MAPREDUCE IS GOOD ENOUGH?

If All You Have is a Hammer, Throw Away Everything That's Not a Nail!

Jimmy Lin
The iSchool, University of Maryland
College Park, Maryland

Abstract

Hadoop is currently the large-scale data analysis "hammer" of choice, but there exist classes of algorithms that aren't "nails" in the sense that they are not particularly amenable to the MapReduce programming model. To address this, researchers have proposed MapReduce extensions or alternative programming models in which these algorithms can be elegantly expressed. This article espouses a very different position: that MapReduce is "good enough," and that instead of trying to invent screwdrivers, we should simply get rid of everything that's not a nail. To be more specific, much discussion in the literature surrounds the fact that iterative algorithms are a poor fit for MapReduce. The simple solution is to find alternative, noniterative algorithms that solve the same problem. This article captures my personal experiences as an academic researcher as well as a software engineer in a "real-world" production analytics environment. From this combined perspective, I reflect on the current state and future of "big data" research.

Introduction

MAPREDUCE[17] HAS BECOME A UBIQUITOUS FRAMEWORK FOR large-scale data processing. The Hadoop open-source implementation enjoys widespread adoption in organizations ranging from two-person startups to Fortune 500 companies. It lies at the core of an emerging stack for data analytics, with support from industry heavyweights such as IBM, Microsoft, and Oracle. Among the advantages of MapReduce are: the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance.

MapReduce, of course, is not a silver bullet, and there has been much work probing its limitations, both from a theoretical[2,30] and empirical perspective, by exploring classes of algorithms that cannot be efficiently implemented with it.[5,12,23,56] Many of these empirical studies take the following form: They present a class of algorithms for which the naive Hadoop solution performs poorly, expose it as a fundamental limitation of the MapReduce programming model,[a] and propose an extension or alternative that addresses the limitation. The algorithms are expressed in this new framework, and, of course, experiments show substantial (an order of magnitude!) performance improvements over Hadoop.

This article espouses a very different position; that MapReduce is "good enough" (even if the current Hadoop implementation could be vastly improved). While it is true that a large class of algorithms are not amenable to MapReduce implementations,

> "MAPREDUCE, OF COURSE, IS NOT A SILVER BULLET, AND THERE HAS BEEN MUCH WORK PROBING ITS LIMITATIONS."

[a]Note that this article attempts to be precise when referring to MapReduce, the programming model, and Hadoop, the popular open-source implementations.

there exist alternative solutions to the same underlying problems that can be easily implemented in MapReduce. Staying in its confines allows more tightly integrated, robust, end-to-end solutions to heterogeneous large-data challenges.

To apply a metaphor, Hadoop is currently the large-scale data processing hammer of choice. We've discovered that, in addition to nails, there are actually screws— and it doesn't seem like hammering screws is a good idea. So instead of trying to invent a screwdriver, let's just get rid of the screws. If there are only nails, then our MapReduce hammer will work just fine. To be specific, much discussion in the literature surrounds the fact that iterative algorithms are not amenable to Map-Reduce. The (simple) solution, I suggest, is to avoid iterative algorithms!

"WE'VE DISCOVERED THAT, IN ADDITION TO NAILS, THERE ARE ACTUALLY SCREWS—AND IT DOESN'T SEEM LIKE HAMMERING SCREWS IS A GOOD IDEA."

I will attempt to support this somewhat radical thesis by exploring three large classes of problems that serve as the poster children for MapReduce bashing: iterative graph algorithms (e.g., PageRank), gradient descent (e.g., for training logistic regression classifiers), and expectation maximization (e.g., for training hidden Markov models, k-means). I begin with vague and imprecise notions of what "amenable" and "good enough" mean, but propose a concrete objective with which to evaluate competing solutions later.

This article captures my personal experiences as an academic researcher, as well as a software engineer, in a production analytics environment. As an academic, I've been fortunate enough to collaborate with many wonderful colleagues and students on "big data" since 2007, primarily using Hadoop to scale a variety of text- and graph-processing algorithms (e.g., information retrieval, statistical machine translation, DNA sequence assembly). Recently, I've just returned from spending an extended two-year sabbatical at Twitter, "in the trenches" as a software engineer wrestling with various "big data" problems and trying to build scalable production solutions.

In earnest, I quip "throw away everything not a nail," tongue-in-cheek to make a point. More constructively, I suggest a two-pronged approach to the development of "big data" systems and frameworks. Taking the metaphor a bit further (and at the expense of overextending it), on the one hand, we should perfect the hammer we already have by improving its weight balance, making a better grip, etc. On the other hand, we should be developing jackhammers—entirely new "game changers" that can do things MapReduce and Hadoop fundamentally cannot do. In my opinion, it makes less sense to work on solving classes of problems for which Hadoop is already "good enough."

