Machine learning in ScalOps, a higher order cloud computing language

Markus Weimer, Tyson Condie, Raghu Ramakrishnan
Yahoo! Labs, Santa Clara, CA 95054
[weimer|tcondie|ramakris]@yahoo-inc.com

1 Introduction

Machine learning practitioners are increasingly interested in applying their algorithms to Big Data. Unfortunately, current high-level languages for data analytics in the cloud (e.g., [2, 15, 16, 5]) do not fully cover this domain. One key missing ingredient is means to express *iteration* over the data (e.g., [19]). Zaharia et al., were the first to answer this call from a systems perspective with Spark [20]. Spark is built on a data abstraction called resilient distributed datasets (RDDs) that reference immutable data collections. The Spark domain-specific language (DSL) defines standard relational algebra transformations—selection, join, group by, etc.—and mechanisms to cache RDDs in memory. The Spark runtime is optimized for *in-memory* computation and has published speed-ups of $30\times$ over Hadoop MapReduce for many machine learning and graph algorithms, which is in line with the gains found using MPI or even special case implementations (e.g. [18]).

However, there still is not a single platform that can capture the entire analytics pipeline for machine learning. Thus, it is common to compose multiple Pig, MPI and MapReduce programs into large work-flows. This fractioning of individual processing steps can be a major pain e.g., for optimization, debugging, and code readability. Our prescription to this dilemma is a new DSL for data analytics called *ScalOps*. Like Pig, ScalOps combines the declarative style of SQL and the low-level procedural style of MapReduce. Like Spark, ScalOps can optimize its runtime—the Hyracks parallel-database engine [3]—for repeated access to data collections. It differs from prior platforms in the following ways, allowing it to handle a wider range of analytics over any (big or small) data.

1. We provide an explicit **loop** construct that captures the iteration in the runtime plan.
2. We use *recursive* query plans to execute iterations in a data-parallel runtime called Hyracks.
3. Hyracks is optimized for computations over both in-memory and external memory datasets.

In this abstract, we describe the ScalOps language from a machine learning perspective by presenting two examples. We begin with a discussion of the related work in Section 2. Section 3 explores two well-known examples from the machine learning and graph analytics domain expressed in ScalOps. In Section 3.1, we describe a ScalOps implementation of batch gradient descent—the quintessentially parallel machine learning algorithm. We then show in Section 3.2 how the graph computation framework like Pregel [14] can be implemented in a few lines of ScalOps code. Section 4 summarizes our current work and proposes some language-level extensions.

2 Related Work

ScalOps draws inspiration from parallel [8, 9] and deductive databases [17], and frameworks for machine learning and graph analysis [13, 14, 20, 10].

Data-parallel computing is an active research field. Since the initial MapReduce paper [7], systems like Dryad [11] and Hyracks [3] have successfully made the case for supporting a richer set of operators beyond map and reduce. These systems are built on a shared-nothing database architecture, which has been proven to achieve very high scalability via partitioned and pipelined parallelism [8, 9]. Of these systems, Hyracks shows the most promise as being a runtime that can support and optimize *recursive* query plans (see [1, 17]) for efficient iterative analysis.

Pig [15] and DryadLINQ [12] reduce the accidental complexity of programming in the lower-level dataflow (e.g., MapReduce) by exposing a high-level declarative language. Both recognize the desire to express data transformations as a series of *imperative* steps but solve them in different ways. Pig is an external DSL that exposes declarative data transformations as expressions that can be assigned to variables and used in subsequent expressions. This adds the notion of a control flow to a Pig program that maybe more palatable to the MapReduce enthusiast. DryadLINQ is an internal DSL that inherits a control flow from its *host* language C#. Like DryadLINQ, ScalOps is an internal DSL and borrows its control flow from its host language Scala. Like Pig, ScalOps exposes similar declarative expressions that can be assigned to variables and referenced in later expressions.

Neither Pig nor DryadLINQ consider data caching in the context of iterative analysis; since their respective runtimes do not consider it. Bu et al., added iteration to a single MapReduce job and showed significant benefits to pipelining between the reduce and map steps [4]. Caching is an important optimization for iterative machine learning tasks [18]. Spark addresses this issue by allowing programmers to *explicitly* cache datasets for "very fast" iterative analysis [20].

The other area we draw experience from is distributed graph processing. Pregel [14] is a system developed at Google that applies a Bulk Synchronous Processing (BSP) model to graphs analytics. In Pregel, nodes send messages to neighboring nodes in the graph via a series of synchronized time-steps called supersteps. The Pregel runtime executes supersteps until no new messages are generated. [1] Parallelization and fault-tolerance in Pregel's runtime is similar to earlier BSP systems [7]. GraphLab [13], on the other hand proposes a computational abstraction based on *asynchronous* updates and omits the supersteps completely. GraphLab has demonstrated exceptional performance and scalability on single massively parallel machines and its abstraction has shown promise in the distributed setting.

