Using R for Iterative and Incremental Processing

Shivaram Venkataraman
UC Berkeley
shivaram@eecs.berkeley.edu

Indrajit Roy Alvin AuYoung Robert S. Schreiber
HP Labs
{*indrajitr, alvina, rob.schreiber*}*@hp.com*

Abstract

It is cumbersome to write complex machine learning and graph algorithms in existing data-parallel models like MapReduce. Many of these algorithms are, by nature, iterative and perform incremental computations, neither of which are efficiently supported by current frameworks. We argue that array-based languages, like R [1], are ideal to express these algorithms, and we should extend these languages for processing in the cloud. In this paper we present the challenges and abstractions to extend R. Early results show that many computations are an order of magnitude faster than processing in Hadoop.

1 Introduction

Randomness combined with linear algebra is a powerful tool for modern problems.
—Ravi Kannan, FCRC 2011 [8]

Many real-world analytics applications are best formulated as iterative linear algebra operations. For example, PageRank and anomaly detection on graphs calculate eigenvectors of large matrices [3, 7] and recommendation systems implement matrix decomposition [17]. Even graph algorithms (shortest path, spanning tree, strongly connected components, etc.) primarily involve array manipulation and can be expressed with linear algebra operators such as matrix multiply [9].

Existing frameworks for large-scale processing [5, 6] are both ill-suited to express and inefficient to implement many applications of this type. For example, it is known that most large-scale frameworks are not efficient for matrix operations [14]. Linear algebraic algorithms execute on *structured data*, such as matrices, while MapReduce-like systems do not retain this structure.

An additional challenge of implementing these applications is that many are incremental in nature, and refine their underlying mathematical models by analyzing newly arriving data. For example, user recommendations in Netflix and Amazon should be updated as new ratings appear, PageRank recalculated as Web pages change, and spammers in a social network detected as they add spurious relationships. Current systems either don't support incremental computations or, if they do, are inefficient for linear algebra (Incoop [2]).

A major challenge of supporting incremental computations is creating appropriate bindings between the storage system and the processing framework. Caching data in memory [16] can speed up iterative programs, but current systems do not have primitives to maintain consistency in the presence of updates to the storage systems.

As our reliance upon linear algebra for analyzing massive data increases, we need large-scale systems that can express and efficiently implement such analysis.

We argue that array-based languages such as R provide a more appropriate programming model to express many machine learning and graph algorithms. The core construct of arrays make these languages ideal to represent vectors and matrices, and perform matrix operations. R has traditionally been the software of choice for machine learning users and statisticians, albeit for small problem sizes. It does not support scalable, fault-tolerant computations and was not built for incremental processing.

In this paper, we embrace R and extend it for large scale incremental processing. Our primary focus is on machine learning and graph algorithms. We hypothesize that by extending R, programmers will not only express the algorithms in a natural, linear algebraic formulation, but that the execution of the underlying implementation will also be significantly more efficient.

We present a prototype called Presto that extends R to run on a cluster and supports incremental processing. Early results show that several algorithms can be expressed in fewer than 140 lines of Presto code and are an order of magnitude faster than Hadoop implementations.

2 Challenges of using R

While R has over 2,000 packages for analysis, it is primarily used as a single threaded, single machine installation. R is not scalable nor does it support incremental processing. We list the challenges of using R, some of which are due to incremental processing while others arise because of the array-based programming model.

Structure and Scalability. Scaling R to run on a cluster has its challenges. Unlike MapReduce, Spark and others, where only one record is addressed at a time, the ease of array-based programming is due to a global view of data. R programs maintain the structure of data by mapping data to arrays and manipulating them. For example, graphs are represented as adjacency matrices and outgoing edges of a vertex are obtained from the corresponding row (used in PageRank, shortest path, etc). In contrast, MapReduce-like programs don't act on the global structure of data which results in inefficiencies: Pregel

observes that MapReduce has to pass the entire state of the graph between steps [10]. Ideally, after scaling R, programmers will still be able to address and manipulate distributed arrays. This goal forces us to ask: what kind of memory management and runtime support is required for scaling an array-based language like R?

Sparse datasets. Most real-world datasets are sparse. For example, the Netflix prize dataset is a matrix with 480K users (rows) and 17K movies (cols) but only 100 million of the total possible 8 billion ratings are available. Similarly, very few of the total edges are present in Web graphs. It is important to store and manipulate such data as sparse matrices and retain only non-zero entries.

