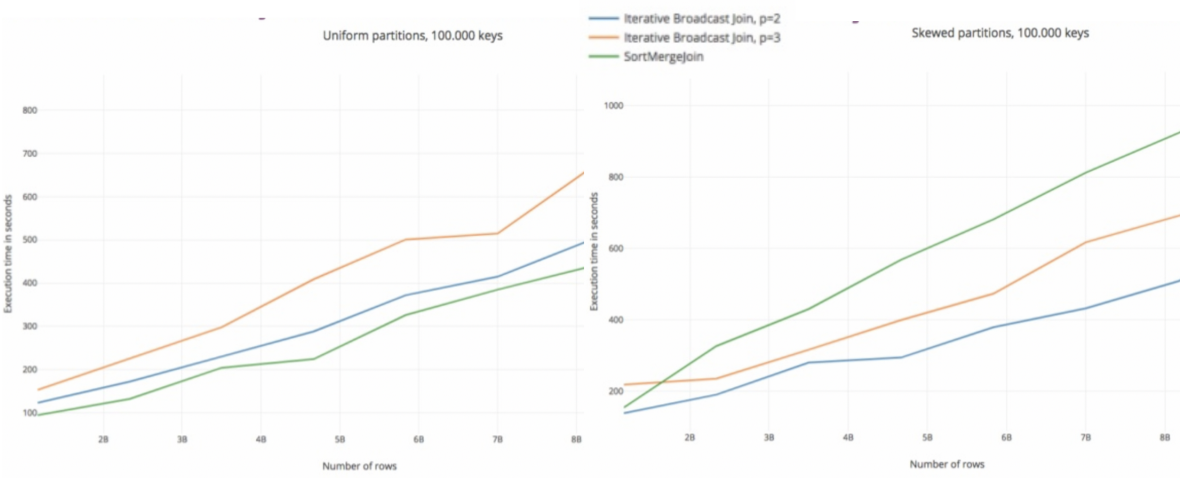


Figure 12: Results of Keevil and Driesprong's performance benchmarks, using chunk sizes of 2 and 3 compared to the original `MergeSortJoin` algorithm. [11]

2.1.2 Partial Broadcast Hash Join

Warren and Karau propose another alternative to the Broadcast Hash Join to manage skewed partition keys:

1. Identify which keys are most skewed by using Spark's `countByKeyApprox()` function, which calls Spark's internal `mapAsSerializableJavaMap()` function to counts by key an RDD.

2. Filter the smaller RDD to be joined so that it contains only these skewed keys, and collect the result into a `HashMap`.

3. Call `broadcast()` to send the smaller RDD to each of the partitions, and perform a manual hash inner join on each worker. At this point, you'll have a smaller joined table with only the most heavily skewed keys.

4. Filter the larger original RDD so that it contains only keys that are not heavily skewed (since these have already been joined).

5. Perform your standard sort merge join on this trimmed down table.
6. Union (stack) the two joined tables (one containing the most skewed keys, and the other containing the remaining keys) together.

The significant advantage this algorithm holds over Keevil and Driesprong's algorithm is that it avoids any intermediate collect and writes to disk, but this comes at the cost of increased complexity and significantly more transformations (two filter operations and a union). In addition, there have been no benchmarks or performance testing data to indicate that this algorithm actually does perform better than the traditional `SortMergeJoin`.

2.1.3 Proposal: Pseudo-Key Bucketing Algorithm

There are clear advantages and disadvantages to both Karau and Warren's and Keevil and Driesprong's solutions, and an ideal answer would be to take the best from both worlds:

	Iterative Broadcast	**Partial Manual Broadcast**
Strengths	• Avoids **any** costly shuffle operations • Allows large RDDs to fit into memory. • Has been proven to provide performance gains over traditional sort-merge-join methods given large dataset sizes.	• Avoids **some** costly shuffle operations • Avoids an intermediate write out to disk (unlike Iterative Broadcast) • Greatly improves the chances that a particular RDD will fit into memory (many rows are filtered out prior to broadcast and shuffle steps).
Weaknesses	• Requires an expensive "collect" and write out action after each iteration • Will only provide incremental performance benefits with a large dataset threshold, and may also require a sizable driver memory to collect each iteration's results into memory on local machine. • Requires tuning of additional hyperparameters, such as number of iterations	• Requires manually deciding which keys meet criteria for "skewed" • Involves significantly more steps (conversion to HashMap, filtering, broadcast, shuffle, refilter, union)

