Editor: Jimmy Lin, jimmylin@uwaterloo.ca

In Defense of MapReduce

Jimmy Lin • *University of Waterloo*

Friends, developers, researchers, lend me your ears! I come to praise MapReduce, not to bury it!

Google Fellow Luiz Barroso once famously said, "The datacenter is now the computer." David Patterson[1] then wondered: What's the instruction set architecture for the datacenter computer? This was circa 2008, and of course by then we already knew the first answer: for MapReduce,[2] the two instructions are MAP and REDUCE.

MapReduce represents a specific instance of a general class of data-parallel dataflow languages, in which computations are conceptualized as directed graphs, where vertices represent operations on records that flow along the directed edges. From this perspective, MAP and REDUCE are the two operators that MapReduce provides, which define particular configurations of the edges that flow into and out of vertices and specify the computations that occur at the vertices themselves.

Such a dataflow computation model dates back to the 1970s,[3,4] but there's one key difference: in today's conception of dataflow languages, the focus is on data-parallelism. With perhaps the exception of sophisticated machine learning algorithms, the dataflow graphs today themselves are relatively simple. The challenges primarily lie in the requirement to process gazillions of records, thus necessitating distributed processing across warehouse-scale clusters.

Dryad is usually credited with the development of the first general dataflow model for distributed data-parallel computations.[5] It provides a rich vocabulary of operators that can be composed to form complex dataflow graphs and an execution engine for managing the specified computations. Over time, we've witnessed the development of alternative data-parallel dataflow languages that primarily differ in the vocabulary of operators that they provide to the developer. Examples include Cascading (www.cascading.org), Google's FlumeJava,[6] Apache Flink (https://flink.apache.org), and of course, Apache Spark (https://spark.apache.org), which is perhaps the most widely touted general-purpose replacement for MapReduce.

In this column, I present a critical analysis of the dataflow operators provided by MapReduce and Spark. To be precise, I'll be specifically referring to the Hadoop implementation of MapReduce, which is slightly different from Google's original. My message is this: In the fashion-driven world of big data where there's a perpetual rush toward new and shiny objects along with a tendency to pooh-pooh everything that has come before, let's not throw the MapReduce baby out with the bathwater!

There are aspects of the Hadoop MapReduce API that provide a well-conceived balance between flexibility and expressiveness, even if there are serious shortcomings with the overall implementation. In particular, comparisons with Spark are instructive for highlighting the distinction between logical and physical operators, and point to a gaping hole where MapReduce is incomplete. If we recognize MAP and REDUCE as the physical operators that they really are, then comparing MapReduce to Spark is actually like comparing apples to oranges (quite contrary to popular portrayals by blogs, the tech press, and other superficial discussions).

Before I start getting hate mail from Spark fanboys, let me be perfectly clear: I really like Spark. On the whole, it represents a far superior implementation of MapReduce. Resilient Distributed Datasets (RDDs) and lazy transformations support pipelining, plan rewrites, and other optimizations difficult to implement in Hadoop. Caching of RDDs accelerates iterative algorithms while lineage information provides robustness. Having a read-eval-print loop (REPL) is a godsend. Language bindings beyond the Java Virtual Machine democratize data processing tools to the large community of Python and R users. Overall, Spark most definitely deserves the mantle as the successor to MapReduce.

Published by the IEEE Computer Society 1089-7801/17/$33.00 © 2017 IEEE

Big Data Bites

"Big Data Bites" is a regular department in *IEEE Internet Computing* that aims to deliver thought-provoking and potentially controversial ideas about all aspects of big data. Interested in contributing? Drop me a line!

—*Jimmy Lin*

But I'm going to complain about the design of some of Spark's dataflow operators anyway. The discussion in the rest of this column is fairly low level and borders on "inside baseball" minutiae, but by design (because that's part of my critique). However, I'll freely admit that this is mostly an academic exercise — although I do think the exercise is instructive in helping us better understand the design of big data analytics platforms in general.

At the end of this article, I'll come back and discuss why none of this particularly matters, and why questions about low-level operator design are inconsequential as big data processing becomes increasingly focused on higher-level abstractions for vertically-informed data manipulation.