How do we efficiently handle distributed sparse arrays and assign tasks to process them? Without careful task assignment performance can suffer from load imbalance: certain tasks may process partitions containing many non-zero elements and end up slowing down the whole system. Load imbalance is not a problem in MapReduce based systems where mappers scan any of the equal-size data partitions but the algorithm pays the additional price of sorting to send the right data to the reducers. Supporting incremental updates is also challenging as array partitions which were previously sparse may become dense and vice-versa. How do we handle such imbalance in the data partitions and perform efficient scheduling and straggler mitigation?

Incremental processing. In incremental processing, if a programmer writes $y = f(x)$, then y is recomputed automatically whenever x changes. Incremental processing raises many challenging questions. Only a few portions of the input may change; hence only the affected parts of the algorithm should be re-executed. How do we express such partial computations without scanning the whole dataset? Since new data continuously enters the system how should we enforce that distributed algorithms run only on a consistent view of the data?

Storage. Our target environment is one where data resides in distributed storage and multiple parallel programs incrementally process the data. What are the storage requirements and interfaces for an array-based system? How do we maintain consistency between the in-memory data of the program and the data stored on disk?

3 Background on R

In this section we briefly review R and the syntax for arrays [1]. R uses interpreted conditional execution (`if`), loops (`for, while, repeat`), and commonly uses procedures written in C, C++ and FORTRAN for better performance. Line 1 in Figure 1 creates a 3×3 matrix. The argument `dim` specifies the shape of the matrix and the sequence $10:18$ is used to fill the matrix. One can refer to entire subarrays by omitting specific index values along a dimension. For example, in line 3 the first row

```
1: > A<-array(10:18,dim=c(3,3))       #3x3 matrix
2: > A
       [,1] [,2] [,3]
  [1,]   10   13   16
  [2,]   11   14   17
  [3,]   12   15   18
3: > A[1,]                            #First row
  [1] 10 13 16
4: > idx<-array(1:3,dim=c(3,2))       #Index vector
5: > idx
       [,1] [,2]
  [1,]    1    1
  [2,]    2    2
  [3,]    3    3
6: > A[idx]                           #Diagonal of A
  [1] 10 14 18
7: > A%*%idx                          #Matrix multiply
       [,1] [,2]
  [1,]   84   84
  [2,]   90   90
  [3,]   96   96
```

Figure 1: Example array use in R.

of the matrix is obtained by $A[1,]$, where the column is left blank to fetch the entire first row. Subsections of a matrix can be easily extracted using *index vectors*. Index vectors are an ordered vector of rows where each row can be the index to another array. To extract the diagonal of `A` we create an index vector `idx` in line 4 whose elements are $(1,1)$,$(2,2)$ and $(3,3)$. In line 6, `A[idx]` returns the diagonal elements of `A`. In a single machine environment, R has native support for matrix multiplication, linear equation solvers, matrix decomposition and others. For example, $\%*\%$ is an R operator for matrix multiplication (line 7).

4 Presto: New R abstractions

Presto extends R with new language extensions and a runtime to manage distributed execution. In addition to scalability, these extensions add parallel execution and incremental processing. As shown in Figure 2, programmers use these extensions to write a Presto program and then submit it to a *master* node. The runtime at the master node is in charge of the overall execution. It fault tolerantly executes the Presto program as distributed tasks across *worker* nodes.

4.1 Storage driver

The storage driver in Presto is used to read input data, handle incremental updates, and save output data. While Presto can be ported to different storage systems using a driver program, it is natural to use it with a distributed *table* store such as HBase or HP Vertica. Using a table store makes it easier to map the complete data or subsets of it to arrays. In this paper we will assume that HBase is the underlying storage layer. Multiple programs, including MapReduce, SQL queries, and Presto programs may execute on the same HBase data. The Presto storage driver exports an interface that allows programs to register callbacks on tables (similar to database triggers). Callbacks are needed to notify the Presto program when data enters the store or is modified during incremental processing.

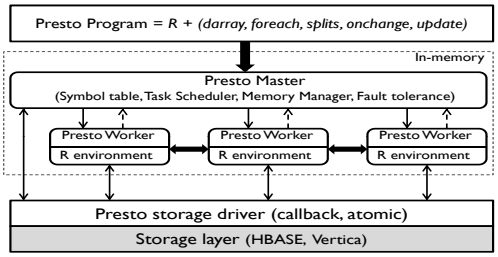

Figure 2: Presto architecture

The storage layer should support atomic multi-row updates which is available in databases and can be built on top of HBase [12].