We propose the following algorithm, implemented over a small toy dataset where a join between a larger table (8 rows) of user name, state of residence, and weight, with a smaller reference table of United States state meta-data (like state capital). In the dataset described, **California** is the skewed key, occupying 50% of all data points:

1. Like Warren and Karau, utilize Spark's `countByKeyApprox()` to obtain a sampled distribution of the keys within to-be-joined table.
2. Using the **Kullback-Leibler (KL) Divergence**, calculate to what extent this distribution diverges from a discrete uniform distribution. Iteratively, we can compute which keys contribute the greatest "divergence" from a uniform distribution, and thus are the most skewed. This allows us to automate the process of identifying skewed keys.

3. Similar to Keevil and Driesprong's `pass` chunking indicator column, add an additional column to the table that serves as an indexer. We will use this `indexer column` as part of the input to hash the skewed key of each row into smaller partitions:

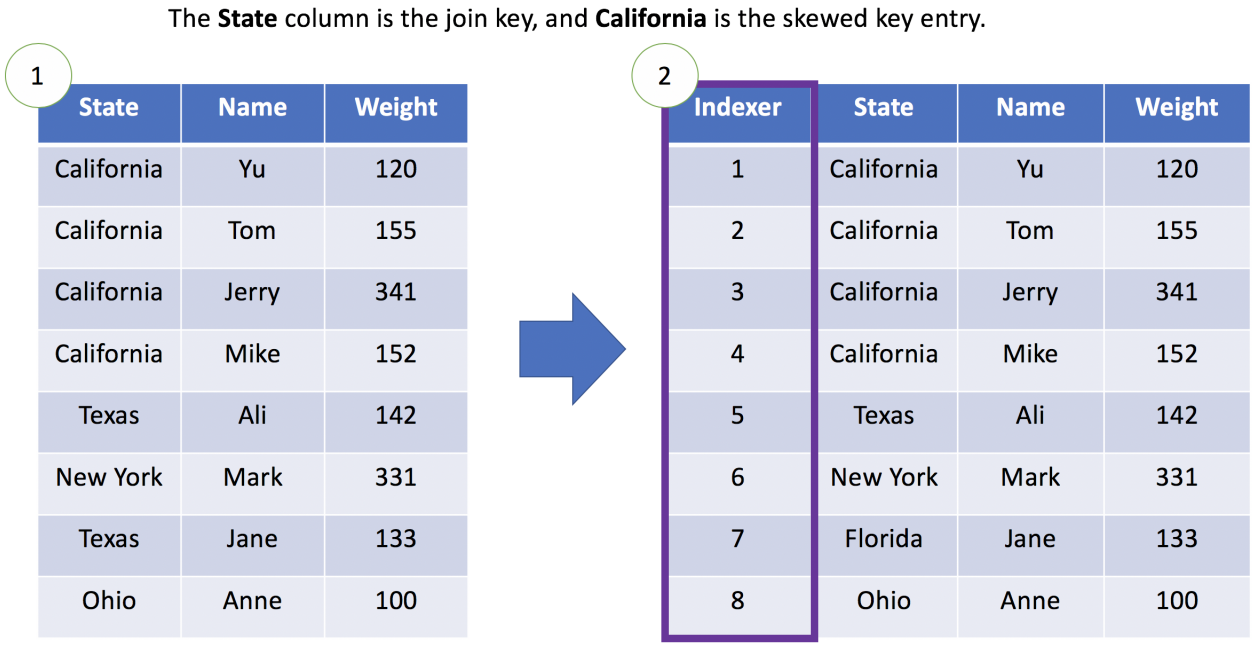


Figure 13

4. Concatenate the indexer column and original key column if the key in a given row matches the skewed key. Else, output the original key.

5. Map again over the newly concatenated column. If the concatenated column value starts with a valid indexer key, then apply a hashing function to uniformly distribute that row (which belongs to a skewed key) into B buckets, where B is a hyperparameter chosen by the user or autoconfigured. As a rule of them, we may select B such that each bucket of the skewed key will contain as many rows as the median key distribution (which we found in the first step using `countByKeyApprox()`. Selecting an optimal B value will allow for the greatest amount of parallelization in later steps.