Iterative Graph Algorithms

Everyone's favorite example to illustrate the limitations of MapReduce is PageRank (or more generally, iterative graph algorithms). Let's assume a standard definition of a directed graph $G = (V, E)$ consisting of vertices V and directed edges E, with $S(v_i) = \{v_j \mid (v_i, v_j) \in E\}$ and $P(v_i) = \{v_j \mid (v_j, v_i) \in E\}$ consisting of the set of all successors and predecessors of vertex v_i (outgoing and incoming edges, respectively). PageRank[48] is defined as the stationary distribution over vertices by a random walk over the graph. That is, for each vertex v_i in the graph, PageRank computes the value $\text{PR}(v_i)$, representing the likelihood that a random walk will arrive at vertex v_i. This value is primarily induced from the graph topology, but the computation also includes a damping factor d, which allows for random jumps to any other vertex in the graph. For nontrivial graphs, PageRank is generally computed iteratively over multiple timesteps t using the power method:

$$\text{PR}(v_i; t) = \begin{cases} 1/|V| & \text{if } t = 0 \\ \frac{1-d}{|V|} + d \sum_{v_j \in P(v_i)} \frac{\text{PR}(v_j; t-1)}{|S(v_j)|} & \text{if } t > 0 \end{cases} \quad (1)$$

The algorithm iterates until either a user-defined maximum number of iterations has completed, or the values sufficiently converge. One common convergence criterion is:

$$\sum |\text{PR}(v_i; t) - \text{PR}(v_i; t-1)| < \epsilon \quad (2)$$

The standard MapReduce implementation of PageRank is well known and is described in many places (see, for example, Ref. 37). The graph is serialized as adjacency lists for each vertex, along with the current PageRank value. Mappers process all the vertices in parallel: For each vertex on the adjacency list, the mapper emits an intermediate key-value pair with the destination vertex as the key and the partial PageRank contribution as the value (i.e., each vertex distributes its present PageRank value evenly to its successors). The shuffle stage performs a large "group by," gathering all key-value pairs with the same destination vertex, and each reducer sums up the partial PageRank contributions.

Each iteration of PageRank corresponds to a MapReduce job.[b] Typically, running PageRank to convergence requires dozens of iterations. This is usually handled by a control program that sets up the MapReduce job, waits for it to complete, and then checks for convergence by reading in the

---

[b]This glosses over the treatment of the random jump factor, which is not important for the purposes here, but see Ref. 37.

updated PageRank vector and comparing it with the previous. This cycle repeats until convergence. Note that the basic structure of this algorithm can be applied to a large class of "message-passing" graph algorithms[39,42] (e.g., breadth-first search follows exactly the same form).

There is one critical detail necessary for the above approach to work: The mapper must also emit the adjacency list with the vertex id as the key. This passes the graph structure to the reduce phase, where it is reunited (i.e., joined) with the updated PageRank values. Without this step, there would be no way to perform multiple iterations.

There are many shortcomings with this algorithm:

- MapReduce jobs have high startup costs (in Hadoop, they can be tens of seconds on a large cluster under load). This places a lower bound on iteration time.
- Scale-free graphs, whose edge distributions follow power laws, often create stragglers in the reduce phase. The highly uneven distribution of incoming edges to vertices produces significantly more work for some reduce tasks (take, for example, the reducer assigned to sum up the incoming PageRank contributions to google.com in the webgraph). Note that since these stragglers are caused by data skew, speculative execution[17] cannot solve the problem. Combiners and other local aggregation techniques alleviate but do not fully solve this problem.
- At each iteration, the algorithm must shuffle the graph structure (i.e., adjacency lists) from the mappers to the reducers. Since, in most cases, the graph structure is static, this represents wasted effort (sorting, network traffic, etc.).
- The PageRank vector is serialized to HDFS, along with the graph structure, at each iteration. This provides excellent fault tolerance, but at the cost of performance.

To cope with these shortcomings, a number of extensions to MapReduce or alternative programming models have been proposed. Pregel[42] implements the Bulk Synchronous Parallel model:[52] Computations are "vertex-centric" and algorithms proceed in supersteps with synchronization barriers between each. In the implementation, all state, including the graph structure, is retained in memory (with periodic checkpointing). HaLoop[12] is an extension of Hadoop that provides support for iterative algorithms by scheduling tasks across iterations in a manner that exploits data locality and by adding various caching mechanisms. In Twister,[23] another extension of Hadoop designed for iteration, intermediate data are retained in memory if possible, thus greatly reducing iteration overhead. PrIter,[56] in contrast, takes a slightly different approach to speeding up iterative computation: It *prioritizes* those computations that are likely to lead to convergence.

All the frameworks discussed above share in supporting iterative constructs and thus elegantly solve one or more of the shortcomings of MapReduce discussed previously. However, they all have one drawback: They're not Hadoop! The reality is that the Hadoop-based stack (e.g., Pig, Hive, etc.) has already gained critical mass as the data processing framework of choice, and there are nontrivial costs for adopting a *separate* framework just for graph processing or iterative algorithms. More on this point in section titled "What's Good Enough."

For now, consider three additional factors. First, without completely abandoning MapReduce, there are a few simple "tweaks" that one can adopt to speed up iterative graph algorithms. For example, the Schimmy pattern[39] avoids the need to shuffle the graph structure by consistent partitioning and performing a parallel merge join between the graph structure and incoming graph messages in the reduce phase. The authors also show that great gains can be obtained by simple partitioning schemes that increase opportunities for partial aggregation.