4.2 Distributed arrays

Presto solves the problem of structure and scalability by introducing distributed arrays in R. Distributed arrays (`darray`) provide a shared, in-memory view of multi-dimensional data stored across multiple machines. Distributed arrays have the following characteristics:

Partitioned. Distributed arrays can be partitioned into chunks of rows, columns or blocks. Users can either specify the size of the partitions or let the runtime determine it. Locally, sparse partitions are stored in a compressed sparse column format. Partitions help programmers specify coarse-grained parallelism by assigning tasks to operate on partitions. Partitions can be referred to by the `splits` function.

Shared. Distributed arrays can be read-shared by multiple concurrent tasks. The runtime can leverage user hints to cache and co-locate data. Such hints reduce the overhead of remote copying during computation. Concurrent writes to array partitions are not allowed in the system. This is a conscious choice as Presto targets linear algebra operators which perform transformations from an input vector space to an output vector space. In such operations, each output element is calculated and written only once. Hence, Presto supports single-writer, multiple-reader consistency semantics.

Dynamic. Distributed arrays can be directly constructed from data in the storage layer. The Presto storage driver supports parallel loading of array partitions. If the array registered a callback on the storage table then whenever the data is changed the array will be notified and updated by the driver. Distributed arrays can also be created, re-sized and updated in the form of intermediate data during a computation. Thus, distributed arrays are dynamic; both the contents and the size of the distributed arrays can change as data is incrementally updated.

4.3 Distributed parallelism

Presto provides programmers with the `foreach` construct to execute deterministic functions in parallel. The Presto runtime creates tasks on worker nodes for parallel execution of the loop body. By default, there is an implicit barrier at the end of the loop to ensure that all parallel tasks finish before statements after the loop are executed. Programmers can set the loop parameter `wait` to false to remove the barrier and create tasks that continue to run without synchronizing.

4.4 Incremental computations

Presto introduces `onchange` and `update` to execute incremental algorithms on a consistent view of data. Programmers express dependencies by waiting on updates to distributed arrays. For example, `onchange(A){..}` implies that the embedded statements will be executed whenever array `A` is updated. `A` can also be a list of distributed arrays or just an array partition.

Programmers use the `update` construct to propagate changes to data. Programmers have the flexibility to determine what constitutes a change. For example, a PageRank calculation (Section 4.5) may batch multiple changes to the whole Web graph matrix, `M`, and then call `update(M)`, or it may absorb changes corresponding to only the top 100 Websites and call update on the sub-matrix, `update(M[v])`. In both cases the runtime will invoke the corresponding tasks that are waiting for the changes.

By calling `update`, programmers not only trigger the corresponding `onchange` tasks but also bind the tasks to the data that they should process. The `update` construct creates a version vector that succinctly describes the state of the array, including the versions of partitions that may be distributed across machines. This version vector is sent to all waiting tasks. Each task fetches the data corresponding to the version vector and, thus, executes on a programmer-defined, consistent view of data.

4.5 Example: PageRank

Figure 3 shows the code for incremental PageRank. The PageRank of a Web page measures its relative importance in the Web. The graph is represented as an adjacency matrix `M`. PageRank is the principal eigenvector of the matrix and is calculated in parallel (lines 5–12) using the power method [3]. In line 1, `M` is partitioned and loaded in parallel from the HBase table. The vector `pgr` is the initial PageRank vector and is partitioned similar to `M`. The PageRank calculation code is embedded inside the `onchange` clause in line 4. Therefore, whenever `M` is updated the changes are propagated to the PageRank task waiting on the `onchange` clause. For clarity we have simplified the code; the actual PageRank calculation occurs on the transition matrix and not the adjacency matrix. Therefore, changes to the adjacency matrix triggers recalculation of the transition matrix, which in turn causes re-computation of PageRank.

```
#Load data in parallel from adjacency matrix in HBase
1 : M<- darray(dim=c(NA,NA),blocks=ROW,drv='HBase')
2 : load(M, table='Web-graph')
3 : pgr<- darray(dim=c(ncol(M),1),blocks=shape(M),sparse=F)
#Calculate PageRank (pgr). Z is a vector of constants
4 : onchange (M){
5 :  repeat{
      #Distributed matrix operations
6 :   foreach(i, 1:numsplits(M),
              prFunc(p=splits(pgr,i), m=splits(M,i),
                     x=splits(xold), z=splits(Z,i)) {
7 :       p<-(m%*%x)+ z
8 :       update(p)
9 :    })
10:    if(norm(pgr-xold)>1e-9) break
11:    xold<-pgr
12:  }
13: }
```

Figure 3: Incremental PageRank on a dynamic Web graph.