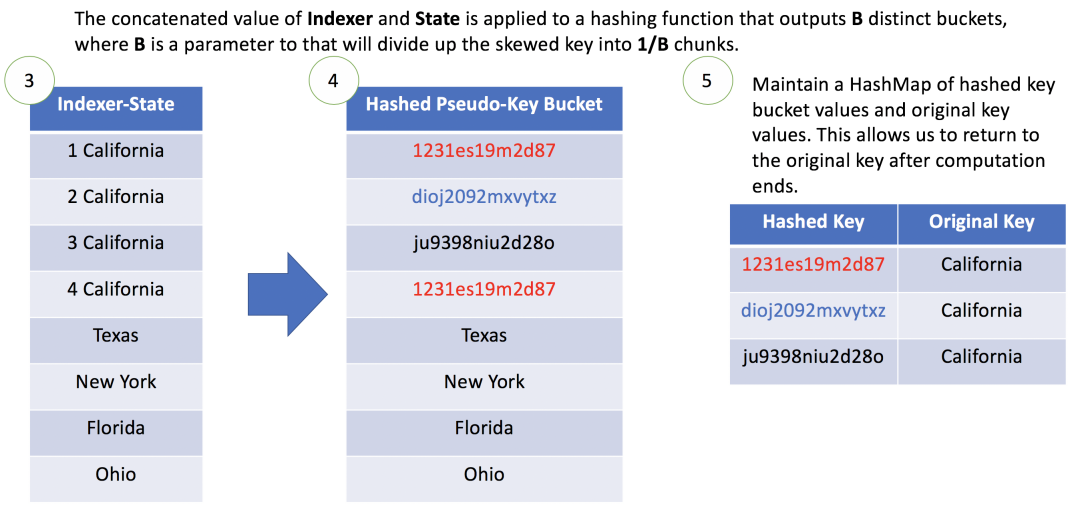


Figure 14

6. During the hashing process, we maintain a small `HashMap` of pseudo-key bucket values and the original key value. Since hashing is essentially a one-way operation, we'll need this data object to allow us to map from the newly created pseudo-key values to the original keys after the join completes.

7. Notice now, our keys are much more uniformly distributed. Initially, the California join key consumed 50% of all rows. After the pseudo-key bucketing, no key comprises more than 25% of the entire dataset. We have split our skewed key, **California**, into 3 smaller manageable chunks. The traditional Spark `SortMergeJoin` can now proceed, and should be able to take advantage of the new (and temporary) uniformity to parallelize computation across workers.