The Mappers

Let's start with mappers in MapReduce. In Hadoop, a mapper is instantiated for every partition of the input dataset — more precisely, each input split, which aligns with data blocks on the Hadoop Distributed File System (HDFS) in the canonical case. The mapper lifecycle begins with a `setup` method and ends with a `cleanup` method; in between the `map` method is called for each key-value pair in the input collection. Specifically, the framework initializes a `RecordReader`, iteratively calls the reader to materialize the next key-value pair, and then calls the `map` method in the mapper.

In Spark, there are a few comparable "map" transformations: `filter`, `map`, `flatMap`, and `mapPartitions`. Why do we need all of them? The `map` transformation takes `f: T ⇒ U` to transform inputs of type `T` to outputs of type `U`. Importantly, `map` in Spark produces exactly one output per input and thus is less flexible than the map method in MapReduce, which can generate zero, one, or more intermediate key-value pairs. This is why Spark additionally needs `filter` (to *not* generate any intermediate records) and `flatMap` (to generate a list of intermediate records, which the framework then flattens). Finally, `mapPartitions` is needed to provide the equivalent of what `setup` and `cleanup` do in MapReduce: `mapPartitions` takes `f: Iterator[T] ⇒ Iterator[U]`, which allows developers to sneak in setup and cleanup.

From the perspective of a functional purist, I can argue that the MapReduce abstraction leaks in two ways. First, the mappers encapsulate per-record processing, and the fact that the input collection is divided into partitions: well, that's an implementation detail. Having `setup` and `cleanup`, in effect, hard codes the existence of some partitioning scheme and a particular physical organization. Second, having control over the mapper lifecycle allows the developer to retain state across map calls and engage in monkey business that breaks the functional abstraction.

With respect to the first point: data processing languages must operate in the real world, and in the real world programmers do need to manage their object lifecycles. The most common use for the `setup` method is to acquire an external resource (for example, a database connection) or load in side data (for example, a dictionary). The `cleanup` method is used to release these resources after all the records have been processed. Because it makes no sense to perform these heavyweight operations on a per-record basis, the `filter`, `map`, and `flatMap` transformations in Spark can't handle this common design pattern.

It appears that the developers of Spark realized this fact somewhat later, as `mapPartitions` wasn't in the original Spark API;[7] see Apache JIRA ticket SPARK-341. Thus, both MapReduce and Spark are adulterated in having abstractions that aren't functionally pure. In Spark, there's actually also something called `mapPartitionsWithIndex`, which provides an integer value representing the index of the partition, which I react with a #facepalm (If you don't get the meme, search the web. I especially love Captain Piccard doing it). You don't need it in MapReduce because in the `setup` you have access to the `Context` object, which gives you a lot of reflection-like information about the state of the mapper and the data it's running on.

With respect to the second point — the ability in MapReduce to retain state across map calls and engage in monkey business — this actually allows you to do some neat optimizations that substantially increase performance via more efficient local aggregations, for example, what I've called the "in-mapper combining" pattern.[8,9] I suppose someone might argue that such techniques are hacky and shouldn't be allowed, but then, the same complaint surely would apply to Spark, because it has `mapPartitions`. Whatever sneaky monkey business you can do in MapReduce, you can do in Spark also.

Here's another way to think about it: the mapper in MapReduce is actually a physical operator, while transformations in Spark aim to be logical operators. The conventional understanding is that a physical operator specifies a particular implementation, whereas a logical operator expresses the computation at a more abstract level (for example, relational operators in the case of SQL).

As discussed previously, mappers in MapReduce make explicit the sequence of computations (method calls) that occur while processing a

collection of key-value pairs. In fact, in Hadoop you call the `Context.write()` method to actually emit intermediate key-value pairs. The map transformations `filter`, `map`, `flatMap` in Spark, on the other hand, don't tie the framework to any particular implementation. There's no reason, for example, why each input record can't be processed by a separate thread, which might make sense if Spark were implemented in Erlang due to its support for lightweight threads.

But here's where the rubber of clean abstraction design meets the road of real-world constraints: `mapPartitions` and `mapPartitionsWithIndex` in Spark hard code aspects of physical execution, which ruin the elegance of Spark transformations. The ability for developers to manipulate physical operators in MapReduce affords a high degree of flexibility (as is generally the case when physical operators are accessible); this control is ceded for cleaner logical operators in Spark, but unfortunately Spark isn't able to completely deliver on elegance.