Second, some of the shortcomings of PageRank in MapReduce are not as severe as the literature suggests. In a real-world context, PageRank (or any iterative graph algorithm) is almost never computed from scratch (i.e., initialized with a uniform distribution over all vertices and run until convergence). Typically, the previously computed PageRank vector is supplied as a starting point on an updated graph. For example, in the webgraph context, the hyperlink structure is updated periodically from freshly crawled pages, and the task is to compute updated PageRank values. It makes little sense to reinitialize the PageRank vector and "start over." Initializing the algorithm with the previously computed values significantly reduces the number of iterations required to converge. Thus, the iteration penalties associated with MapReduce become much more tolerable.

Third, the existence of graph-streaming algorithms for computing PageRank[49] suggests that there may be noniterative solutions (or at least approximations thereof) to a large number of iterative graph algorithms. This, combined with a good starting distribution (previous point), suggests that we can compute solutions efficiently, even within the confines of MapReduce.

Given these observations, perhaps we might consider MapReduce to be "good enough" for iterative graph algorithms? But what exactly does "good enough" mean? Let's return to this point in section titled "What's Good Enough."

Gradient Descent

Gradient descent (and related quasi-Newton) methods for machine learning represent a second large class of problems that are poorly suited for MapReduce. To explain, let's consider a specific type of machine-learning problem, supervised classification. We define X to be an input space and Y to be an output space. Given a set of training samples

$D = \{(x_i, y_i)\}_{i=1}^n$ from the space $X \times Y$, the task is to induce a function $f : X \to Y$ that best explains the training data. The notion of "best" is usually captured in terms of minimizing "loss," via a function that quantifies the discrepancy between the functional prediction $f(x_i)$ and the actual output y_i, for example, minimizing the quantity:

$$\frac{1}{n} \sum_{i=0}^{n} \ell(f(x_i), y_i) \tag{3}$$

which is known as the *empirical risk*. Usually, we consider a family of functions $\mathcal{F}$ (i.e., the hypothesis space) that is parameterized by the vector θ, from which we select:

$$\arg \min_{\theta} \frac{1}{n} \sum_{i=0}^{n} \ell(f(x_i; \theta), y_i) \tag{4}$$

That is, we learn the parameters of a particular model. In other words, machine learning is cast as a functional optimization problem, often solved with gradient descent.

Rewriting Equation (4) as $\arg\min_w L(\theta)$ simplifies our notation. The gradient of L, denote ∇L, is defined as follows:

$$\nabla L(\theta) = \left[\frac{\partial L(\theta)}{\partial w_0}, \frac{\partial L(\theta)}{\partial w_1}, \cdots \frac{\partial L(\theta)}{\partial w_d} \right] \tag{5}$$

The gradient defines a vector field pointing to the direction in which L is increasing the fastest and whose magnitude indicates the rate of increase. Thus, if we "take a step" in the direction opposite of the gradient from an arbitrary point a, b, = a $- \gamma \nabla L$(a), then L(a) $\geq L$(b), provided that γ (known as the step size) is a small value greater than zero.

If we start with an initial guess of $\theta^{(0)}$ and repeat the above process, we arrive at gradient descent. More formally, let us consider the sequence $\theta^{(0)}, \theta^{(1)}, \theta^{(2)} \ldots$, defined with the following update rule:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma^{(t)} \nabla L(\theta^{(t)}) \tag{6}$$

We have:

$$L(\theta^{(0)}) \geq L(\theta^{(1)}) \geq L(\theta^{(2)}) \ldots \tag{7}$$

where the sequence converges to the desired local minimum. If the loss function is convex and γ is selected carefully (which can vary per iteration), we are guaranteed to converge to a global minimum.

Based on the observation that our loss function decomposes linearly, and therefore the gradient as well, the MapReduce implementation of gradient descent is fairly straightforward. We process each training example in parallel and compute its partial contribution to the gradient, which is emitted as an intermediate key-value pair and shuffled to a single reducer. The reducer sums up all partial gradient contributions and updates the model parameters. Thus, each iteration of gradient descent corresponds to a MapReduce job. Two more items are needed to make this work:

- Complete classifier training requires many MapReduce jobs to be chained in a sequence (hundreds, even thousands, depending on the complexity of the problem). Just as in the PageRank case, this is usually handled by a driver program that sets up a MapReduce job, waits for it to complete, and then checks for convergence, repeating as long as necessary.
- Since mappers compute partial gradients with respect to the training data, they require access to the current model parameters. Typically, the parameters are loaded in as "side data" in each mapper (in Hadoop, either directly from HDFS or from the distributed cache). However, at the end of each iteration the parameters are updated, so it is important that the updated model is passed to the mappers at the next iteration.

Any number of fairly standard optimizations can be applied to increase the efficiency of this implementation, for example, combiners to perform partial aggregation or the in-mapper combining pattern.[37] As an alternative to performing gradient descent in the reducer, we can substitute a quasi-Newton method such as L-BFGS[41] (which is more expensive, but converges in few iterations). However, there are still a number of drawbacks:

- As with PageRank, Hadoop jobs have high startup costs.
- Since the reducer must wait for all mappers to finish (i.e., all contributions to the gradient to arrive), the speed of each iteration is bound by the *slowest* mapper, and hence sensitive to stragglers. This is similar to the PageRank case, except in the map phase.
- The combination of stragglers and using only a single reducer potentially causes poor cluster utilization. Of course, the cluster could be running other jobs, so from a throughput perspective, this is only a minor concern.