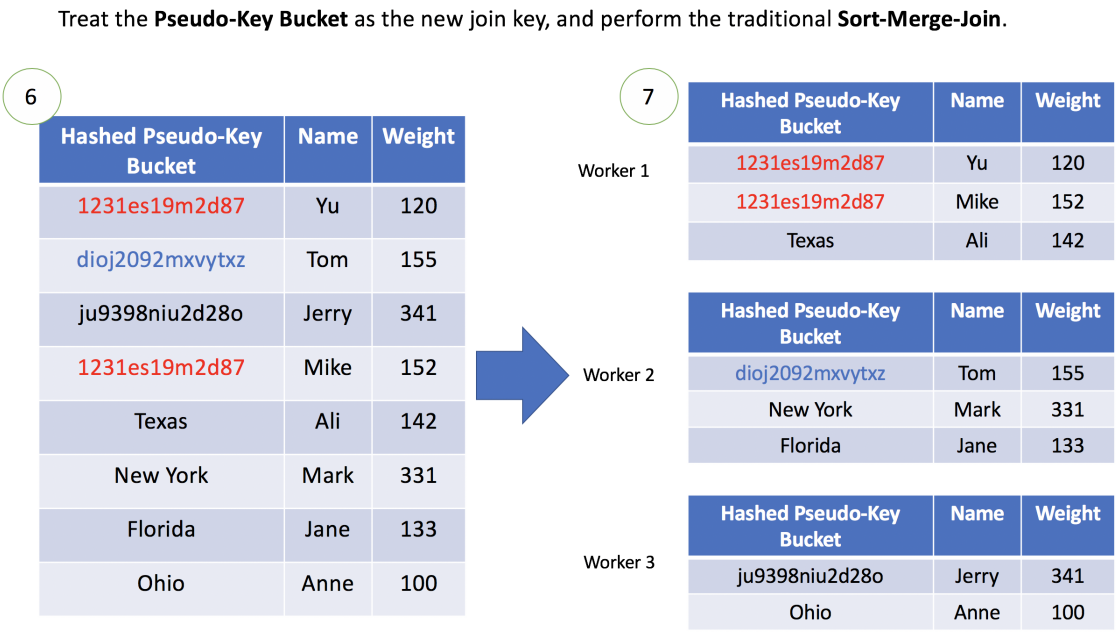


Figure 15

8. Finally, broadcast our small HashMap of pseudo-key bucket values and original key values to each worker.

This data object should be relatively small ($B * S$), where B is the number of chunks per skewed key, and S is the number of skewed keys.

9. Map the pseudo-key bucket column back to the original key values (in this case, California).

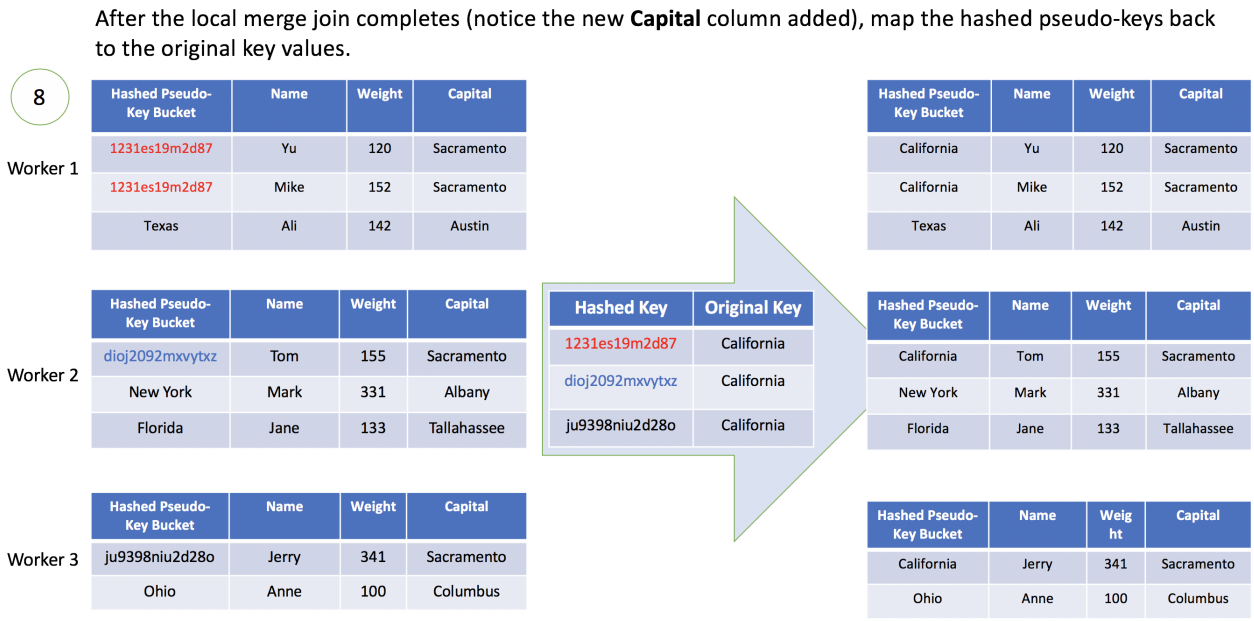


Figure 16

Once this process is complete, we will have successfully performed a relatively parallel join across a heavily skewed initial join key distribution. Granted, the above procedure has only been illustrated through pseudo-code, but we believe the **Pseudo-Key Bucketing** algorithm to be the optimal tradeoff between the strengths and limitations of the aforementioned Partial Manual Broadcast Join and the Iterative Broadcast. To its credit, it