The Reducers

Now let's turn our attention to reducers. The reducers in MapReduce have a signature of `g: (T, Iterator[U]) ⇒ Seq[(R, S)]`. That is, the reducer `g` takes an object of type `T` (the intermediate key) and an iterator over values of type `U` (the values associated with that intermediate key), and returns any number of output key-value pairs (of any arbitrary type). In MapReduce there are also combiners, with the same signature. Combiners perform per-key aggregation on the output of the mappers, prior to the network shuffle. In reality, however, Hadoop also sneaks in combiner execution on the reduce end, post-shuffle. And oh, one more detail: the `T`'s are sorted (more on this later).

Similar to the mappers, developers can manage the reducer and combiner lifecycles via the `setup` and `cleanup` API hooks. To accomplish the equivalent in Spark, you'd have to do something like `groupByKey` followed by `mapPartitions`. At that point, you're basically just writing MapReduce in Spark, which is fine — but just don't be hatin' MapReduce. (Not to mention that a straight-up `groupByKey` isn't particularly efficient because it doesn't do map-side aggregation; of course, you can add in map-side aggregation, but then, my original point remains — it's basically back to MapReduce.)

Okay, so in MapReduce we have reducers and combiners, and that's it. Oh, there are also partitioners, which simply divide up the intermediate key space, but you can't get away without having something like that (and Spark has partitioners also). In Spark, there's a number of reduce-like operations (leaving aside joins and cogrouping for now): `groupByKey`, `reduceByKey`, and `aggregateByKey`. What do these all do?

The `reduceByKey` transformation takes `g: V × V ⇒ V`. In other words, `V` forms a commutative monoid with `g` as its associative binary operation (more precisely, a commutative semigroup, because left unspecified is the identity element). Implicit in the semantics of `reduceByKey` is that `g` must be associative and commutative, because otherwise the execution simply wouldn't be correct: Spark's documentation was previously muddled (see SPARK-12844) but it has since been fixed. In contrast, a system like Summingbird renders the algebraic properties of the types explicit.[10]

Leaving issues with precise execution semantics aside, `reduceByKey` is actually quite restrictive, because the input and output types can't change. This is both a plus and a minus. The positive is that Spark can take the function `g` that goes into `reduceByKey` and move it over to the map side, that is, before the network shuffle, in a completely transparent manner. So, there's no need to explicitly write combiners — the framework optimizes for you. The downside is that all the values have to be type `V` and the function must also operate on that type.

What if I want to be a bit more flexible on the output type? This is where `aggregateByKey` comes in: the transformation takes `f: U × V ⇒ U`, `g: U × U ⇒ U`; in other words, `f` allows you to convert from type `V` (the input type) to type `U` (the output type) while performing intermediate aggregation, and `g` specifies how you combine values of type `U` together. With this setup, `f` automatically can be pushed over to the map side for efficient intermediate aggregation.

What if you still can't get all the types to work out correctly? Well, then you're back to `groupByKey` followed by a map-like transformation — get the framework to group together all the values with the same intermediate key for you, and then apply whatever computation you want yourself. At that point, though, the framework can't perform any optimizations behind the scenes, and performance will suffer because of the shuffling of lots of intermediate data across the network.

One way to think about these reduce-like operations in Spark is in terms of the tradeoff between simplicity, flexibility, and performance. With `reduceByKey`, you get simplicity and performance, at the cost of flexibility. With `aggregeteByKey`, you gain a bit of flexibility at the cost of simplicity, but still get good performance. With `groupByKey`, you get simplicity and maximum flexibility, but at the cost of performance. Contrast this with MapReduce, where you only get reducers and combiners, but there isn't anything you fundamentally can't do because you have low-level control over the physical execution. It's not clear that the Spark panoply of reduce-like transformations is easier to understand or use.

What do we see if we compare reducers in MapReduce with reduce-like transformations in Spark through the lens of the logical/physical distinction discussed previously? The reduce in MapReduce most definitely

describes a physical operator: keys arrive at the reducer in sorted order, which all but prescribes how the shuffle group-by must be implemented. (In reality, it was closer to the other way around: the framework developers came up with an efficient sort-based shuffle grouping implementation, and then shrugged, "well, we might as well expose this in the API.")