The shortcomings of gradient descent implementations in MapReduce have prompted researchers to explore alternative architectures and execution models that address these issues. All the systems discussed previously in the context of PageRank are certainly relevant, but we point out two more alternatives. Spark[54] introduces the resilient distributed datasets (RDD) abstraction, which provides a restricted form of shared memory based on coarse-grained transformations rather than fine-grained updates to shared state. RDDs can either be cached in memory or materialized from durable storage when needed (based on lineage, which is the sequence of transformations applied to the data). Classifier training is one of the demo applications in Spark. Another approach with similar goals is taken by Bu et al.,[11] who translate iterative MapReduce and Pregel-style programs into recursive

queries in Datalog. By taking this approach, database query optimization techniques can be used to identify efficient execution plans. These plans are then executed on the Hyracks data-parallel processing engine.[7]

In contrast to these proposed solutions, consider an alternative approach. Since the bottleneck in gradient descent is the iteration, let's simply get rid of it! Instead of running *batch* gradient descent to train classifiers, let us adopt *stochastic* gradient descent, which is an *online* technique. The simple idea is that instead of updating the model parameters after only considering *every* training example, let us update the model after *each* training example (i.e., compute the gradient with respect to each example).

Online learning techniques have received renewed interest in the context of big data since they operate in a streaming fashion and are very fast.[9,10,35,50] In practice, classifiers trained using online gradient descent achieve accuracy comparable to classifiers trained using traditional batch learning techniques but are an order of magnitude (or more) faster to train.[9]

In stochastic gradient descent, the learner streams through all the training examples, examines them one at a time, and updates its internal state (i.e., the model parameters) after processing each example. Often, it suffices to make a single pass over the data, but even if multiple passes are taken, the "iteration" is trivial—merely reading the data multiple times, requiring no external synchronization or exchange of state. In efficient implementations of stochastic gradient descent, the bottleneck is usually disk throughput in being able to deliver the training examples fast enough to the learner.

Stochastic gradient descent addresses the iteration problem, but does not solve the single reducer problem. For that, ensemble methods come to the rescue.[19,33] Instead of training a single classifier, let us train an *ensemble* of classifiers and combine predictions from each (e.g., simple majority voting, weighted interpolation, etc.). The simplest way of building ensembles—training each classifier on a partition of the training examples—is both embarrassingly parallel and surprisingly effective in practice.[43,44]

Combining online learning with ensembles addresses the shortcomings of gradient descent in MapReduce. As a case study, this is how Twitter integrates machine learning into Pig in a scalable fashion:[38] folding the online learning inside storage functions and building ensembles by controlling data partitioning. To reiterate the argument, if MapReduce is not amenable to a particular class of algorithms, let's simply find a different class of algorithms that will solve the same problem and *is* amenable to MapReduce.

Expectation Maximization

A third class of algorithms not amenable to MapReduce is expectation maximization (EM)[18] and EM-like algorithms. Since EM is related to gradient descent (both are first-order optimization techniques) and many of my arguments are quite similar, the discussion in this section will be more superficial.

> **"WHETHER AN ALGORITHM IS 'AMENABLE' TO MAPREDUCE IS A RELATIVE JUDGMENT THAT IS ONLY MEANINGFUL IN THE CONTEXT OF AN ALTERNATIVE."**

EM is an iterative algorithm that finds a successive series of parameter estimates $\theta^{(0)}, \theta^{(1)}, \ldots$ that improve the marginal likelihood of the training data, used in cases where there are incomplete (or unobservable) data. The algorithm starts with some initial set of parameters $\theta^{(0)}$ and then updates them using two steps: expectation (E-step), which computes the posterior distribution over the latent variables given the observable data and a set of parameters $\theta^{(i)}$, and maximization (M-step), which computes new parameters $\theta^{(i+1)}$ maximizing the expected log likelihood of the joint distribution with respect to the distribution computed in the E-step. The process then repeats with these new parameters. The algorithm terminates when the likelihood remains unchanged.

Similar to iterative graph algorithms and gradient descent, each EM iteration is typically implemented as a Hadoop job, with a driver to set up the iterations and check for convergence. In broad strokes, the E-step is performed in the mappers and the M-step is performed in the reducers. This setup has all the shortcomings discussed before, and EM and EM-like algorithms can be much more elegantly implemented in alternative frameworks that better support iteration (e.g., those presented previously).

Let's more carefully consider terms that I've been using quite vaguely. What does it mean for an algorithm to be *amenable* to MapReduce? What does it mean for MapReduce to be "good enough"? And the point of comparison? Here are two case studies that build up to my point.

Dyer et al.[22] applied MapReduce to training translation models for a statistical machine translation system—specifically, the word-alignment component that uses hidden Markov models (HMMs) to discover word correspondences across bilingual corpora.[51] The point of comparison was GIZA++,[c] a widely adopted in-memory, single-threaded

---

[c]code.google.com/p/giza-pp/

implementation (the *de facto* standard used by researchers at the time the work was performed, and still commonly used today). The authors built a Hadoop-based implementation of the HMM word-alignment algorithm, which demonstrated linear scalability compared to GIZA++, reducing per-iteration training time from hours to minutes. The implementation exhibited all the limitations associated with EM algorithms (high job startup costs, awkward passing of model parameters from one iteration to the next, etc.), yet compared to the previous single-threaded approach, MapReduce represented a step forward.[d] Here is the key point: whether an algorithm is "amenable" to MapReduce is a relative judgment that is only meaningful in the context of an alternative. Compared to GIZA++, the Hadoop implementation represented an advance. However, this is not inconsistent with the claim that EM algorithms could be more elegantly implemented in an alternate model that better supports iteration (e.g., any of the work discussed previously).