- allows for **virtually all** datasets to fit into memory. Similar to the iteration hyperparameter of Keevil and Driesprong's iterative Broadcast, we can increase the number of buckets B to reduce the number of rows corresponding to the skewed join key that each worker's partition must hold. This comes at a tradeoff of an increased size in the `HashMap`, but this data object scales extremely well- remember that B (buckets per skewed key) and S (number of skewed keys to bucket) are likely to orders of magnitude smaller than N, the size of the dataset.
- parallelizes a formerly skewed workload- all workers are essentially guaranteed to be assigned a fair amount of work, and the likelihood of a state where many idle workers wait while the lagging worker finishes an extremely expensive task is minimized.
- like Karau and Warren's algorithm, it skips any inefficient, "hacky" write to disk intermediate step to force Spark to artificially create stage barriers and keep RDDs within memory (as happens with the Broadcast Hash Join).

With that said, the algorithm clearly comes at a tradeoff, and we anticipate the following limitations:

- It likely performs more overall work than either Keevil/Driesprong and Warren/Karau's algorithms. It performs an initial `map()` call to create and concatenate the indexer column with the key, another `map()` function to create the hashed bucket function, and then a final mapping back to the original values. This is discounting the (relatively trivial construction of the `HashMap` and broadcast costs to each worker node).

- It performs an expensive shuffle across workers within the cluster, although this shuffle is more or less optimized by being significantly more uniformly distributed than a raw skewed key dataset.

In all three algorithms highlighted, workers attempt to share work equally, but incur an **equality tax**, manifested in different forms (intermediate writes to disk, extra filtering and union transforms, shuffles across the network, ...). In both the Iterative Broadcast Join and the Partial Manual Broadcast Join, work is brought (broadcasted) to the original partitions, with the belief that avoiding a costly shuffle will result in incremental performance gains. In the Pseudo-Key Bucketing algorithm, work is assigned equally to each worker, with the belief that even after paying the cost for shuffling, the optimized and balanced local join tasks will overshadow this "equality tax".

We look forward to implementing this pseudo code and benchmarking its performance against existing solutions and Spark's native `SortMergeJoin`.

Appendix A Other Spark Memory Optimizations

A.0.1 Pre-Join Filters

Typically, not all keys are used in both datasets (for example, if we are joining California residents' data (let's say it is in a table called `california` to a much larger table called `world` containing other information for each person in the world, we do not need to shuffle and sort all keys in `world`. Holden Karau and Rachel Warren provide an example of two extremely similar commands below (both attempt to create a table that contains a user's top score alongside their address information by joining an RDD called `scores` with one called `addresses`)[**?**]:

```
/* this method forces all score values to be shuffled across the cluster and reassigned a new
    partition according to its key */
addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, Option[String]))]= {
  val joinedRDD = scoreRDD.join(addressRDD) /* join scores to address table by user ID (the Long
      data type)*/
  joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y ) /* get the max for each user

/* this method prefilters the scores prior to joining */
addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {
  val bestScoreData = scoreRDD.reduceByKey((x, y) => if(x > y) x else y)
  bestScoreData.join(addressRDD)
```

The cost saved during the shuffle stage is proportional to the average number of scores per user in the `scores` table. Thus, if each user had on average, 10 recorded scores, this strategy would reduce the amount of data shuffled from the `scores` table by a factor of 10.

A.0.2 Rule-Based Costing Algorithms

Spark utilizes a rule-based approach to calculate reordering for many optimizations, including when and where to reorder joins, push down filters, etc. The primary benefit of this is simplicity, but leaves a variety of optimizations on the table, and represents a classic tradeoff question between exploration and exploitation: *How much time should we spend searching for the optimal query plan, versus simply executing it?* For instance, the `org.apache.spark.sql.catalyst.optimizer.CostBasedJoinReOrder` class uses a simple weighting between the relative differences in cardinality (number of columns, `this.planCost.card`) and size (number of rows, `this.planCost.size`) of the old (`other`) and new (`this`):

```
  def betterThan(other: JoinPlan, conf: SQLConf): Boolean = {
    if (other.planCost.card == 0 || other.planCost.size == 0) {
      false
    } else {
      val relativeRows = BigDecimal(this.planCost.card) / BigDecimal(other.planCost.card)
      val relativeSize = BigDecimal(this.planCost.size) / BigDecimal(other.planCost.size)
      relativeRows * conf.joinReorderCardWeight +
```

```
        relativeSize * (1 - conf.joinReorderCardWeight) < 1
    }
  }
}
```