In contrast, the Spark reduce-like transformations leave open the implementation, and indeed Spark can select between a hash-based and a sort-based shuffle scheme. In practice, however, hash-based shuffling suffers from scalability limitations beyond a certain point, and thus sort-based shuffling has been the default since Spark 1.2. So, while reduce-like transformations are nominally logical operators in Spark, they're severely constrained by the practicalities of execution performance. Once again, the rubber of clean design meets the road of implementation realities.

Other Spark Transformations

At this point, I've compared Spark's map-like transformations with mappers in MapReduce and Spark's reduce-like transformations with combiners and reducers in MapReduce, arguing that there's a certain elegance in the simplicity of MapReduce, particularly in providing developer access to physical operators, including the ability to explicitly manage object lifecycles. But Spark has many transformations beyond map-like and reduce-like transformations, and this is where Spark really shines.

In Spark, joins are first-class citizens expressed concisely at the logical level. Spark transformations let the developer explicitly reference the two different RDDs that are participating in the join (inner, left, right, or full outer). In contrast, getting MapReduce to do joins is a huge kludge. In essence, the developer must code up the actual physical join plan using only map and reduce, unless a SQL-on-Hadoop platform like Hive is used.

In a standard MapReduce reduce-side join, because we can only map over a single input, to join *R* and *S* we'd have to mash together *R* and *S* in the input specification, and then in the mapper figure out if we're dealing with a record from *R* or a record from *S* (for example, by examining the input path). The join key is emitted as the intermediate key with the record as the value; the framework brings all records with the same join key together in the reducers, where the join processing actually happens.

On the other hand, if we wanted to do a copartitioned sort–merge join, often called a map-side join in MapReduce parlance, the implementation is even uglier. Typically, we map over one of the collections (say, *R*) and inside the mapper read (directly from HDFS) records from the other collection (*S*). Yuck!

Here, Spark claims a mic-drop moment. A join is simply `r.join(s)`. That's it. The join is specified logically, and thus Spark is able to figure out the best physical join plan behind the scenes. Beyond joins, Spark's set-like operations (`union`, `intersect`, `distinct`, `cartesian`) also have no equal in MapReduce. It's doable in MapReduce, but ugly.

So What Do We Really Need?

If MapReduce specifies two physical operators — partitioned per-record processing (map) and partitioned sort-based shuffle grouping (reduce) — we might wonder what's missing in its repertoire?

The major glaring hole in the design of MapReduce is the lack of operators that process two distinct datasets — which is why joins are so painful to implement. In particular, I see the need for two operators: a shuffle cogrouping operator (let's call it coreduce) and a copartitioned, cogrouping operator (let's call it comap). Both operators would take two collections of key-value pairs, `R: (K, V1)` and `S: (K, V2)`, and guarantee that values with the same key from both collections are available for processing together, something like `process(K, Iterator<V1>, Iterator<V2>)`. The coreduce operator would provide a general implementation via shuffling, whereas comap would assume that the input collections are copartitioned. With coreduce and comap, we can efficiently implement joins as well as the set-like transformations that Spark provides.

Once again, the rubber of clean design meets the road of implementation realities.

Interestingly, with map, comap, reduce, and coreduce, we arrive at something that's pretty close to Spark's actual physical operators. If we take a look at Figure 4 in the original Spark paper,[7] the wide dependencies are essentially reduce and coreduce, and the narrow dependencies are essentially map and comap. Unfortunately, in Spark, you don't have access to these. Often, I wish I did.

Given this discussion, we arrive at another way to think about the relationship between MapReduce and Spark in the context of data-parallel dataflow languages: MapReduce provides two physical operators that specify exactly how to wire up a dataflow graph for execution. The two physical operators are impoverished, which makes wiring up dataflow graphs to accomplish certain tasks (for example, joins) rather painful.

Spark, on the other hand, provides a set of logical transformations on collections that afford different physical

execution plans — in other words, it provides a higher level of abstraction. My nitpicky complaint, summarizing the previous discussion, is that these logical transformations are in some cases inelegant and adulterated with abstraction-breaking assumptions about physical execution.