The second example is the venerable Lloyd's method for *k*-means clustering, which can be understood in terms of EM (not exactly EM, but can be characterized as EM-like). A MapReduce implementation of *k*-means shares many of the limitations discussed thus far. It is true that the algorithm can be expressed in a simpler way using a programming model with iterative constructs and executed more efficiently with better iteration support (and indeed, many of the papers discussed above use *k*-means as a demo application). However, even within the confines of Map-Reduce, there has been a lot of work on optimizing clustering algorithms (e.g., Ref. 16, 24). It is not entirely clear how these improvements would stack up against using an entirely different framework. Here, is MapReduce "good enough"?

These two case studies provide the segue for my attempt to more clearly define what it means for MapReduce to be "good enough," and a clear objective for deciding between competing solutions.

What's "Good Enough"?

I propose a pragmatic, operational, engineering-driven criterion for deciding between alternative solutions to large-data problems. First, though, are my assumptions:

- The Hadoop stack, for better or for worse, has already become the *de facto* general-purpose, large-scale data processing platform of choice. As part of the stack, I include higher-level layers such as Pig and Hive.
- Complete, end-to-end, large-data solutions involve heterogeneous data sources and must integrate dif-

ferent types of processing—relational processing, graph analysis, text mining, machine learning, etc.
- No single programming model or framework can excel at every problem; there are always tradeoffs between simplicity, expressivity, fault tolerance, performance, etc.

Given these assumptions, the decision criterion I propose is this: In the context of an end-to-end solution, would it make sense to adopt framework X (HaLoop, Twister, PrIter, Spark, etc.) over the Hadoop stack for solving the problem at hand?[e] Put another way: are the gains gotten from using X worth the integration costs incurred in building the end-to-end solution? If no, then operationally, we can consider the Hadoop stack (including Pig, Hive, etc., and by extension, Map-Reduce) to be "good enough."

Note that this way of thinking takes a broader view of end-to-end system design and evaluates alternatives in a global context. Considered in isolation, it naturally makes sense to choose the best tool for the job, but this neglects the fact that there are substantial costs in knitting together a patchwork of different frameworks, programming models, etc. The alternative is to use a common computing platform that's already widely adopted (in this case, Hadoop), even if it isn't a perfect fit for some of the problems.

I propose this decision criterion because it tries to bridge the big gap between "solving" a problem (in a research paper) and deploying the solution in production (which has been brought into stark relief for me personally based on my experiences at Twitter). For something to "work" in production, the solution must be continuously running, processes need to be monitored, someone needs to be alerted when the system breaks, etc. Introducing a new programming model, framework, etc., significantly complicates this process—even mundane things like getting the data imported into the right format and results exported to the right location become nontrivial if it's part of a long chain of dependencies.

A natural counter-argument would be: Why should academics be concerned with these (mere) "production issues"? This ultimately comes down to one's criteria for success. For me personally, the greatest reward comes from seeing my algorithms and code "in the wild," whether it's an end-to-end userfacing service that millions are using on a daily basis, or an internal improvement in the stack that makes engineers' and data scientists' lives better. I consider myself incredibly lucky to have accomplished both during my time at Twitter. I firmly believe that in order for any work to have a meaningful impact (in the way that I define it, recognizing, of course, that others are guided by different utility functions), how a

---

[d]HMM training is relatively expensive computationally, so job startup costs are less of a concern. Furthermore, these algorithms typically run for less than a dozen iterations.
[e]Hadoop is already a proven production system, whereas all the alternatives are at best research prototypes; let's even say for the sake of argument that X has already been made production-ready.

particular solution fits into the broader ecosystem is an important consideration.[f]

Different programming models provide different ways of thinking about the problem. MapReduce provides "map" and "reduce," which can be composed into more complex dataflows (e.g., via Pig). Other programming models are well suited to certain types of problems precisely because they provide a different way of thinking about the problem. For example, Pregel provides a vertex-centered approach in which "time" is dictated by the steady advance of the superstep synchronization barriers. We encounter an impedance mismatch when trying to connect different frameworks that represent different ways of thinking. The advantages of being able to elegantly formulate a solution in a particular framework must be weighed against the costs of integrating that framework into an end-to-end solution.

> "OF COURSE IT MAKES SENSE TO USE THE RIGHT TOOL FOR THE JOB, BUT WE MUST ALSO RECOGNIZE THE COST ASSOCIATED WITH SWITCHING TOOLS—IN SOFTWARE ENGINEERING TERMS."

To illustrate, I'll present a hypothetical but concrete example. Let's say we wish to run PageRank on the interaction graph of a social network (i.e., the graph defined by *interactions* between users). Such a graph is *implicit* and needs to be constructed from behavior logs, which is natural to accomplish in a dataflow language such as Pig (in fact, Pig was exactly designed for log mining). Let's do exactly that.