In the rule above, the `conf.joinReorderCardWeight` is a value between 0 and 1, and represents how much weighing to put on the relative difference in number of columns. It defaults to `0.7`. The documentation for `CostBasedJoinReOrder` also notes that the algorithm is greedy, evaluating only its local search space of `m` items at a particular level.

```
/**
 * When building m-way joins, we only keep the best plan (with the lowest cost) for the same set
 * of m items. E.g., for 3-way joins, we keep only the best plan for items {A, B, C} among
 * plans (A Join B) Join C, (A Join C) Join B, and (B Join C) Join A.
 */
```

There are clearly more sophisticated algorithms for selecting join order, including those guaranteed to arrive at a global minimum cost. Moreover, a variety of dimensions that affect join performance can be factored into the calculation for `betterThan`:

- **column data types**: certain nested data structures, like JSON-like dictionaries should be targeted for elimination through joins as early on in the processing pipeline as possible, since they involve serialization/deserialization and consume significantly more space than primitives.
- **distribution of join keys**: if keys are extremely skewed in terms of distribution, that particular join may be significantly more costly than it would appear simply looking at cardinality and row sizes

However, there is likely two primary reasons why these more advanced methods have yet to be implemented: **1.** the search space would likely increase quadratically, and in many cases it may not make sense to send so much optimizing the query plan versus simply executing a slightly less efficient plan earlier (this is akin to a database query engine choosing not to utilize an index and simply perform a sequential scan since the cost of index lookup and planning may outweigh the cost of a sequential scan for small row sizes). **2.** The issue of unknown key distribution likely outweighs and completely subsumes any performance gain derived from reordered joins.