3 Machine Learning Query Plans

ScalOps is a high-level language that moves beyond single pass data analytics (i.e., MapReduce) to include multi-pass workloads. ScalOps supports iteration over algorithms expressed as relational queries on the training and model data. ScalOps' expressiveness follows from Chu et al. [6]—showing that many ML algorithms are naturally represented in the MapReduce programming model. ScalOps goes beyond the prior work by (a) extracting expressions over the model data and (b) pushing iterations over query plans into the execution engine.

Today, the programmer is responsible for writing the *optimal* runtime plan, which is hard in a shared cloud computing environment. By providing a query plan abstraction, ScalOps facilitates runtime optimizations that consider hardware specifications, data statistics and the current load when generating a runtime plan. This liberates the programmer from having to write low-level code that convolutes the machine learning algorithm. This section presents the ScalOps DSL via two example algorithms: Batch Gradient Descent and Pregel [14]. The key observation is that ScalOps abstracts away low-level details with a succinct interpretation of the algorithm as code.

3.1 Map Reduce: Batch Gradient Descent

Consider the update rule for learning a L2-regularized linear model through batch gradient descent:

$$w_{t+1} = \left(w_t - \eta \sum_{x,y} \delta_{w_t}(l(y, \langle w_t, x \rangle)) \right) (1 - \eta\lambda) \tag{1}$$

Here, λ and η are the regularization constant and learning rate, and δ_{w_t} denotes the derivative with respect to w_t. The sum in (1) decomposes per data point (x, y) and can therefore be trivially parallelized over the data set. Furthermore, the application of the gradient computation can be phrased as `map` in a MapReduce framework, while the sum itself can be phrased as the `reduce` step, possibly implemented via an aggregation tree for additional performance.

[1] Pregel also adds a vote to halt protocol that can be modeled via sending a "self" message.

Listing 1: Batch Gradient Descent in ScalOps

```
1  def bgd(xy: Table[Example], g:(Example, Vector)=>Vector, e:Double, l:Double) =
2    loop(zeros(1000), 0 until 100) {
3      w => (w - (xy.map(x => g(x, w)).reduce(_+_) * e) * (1.0 - e * l)
4    }
```

Listing 2: Pregel in ScalOps

```
1  class Node(id: Int, neighbors: List[Int])
2  class Message(to: Int)
3
4  def pregel(nodes: Table[Node], f: (Node, Bag[Message]) => (Node, Bag[Message])) {
5    val messages = Table[Message]()
6
7    loop(False, (b: BooleanType) => !b) { b => {
8        val groups = cogroup(nodes by (_.id) outer, messages by (_.to))
9        val msgAndNodes = groups.map(e => f(e._2.head, e._3))
10       messages := msgAndNodes.map(_._2.FLATTEN)
11       nodes := msgAndNodes.map(_._1)
12       messages.isEmpty
13     }
14   }
15 }
```

Listing 1 shows an implementation of this parallelized batch gradient descent in ScalOps. Line 1 defines a function `bgd` with the following parameters: the input data `xy`, the gradient function `g` (δ_{w_t} in (1)), the learning rate `e` (η) and the regularization constant `l` (λ). Line 2 defines the body of the function to be a `loop`, which is ScalOps' looping construct. It accepts three parameters: (a) Input state — here, we assume 1000 dimensions in the weight vector — (b) a "while" condition that (in this case [2]) specifies a range (`0 until 100`) and (c) a loop body. The loop body is a function from input state to output state, which in this case is from a Vector `w` to its new value after the iteration. In the loop body, the gradient is computed for each data point via a `map` call and then summed up in a `reduce` call. Note that _+_ is a scala shorthand for the function literal `(x,y) => x+y`.

ScalOps bridges the gap between imperative and declarative code by using data structures and language constructs familiar to the machine learning domain. This is evident in Listing 1, which is nearly a 1:1 translation of Equation 1 and directly compiles to executable code. ScalOps query plans capture the semantics of the user's code in a form that can be reasoned over and optimized. For instance, since the query plan contains the loop information we can recognize the need to cache as much of `xy` in memory as possible, without explicit user hints. Furthermore, the expression inside of the loop body itself is also captured. [3] This could offer further optimization opportunities reminiscent to traditional compilers i.e., dead-code elimination and constant-propagation.

3.2 Bulk Synchronous Processing: Pregel

As a second example we discuss an implementation of Pregel [14] in ScalOps following a similar approach to the one described in [20] using Spark. Pregel is a graph-centric computation framework where computation happens in *supersteps*. In each superstep, the nodes of the graph process messages sent to them in parallel and produce a new set of messages and vertex state that is to be processed in the next superstep. The program terminates when there are no more messages to be processed. Nodes that have messages in a given superstep are referred to as *active* in that superstep. A node can ensure to be active in the next superstep by sending a message to itself.

Listing 2 shows an implementation of Pregel in ScalOps. Lines 1 and 2 define the `Node` and `Message` classes the user would typically subclass for the algorithm in question. Line 4 defines the function `pregel` which accepts the following parameters: The `Table nodes` containing the `Nodes` of the graph and a function `f` that accepts a `Node` and a set of `Messages` to produce an

[2]We also support boolean expressions on the state object i.e., the weight vector.