With the interaction graph now materialized, we wish to run PageRank. Consider two alternatives: use Giraph,[g] the open-source implementation of Pregel, or implement PageRank directly in Pig.[h] The advantage of the first is that the BSP model implemented by Giraph/Pregel is perfect for PageRank and other iterative graph algorithms (in fact, that's exactly what Pregel was designed to do). The downside is lots of extra "plumbing": munging Pig output into a format suitable for Giraph, triggering the Giraph job, waiting for it to finish, and figuring out what to do with the output (if another Pig job depends on the results, then we must munge the data back into a form that Pig can use).[i] In the second alternative, we simply write PageRank in Pig, with all the shortcomings of iterative MapReduce algorithms discussed in this article. Each iteration might be slow due to stragglers, needless shuffling of graph structure, etc., but since we likely have the PageRank vector from yesterday to start from, the Pig solution would converge mercifully quickly. And with Pig, all of the addi-

tional "plumbing" issues go away. Given these alternatives, I believe the choice of the second is at least justifiable (and arguably, preferred), and hence, in this particular context, I would argue that MapReduce is good enough.

In my opinion, the arguments are even stronger for the case of stochastic gradient descent. Why adopt a separate machine-learning framework simply for running batch gradient descent when it could be seamlessly integrated into Pig by using stochastic gradient descent and ensemble methods?[38] This approach costs nothing in accuracy but gains tremendously in terms of performance. In the Twitter case study, machine learning is accomplished by *just another Pig script,* which plugs seamlessly into existing Pig workflows.

To recap, of course it makes sense to use the right tool for the job, but we must also recognize the cost associated with switching tools—in software engineering terms, the costs of integrating heterogeneous frameworks into an end-to-end workflow are nontrivial and should not be ignored. Fortunately, recent developments in the Hadoop project promise to substantially reduce the costs of integrating heterogeneous frameworks. Hadoop NextGen (aka YARN) introduces a generic resource scheduling abstraction that allows multiple application frameworks to coexist on the same physical cluster. In this context, MapReduce is just one of many possible application frameworks, others include Spark[j] and MPI.[k] This "meta-framework" could potentially reduce the costs of supporting heterogeneous programming models—an exciting future development that might let us "have our cake and eat it too." However, until YARN proves itself in production environments, it remains an unrealized potential.

Constructive Suggestions

Building on the arguments above and reflecting on my experiences over the past several years working on "big data" in both academia and industry, I'd like to make the following constructive suggestions.

Continue plucking low-hanging fruit, or, refine the hammer we already have. I do not think we have yet sufficiently

---

[f]As a side note, unfortunately, the faculty promotion and tenure process at most institutions do not reward these activities, and in fact, some would argue actively have a disincentive effect on these activities since they take time away from writing papers and grants.

[g]incubator.apache.org/giraph/.

[h]techblug.wordpress.com/2011/07/29/ pagerank-implementation-in-pig/

[i]Not to mention all the error-reporting, alerting, error-handling mechanisms that now need to work across Pig and Giraph.

[j]github.com/mesos/spark-yarn.

[k]issues.apache.org/jira/browse/MAPREDUCE-2911.

pushed the limits of MapReduce in general and the Hadoop implementation in particular. In my opinion, it may be premature to declare it obsolete and call for a fresh ground-up redesign.[4,8] MapReduce is less than 10 years old, and Hadoop is even younger. There has already been plenty of interesting work within the confines of Hadoop, just from the database perspective: integration with a traditional RDBMS,[1,3] smarter task scheduling,[53,55] columnar layouts,[25,27,28,29,40] embedded indexes,[20,21] cube materialization,[45] and a whole cottage industry on efficient join algorithms;[31,36,47] we've even seen traditional HPC ideas, such as work stealing adapted to Hadoop.[34] Much more potential remains untapped.

The data management and distributed systems communities have developed and refined a large "bag of tricks" over the past several decades. Researchers have tried applying many of these in the Hadoop context (see above), but there are plenty remaining in the bag waiting to be explored. Many, if not most, of the complaints about Hadoop lacking basic features or optimization found in other data processing systems can be attributed to immaturity of the platform, not any fundamental limitations. More than a "matter of implementation," this work represents worthy research. Hadoop occupies a very different point in the design space when compared to parallel databases, so the "standard tricks" often need to be reconsidered in this new context.

So, in summary, let's fix all the things we have a good idea how to fix in Hadoop (low-risk research), and then revisit the issue of whether MapReduce is good enough. I believe this approach of incrementally refining Hadoop has a greater chance of making impact (at least by my definition of impact in terms of adoption) than a strategy that abandons Hadoop. To invoke another cliché, let's pluck all the low-hanging fruit first before climbing to the higher branches.

Work on game changers, or, develop the jackhammer. To displace (or augment) MapReduce, we should focus on capabilities that the framework fundamentally cannot support. To me, faster iterative algorithms illustrated with PageRank or gradient descent aren't "it"—given my prior arguments on how, for those, MapReduce is "good enough." I propose two potential game changers that reflect pain points I've encountered during my time in industry.