Appendix B Sort Merge Join

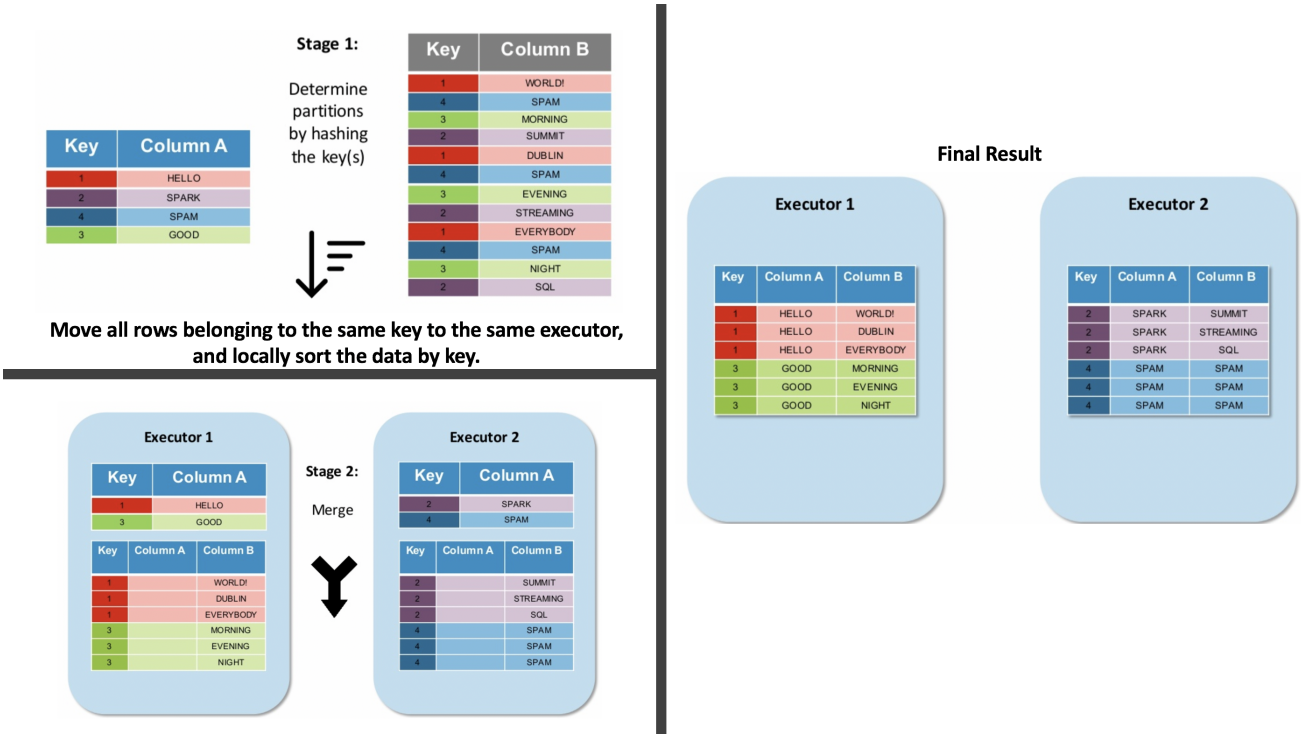


Figure 17: A typical sort merge join between two tables in Spark. A hashing function is applied to uniformly distribute the data by key across available executors. The data is then sorted locally so that data belonging to the same key is collocated next to each other using in-memory columnar storage. Diagrams taken from Keevil and Driespong talk at Spark Summit EU 2017.[11]

Notice, that the join features 4 distinct keys, with 3 rows per key. This allows the two executors to equally partition work, and represents the ideal state.

Appendix C Broadcast Hash Join

Keevil and Driespong's solution builds upon an established join algorithm called the Broadcast Hash Join, which is a useful technique to avoid shuffling data across partitions (and all the corresponding memory issues associated with partitioning skewed data by their join key).

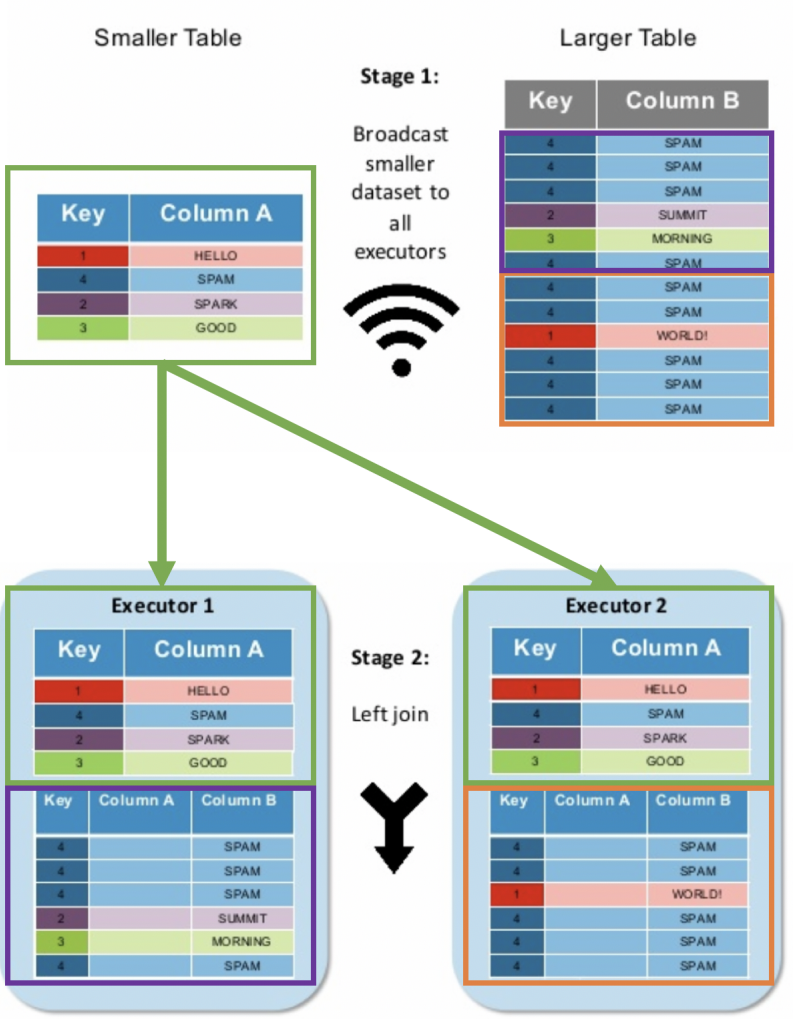


Figure 18: The broadcast hash join that forms the basis of Keevil and Driespong's iterative broadcast solution. Using this technique, the executors hold data that is not partitioned based upon a join key. Instead, the smaller table is broadcast to all executors, and then each partition performs the merge locally.