[3]We currently capture basic math operators (+, -, *, /) over primitive types (int, float, double, vector, etc.).

updated `Node` and a new set of `Messages` to be processed in the next superstep. The function `f` is assumed to be defined for the case of no messages, e.g. the first iteration where it shall emit the unchanged `Node` and no messages. Line 5 declares the (currently empty) message `Table` to be used within the loop. In Line 7, the `loop` operator is used to form the Pregel main loop. Here, the loop is not defined over a set range of iterations as above, but a boolean expression indicating whether the message table is empty. Its initial value is set to `false` and the loop continues as long as it stays `false` as defined by `(b: BooleanType) => !b`. Note that `BooleanType` is not a Scala type, but one defined by ScalOps that extracts expression trees to be part of the query plan. This facilitates certain optimizations (see below). Different from the Batch Gradient Descent example, the loop body now consists of multiple statements to implement a Pregel superstep. In Line 8, the messages are grouped with their target nodes. In Line 9 the update function is applied to compute new nodes and messages, resulting in a `Table[Node, Bag[Message]]`. This `Table` is then split into the updated nodes and new messages in Lines 10 and 11. Lastly, Line 12 returns whether the message table is empty to the `loop` operator.

Listing 2 compiles to a query plan that repeatedly executes supersteps while (some) vertices continue to produce messages. We can optimize this iterative query plan in two ways. Firstly, by observing that the `isEmpty` statement need only be **true** for one (message) instance in order to be globally **true**. This permits early detection of the need for a subsequent superstep; allowing the runtime environment to start planning its resource requirements for the next iteration. Secondly, the `cogroup` operator can include a function to pre-aggregate—or summarize—the output state. This is reminiscent to the `combiner` function in MapReduce and can be a very effective optimization to reducing network traffic and overall memory pressure.

4 Conclusion

ScalOps is a new internal domain-specific language (DSL) for Big Data analytics that targets machine learning and graph-based algorithms. It unifies the so-far distinct DAG processing as found in e.g. Pig and the iterative computation needs of machine learning in a single language and runtime. It exposes a declarative language that is reminiscent to Pig with iterative extensions. The `loop` block captures iteration in a recursive query plan so that it can be optimized for any given cloud environment. The Hyracks runtime directly supports these iterations as recursive data-parallel runtime plans, thereby avoiding the pitfalls of an outer driver loop. We highlighted the expressiveness of ScalOps by presenting two example implementations: Batch Gradient Descent—a trivially parallel algorithm—and Pregel, a computational framework of its own. The resulting code was nearly a 1:1 translation of the target mathematical description.

ScalOps benefits from being a DSL internal to the Scala language, primary in terms of programmer productivity and code readability, but also in optimization opportunities. In Scala, programmers have access to all Java libraries and benefits of running on a JVM (ubiquity, administrative tools, profiling, garbage collection, etc.). ScalOps seamlessly integrates with well-known IDEs (Eclipse, NetBeans, etc.) thanks to the respective Scala plugins. ScalOps inherits a *static* type-checker from Scala; something foreign to many existing data-flow languages. ScalOps uses Scala closures to package user code for runtime execution. In Pig, this code would be *external* Java code that programmers would need to explicitly package and ship to the runtime. Moreover, Pig lacks visibility into this code and can not optimize for associativity and commutativity properties in the absence of explicit hints. [4]

Looking forward, we plan on adding asynchronous extensions and a graph API akin to GraphLab. Graph-based analytics shows the most promise in terms of benefiting from a recursive query infrastructure. Much of the looping support described here is needlessly verbose when the underlying runtime supports it natively. Frameworks like GraphLab expose just the right amount of API code, and capture 'iteration" in the semantics. This leads to a natural recursive query representation of the GraphLab runtime but will first require the asynchronous extensions.

[4]In Pig, a UDF can inherit from a special base UDF class that indicates the function is associative and commutative.

References

[1] Foto N. Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 1–8, New York, NY, USA, 2011. ACM.

[2] The Hive Project. http://hive.apache.org/.

[3] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE*, pages 1151–1162. IEEE Computer Society, 2011.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proc. VLDB*, 2008.

[6] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[8] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2:44–62, March 1990.

[9] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35:85–98, June 1992.

[10] Philipp Haller and Heather Miller. Parallelizing machine learning-functionally: A framework and abstractions for parallel graph processing. In *The 2nd annual Scala Workshop*, Stanford, CA, June 2011.

[11] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[12] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 987–994, New York, NY, USA, 2009. ACM.

[13] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, page 6, New York, NY, USA, 2009. ACM.

[15] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4):227–298, 2005.

[17] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[18] Markus Weimer, Sriram Rao, and Martin Zinkevich. A convenient framework for efficient parallel multipass algorithms. In *LCCC : NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, December 2010.

[19] Jerry Ye, Jyh H. Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 2061–2064, New York, NY, USA, 2009. ACM.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.