First, real-time computation on continuous, large-volume streams of data is not something that MapReduce is capable of doing. MapReduce is fundamentally a batch processing framework—and despite efforts in implementing "online" MapReduce,[15] I believe solving the general problem requires something that looks very different from the current architecture. For example, let's say I want to keep track of the top thousand most-clicked URLs posted on Twitter in the last n

minutes. The current solution is to run batch MapReduce jobs with increasing frequency (e.g., every five minutes), but there is a fundamental limit to this approach (job startup time), and (near) real-time results are not obtainable (for example, if I wanted up-to-date results over the last 30 seconds).

One sensical approach is to integrate a stream-processing engine—a stream oriented RDBMS (e.g., Refs. 13, 26, 32), S4,[46] or Storm[l]—with Hadoop, so that the stream-processing engine handles real-time computations, while Hadoop performs aggregate "roll ups." More work is needed along these lines, and indeed researchers are already beginning to explore this general direction.[14] I believe the biggest challenge here is to seamlessly and efficiently handle queries across vastly different time granularities, from "over the past 30 seconds" (in real time) to "over the last month" (where batch computations with some lag would be acceptable).

Second, and related to the first, real-time interactions with large datasets are a capability that is sorely needed, but is something that MapReduce fundamentally cannot support. The rise of "big data" means that the work of data scientists is increasingly important—after all, the value of data lies in the insights that they generate for an organization. Tools available to data scientists today are primitive. Write a Pig script and submit a job. Wait five minutes for the job to finish. Discover that the output is empty because of the wrong join key. Fix simple bug. Resubmit. Wait another five minutes. Rinse, repeat. It's fairly obvious that long debug cycles hamper rapid iteration. To the extent that we can provide tools to allow rich, interactive, incremental interactions with large data sets, we can boost the productivity of data scientists, thereby increasing their ability to generate insights for the organization.

Open source everything. Open source releasing of software should be the default for any work that is done in the "big data" space. Even the harshest critic would concede that open source is a key feature of Hadoop, which facilitates rapid adoption and diffusion of innovation. The vibrant ecosystem of software and companies that exist today around Hadoop can be attributed to its open source license.

Beyond open sourcing, it would be ideal if the results of research papers were submitted as patches to existing open source software (i.e., associated with JIRA tickets). An example is recent work on distributed cube materialization,[45] which has been submitted as a patch in Pig.[m] Of course, the costs associated with this can be substantial, but this represents a great potential for collaborations between academia and industry; committers of open source projects (mostly software engineers in industry) can help shepherd the patch. In many cases, transitioning academic research projects to

---

[l]github.com/nathanmarz/storm
[m]issues.apache.org/jira/browse/PIG-2167

production-ready code makes well-defined summer internships at companies. These are win–win scenarios for all. The company benefits immediately from new features, the community benefits from the open sourcing, and the students gain valuable experience.

Conclusion

The cliché is "If all you have is a hammer, then everything looks like a nail." I argue for going one step further: "If all you have is a hammer, throw away everything that's not a nail!" It'll make your hammer look amazingly useful. At least for some time. Sooner or later, however, the flaws of the hammer will be exposed—but let's try to get as much hammering done as we can before then. While we're hammering, though, nothing should prevent us from developing jackhammers.

Author Disclosure Statement

No conflicts of interest exist.

Acknowledgments

I'd like to thank Jeffrey Ullman for comments on a previous version of this article, as well as Daniel Abadi for the encouragement to stand by the controversial message.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. SIGMOD 2009.
2. Afrati, F., Sarma, A., Salihoglu, S., and Ullman, J. Vision paper: Towards an understanding of the limits of MapReduce computation. arXiv:1204.1754v1 2012.
3. Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Paulson, E. Efficient processing of data warehousing queries in a split execution environment. SIGMOD 2011.
4. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., and Warneke, D. Nephele/-PACTs: A programming model and execution framework for web-scale analytical processing. SoCC 2010.
5. Bhatotia, P., Wieder, A., Akkus, I., Rodrigues, R., and Acar, U. Large-scale incremental data processing with change propagation. HotCloud 2011.
6. Blanas, S., Patel, J., Ercegovac, V., Rao, J., Shekita, E., and Tian, Y. A comparison of join algorithms for log processing in MapReduce. SIGMOD 2010.
7. Borkar, V., Carey, M., Grover, R., Onose, N., and Vernica, R. Hyracks: A flexible and extensible foundation for data-intensive computing. ICDE 2011.
8. Borkar, V., Carey, M., and Li, C. Inside "big data management": Ogres, onions, or parfaits? EDBT 2012.
9. Bottou, L. Large-scale machine learning with stochastic gradient descent. COMPSTAT 2010.
10. Bottou, L., and LeCun, Y. Large scale online learning. NIPS 2003.
11. Bu, Y., Borkar, V., Carey, M., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., and Ramakrishnan, R. Scaling Datalog for machine learning on big data. arXiv:1203.0160v2 2012.
12. Bu, Y., Howe, B., Balazinska, M., and Ernst, M. HaLoop: Efficient iterative data processing on large clusters. VLDB 2010.
13. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. Monitoring streams—a new class of data management applications. VLDB 2002.
14. Chandramouli, B., Goldstein, J., and Duan, S. Temporal analytics on big data for web advertising. ICDE 2012.
15. Condie, T., Conway, N., Alvaro, P., Hellerstein, J., Elmeleegy, K., and Sears, R. MapReduce online. NSDI 2010.
16. Cordeiro, R., Traina, C., Traina, A., López, J., Kang, U., and Faloutsos, C. Clustering very large multi-dimensional datasets with MapReduce. SIGKDD 2011.
17. Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. OSDI 2004.
18. Dempster, A., Laird, N., and Rubin, D. Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society 1977; 39:1–38.
19. Dietterich, T. Machine learning research: Four current directions. AI Magazine 1997; 18:97–136.
20. Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). VLDB 2010.
21. Dittrich, J., Quiané-Ruiz, J.-A., Richter, S., Schuh, S., Jindal, A., and Schad, J. Only aggressive elephants are fast elephants. VLDB 2012.
22. Dyer, C., Cordova, A., Mont, A., and Lin, J. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. SMT Workshop 2008.
23. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. Twister: A runtime for iterative MapReduce. MAPREDUCE Workshop 2010.
24. Ene, A., Im, S., and Moseley, B. Fast clustering using MapReduce. SIGKDD 2011.
25. Floratou, A., Patel, J., Shekita, E., and Tata, S. Column-oriented storage techniques for MapReduce. VLDB 2011.
26. Gehrke, J. Special issue on data stream processing. Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society, 2003; 26.
27. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., and Xu, Z. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. ICDE 2011.
28. Jindal, A., Quiané-Ruiz, J.-A., and Dittrich, J. Trojan data layouts: Right shoes for a running elephant. SoCC 2011.