Appendix D Iterative Broadcast Join (Keevil and Driespong)

Keevil and Driespong's code for implementing their Iterative Broadcast Join is available in a Github repository.

```
package com.godatadriven.join

import com.godatadriven.SparkUtil
import com.godatadriven.common.Config
import org.apache.spark.sql.functions._
import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.annotation.tailrec

object IterativeBroadcastJoin extends JoinStrategy {

  @tailrec
  private def iterativeBroadcastJoin(spark: SparkSession,
                                     result: DataFrame,
```

```
                                broadcast: DataFrame,
                                iteration: Int = 0): DataFrame =
    if (iteration < Config.numberOfBroadcastPasses) {
      val tableName = s"tmp_broadcast_table_itr_$iteration.parquet"

      val out = result.join(
        broadcast.filter(col("pass") === lit(iteration)),
        Seq("key"),
        "left_outer"
      ).select(
        result("key"),

        // Join in the label
        coalesce(
          result("label"),
          broadcast("label")
        ).as("label")
      )

      SparkUtil.dfWrite(out, tableName)

      iterativeBroadcastJoin(
        spark,
        SparkUtil.dfRead(spark, tableName),
        broadcast,
        iteration + 1
      )
    } else result

  override def join(spark: SparkSession,
                    dfLarge: DataFrame,
                    dfMedium: DataFrame): DataFrame = {
    broadcast(dfMedium)
    iterativeBroadcastJoin(
      spark,
      dfLarge
        .select("key")
        .withColumn("label", lit(null)),
      dfMedium
    )
  }
}
```

References

[1] Jeff Smith and Rohan Aletty. Spark Summit 2017. *Spark as the Gateway Drug to Typed Functional Programming*. February 14th, 2017.

[2] Steve Chapin. Week 6, Principles of Operating Systems. *Early/Late Binding: Eager/Lazy Evaluation*. Syracuse University College of Engineering.

[3] StackOverflow, user Sathish. Accepted answer to *How DAG works under the covers in RDD?*. Accessed November 30th, 2018. Link.

[4] Jacek Laskowski. Partitions and Partitioning. *Mastering Apache Spark*. Accessed November 30th, 2018. Link.

[5] Aaron Davidson. A Deeper Understanding of Spark Internals. *Talk delivered at Spark Summit 2014, San Francisco, CA*. Accessed November 30th, 2018. Slides.

[6] Daoyuan Wang and Jie Juang. Tuning Java Garbage Collection for Apache Spark Applications. Accessed November 30th, 2018. Link.

[7] Ankunda R. Kiremire. The Application of the Pareto Principle in Software Engineering. Accessed November 30th, 2018. Link.

[8] *Human Development Report 1992*, Chapter 3. Accessed November 30th, 2018. Link.

[9] Paula Rooney. Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features. Accessed November 30th, 2018. Link.

[10] Tim Berglund. Distributed Systems in One Lesson. *Devoxx Poland (Krakow) June 21-23, 2017*. Accessed November 30th, 2018. Link.

[11] Rob Keevil and Fokko Driesprong. Working with Skewed Data: The Iterative Broadcast. Talk delivered at *Spark Summit EU 2017*. Accessed November 30th, 2018. Link.

[12] Vida Ha. Optimizing Apache Spark SQL Joins. Talk delivered at *Spark Summit 2017*. Accessed November 30th, 2018. Link.

[13] Wei Li, Dengfeng Gao, and Richard T. Snodgrass. *Skew Handling Techniques in Sort-Merge Join*. University of Arizona and Oracle Corporation. Accessed November 30th, 2018. Link.

[14] Holden Karau and Rachel Warren. Joins (SQL and Core). *High Performance Spark*. Accessed November 30th, 2018. Link.

[15] Ram Ghadiyaram. Answer to *Apache Spark: map vs mapPartitions?*. Accessed November 30th, 2018. Link.

[16] Rob Keevil and Fokko Driesprong. Iterative Broadcast Join Github repository. Link.

[17] Jim Mohr. Introduction to Operating Systems. Link.