Thus, comparing MapReduce to Spark is a bit like comparing apples to oranges. Better points of comparison are actually Pig[11] and DryadLINQ,[12] both of which provide high-level transformations for manipulating collections of records. However, a thorough analysis is perhaps better left for another column.

As I intimated at the beginning of this column, although this discussion might be interesting from an academic or historic perspective, none of these issues particularly matter. In the evolution of big data processing technologies, the broader trend is toward increasing levels of abstraction that isolate the data scientists from the particulars of execution. In the Spark ecosystem, for example, there's a growing emphasis on DataFrames[13] as the default abstraction for manipulating datasets, as opposed to raw transformations on RDDs (which basically makes all discussion in this column irrelevant).

Today, these increasing levels of abstraction naturally segment into different verticals. As my colleague Ihab Ilyas opines, "data analytics will go vertical" — tackling specific market segments such as financial, pharmaceutical, healthcare, energy, the Internet of Things, and so on. I hear similar musings from investors: general infrastructure plays are becoming increasingly difficult in today's already crowded space, and going vertical is one avenue for differentiation.

At a high level, this column is a navel-gazing critique about instruction-set architectures for the datacenter computer. If the reduced versus complex instruction-set computing (RISC versus CISC) wars of the 1980s are a guide, ultimately, it doesn't matter,[14] and people only care about applications in the end. Well, not quite: people who write compilers still care — and if we follow this analogy, analytics infrastructure builders are the compiler writers of the 21st century for datacenter computers. We'll quibble about how high-level data science directives (such as "Train this machine-learning model!") translate into physical execution on warehouse-scale clusters; obviously important, but mostly relegated to a highly-specialized and esoteric craft.

Acknowledgments

I thank Semih Salihoglu, Reynold Xin, and Matei Zaharia for comments on previous drafts of this column.

References

1. D.A. Patterson, "The Data Center Is the Computer," *Comm. ACM*, vol. 52, no. 1, 2008, p. 105.
2. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. 6th Usenix Symp. Operating System Design and Implementation*, 2004, pp. 137–150.
3. J. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symp.*, B. Robinet, ed., LNCS 19, 1974, pp. 362–376.
4. J. Dennis, "Data Flow Supercomputers," *Computer*, vol. 13, no. 11, 1980, pp. 48-56.
5. M. Isard et al., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *Proc. ACM Sigops/EuroSys European Conf. Computer Systems*, 2007, pp. 59-72.
6. C. Chambers et al., "FlumeJava: Easy, Efficient Data-Parallel Pipelines," *Proc. ACM Sigplan Conf. Programming Language Design and Implementation*, 2010, pp. 363–375.
7. M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," *Proc. 9th Usenix Symp. Networked Systems Design and Implementation*, 2012; www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.
8. J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, Morgan & Claypool, 2010.
9. J. Lin and Michael Schatz, "Design Patterns for Efficient Graph Algorithms in MapReduce," *Proc. 8th Workshop Mining and Learning with Graphs*, 2010, pp. 78-85.
10. O. Boykin et al., "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," *Proc. Very Large Data Bases*, vol. 7, no. 13, 2014, pp. 1441–1451.
11. C. Olston et al., "Pig Latin: A Not-So-Foreign Language for Data Processing," *Proc. ACM Sigmod Int'l Conf. Management of Data*, 2008, pp. 1099–1110.
12. Y. Yu et al., "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," *Proc. 8th Usenix Symp. Operating System Design and Implementation*, 2008; www.usenix.org/legacy/event/osdi08/tech/full_papers/yu_y/yu_y.pdf.
13. M. Armbrust et al., "Scaling Spark in the Real World: Performance and Usability," *Proc. Very Large Data Bases*, vol. 8, no. 12, 2015, pp. 1840–1843.
14. E. Blem et al., "ISA Wars: Understanding the Relevance of ISA Being RISC or CISC to Performance, Power, and Energy on Modern Architectures," *ACM Trans. Computer Systems*, vol. 33, no. 1, 2015, article no. 3.

Jimmy Lin holds the David R. Cheriton Chair in the David R. Cheriton School of Computer Science at the University of Waterloo. His research lies at the intersection of information retrieval and natural language processing, with a particular focus on big data and large-scale distributed infrastructure for text processing. Lin has a PhD in electrical engineering and computer science from MIT. Contact him at jimmylin@uwaterloo.ca.