29. Kaldewey T., Shekita, E., and Tata., S. Clydesdale: Structured data processing on MapReduce. EDBT 2012.

30. Karloff, H., Suri, S., and Vassilvitskii, S. A model of computation for MapReduce. SODA 2010.

31. Kim, Y., and Shim, K. Parallel top-k similarity join algorithms using MapReduce. ICDE 2012.

32. Krishnamurthy, S., Franklin, M., Davis, J., Farina, D., Golovko, P., Li, A., and Thombre, N. Continuous analytics over discontinuous streams. SIGMOD 2010.

33. Kuncheva., L. Combining Pattern Classifiers: Methods and Algorithms. New York: Wiley-Interscience 2004.

34. Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. SkewTune: Mitigating skew in MapReduce applications. SIGMOD 2012.

35. Langford, J., Li, L., and Zhang, T. Sparse online learning via truncated gradient. JMLR 2009; 10:777–801.

36. Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., and Zhang, X. YSmart: Yet another SQL-to-MapReduce translator. ICDCS 2011.

37. Lin, J., and Dyer, C. Data-Intensive Text Processing with MapReduce. San Rafael, CA: Morgan & Claypool Publishers 2010.

38. Lin, J., and Kolcz, A. Large-scale machine learning at Twitter. SIGMOD 2012.

39. Lin, J., and Schatz, M. Design patterns for efficient graph algorithms in MapReduce. MLG Workshop 2010.

40. Lin, Y., Agrawal, D., Chen, C., Ooi, B., and Wu, S. Llama: Leveraging columnar storage for scalable join processing in the MapReduce framework. SIGMOD 2011.

41. Liu, D., Nocedal, J., Liu, D., and Nocedal, J. On the limited memory BFGS method for large scale optimization. Mathematical Programming B 1989; 45:503–528.

42. Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. SIGMOD 2010.

43. Mann, G., McDonald, R., Mohri, M., Silberman, N., and Walker, D. Efficient largescale distributed training of conditional maximum entropy models. NIPS 2009.

44. McDonald, R., Hall, K., and Mann, G. Distributed training strategies for the structured perceptron. HLT/NAACL 2010.

45. Nandi, A., Yu, C., Bohannon, P., and Ramakrishnan, R. Distributed cube materialization on holistic measures. ICDE 2011.

46. Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. S4: Distributed stream computing platform. KDCloud Workshop 2010.

47. Okcan, A., and Riedewald, M. Processing theta-joins using MapReduce. SIGMOD 2011.

48. Page, L., Brin, S., Motwani, R., and Winograd, T. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford University 1999.

49. Sarma, A., Gollapudi, S., and Panigrahy, R. Estimating PageRank on graph streams. PODS 2008.

50. Shalev-Shwartz, S., Singer, Y., and Srebro, N. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. ICML 2007.

51. Tiedemann, J. Bitext Alignment. San Rafael, CA: Morgan & Claypool Publishers 2011.

52. Valiant, L. A bridging model for parallel computation. Communications of the ACM 1990; 33:103–111.

53. Zaharia, M., Borthakur, D., Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Job scheduling for multi-user MapReduce clusters. Technical report, Berkeley 2009.

54. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI 2012.

55. Zaharia, M., Konwinski, A., Joseph, A., Katz, R., and Stoica, I. Improving MapReduce performance in heterogeneous environments. OSDI 2008.

56. Zhang, Y., Gao, Q., Gao, L., and Wang, C. PrIter: A distributed framework for prioritized iterative computations. SoCC 2011.

Address correspondence to:

Jimmy Lin
The iSchool
University of Maryland
College Park, MD 20742-4235

jimmylin@umd.edu