5 Design and implementation sketch

The Presto master acts as the control thread for program execution. New tasks are created on workers whenever `foreach` loops are encountered in the program. The master keeps a map of the variables and their physical location which is used by workers to exchange information using pairwise communication. Presto reuses results from previous computations using task level memoization. For example, in the matrix multiplication $C = A \times B$, if only a few rows of A change then only the corresponding blocks in C will be re-calculated. We briefly describe other mechanisms used in Presto.

Dynamic partitioning. Partitioning a sparse matrix uniformly by rows or columns may lead to an uneven distribution of non-zero elements. Since it is important to distribute the computation evenly across tasks, the Presto runtime checks the size of partitions and if required, divides them further to reduce load imbalance. As the input for our programs are from a table in HBase, we reuse the HBase region boundaries to create the initial sparse matrix partitions. For iterative algorithms, we refine the partitions based on the execution of the first few iterations. Although repartitioning may involve copying data, it is expensive to calculate optimal partitions statically and static partitions do not help in cases where the data is incrementally updated. Dynamic partitioning also provides the flexibility to increase or decrease the amount of parallelism (tasks) in the program at runtime.

Versioning. Presto uses versioning to ensure correctness when arrays are updated across iterations or data is incrementally added from external sources. For example, write conflicts may arise if tasks read share an array which is also written to within that iteration. To avoid conflicts, each partition of a distributed array has a version. The version of a distributed array is a concatenation of the versions of its partitions, similar in spirit to vector clocks. Writes to array partitions create a new version of the partition. This version update ensures that concurrent readers of previous versions still have access to data. By versioning arrays Presto can safely execute iterative algorithms and multiple concurrent `onchange` tasks.

Co-location and caching. Presto workers execute functions which generally require multiple array partitions including remote ones. Presto uses two mechanisms to reduce communication overhead: partition co-location and caching. Partitions which are accessed and modified together in the same function are co-located on the same worker. Further, Presto automatically caches remote arrays partitions that are fetched during task execution. Workers use the version vector to make sure the cached arrays are valid and use the least recently used policy to evict older entries. Due to automatic caching, Presto does not need to provide explicit directives such as broadcast variables [16].

Handling dependences. Programmers can use the `onchange` clause to express dependence on multiple arrays. Presto resolves a multi-dependence as a logical conjunction. Statements embedded in `onchange` are invoked only when `update` has been called on *all* the arrays present in the dependence. Presto uses task queues to store tasks that will be invoked when data dependences have been satisfied. Task queues store a handle to functions embedded in the `onchange` clause. The `onchange` construct registers callbacks on distributed arrays that are specified in the clause. Whenever `update` is called on an array, Presto notifies the registered onchange task. The notification includes the version vector of the array which the onchange task should process. When all the dependences of an onchange task are satisfied, it is executed by the runtime.

Fault tolerance. Presto uses primary-backup replication to withstand failures of the master node. Only the meta-data information like the symbol table, program execution state, and worker information is replicated. When workers fail they are restarted and the corresponding functions are re-executed. Presto uses re-execution to transitively reconstruct lost partitions. Partitions are periodically made durable on HBase for faster recovery.

6 Evaluation

We have implemented six algorithms in Presto: matrix multiply, PageRank, Netflix recommendation [17], Twitter anomaly detection (calculates eigenvalues using iterative Lanczos) [7], vertex centrality in a graph, and Smith Waterman alignment used in gene sequencing. Each of these algorithms is implemented in fewer than 140 lines of code. Our comparison with Apache Mahout shows that Presto can be 20× faster than Hadoop. We are implementing other graph algorithms in Presto and plan to compare against recent systems like Spark.

Figure 4(a) compares the per-iteration performance of PageRank with that of Hadoop on the 100M node and

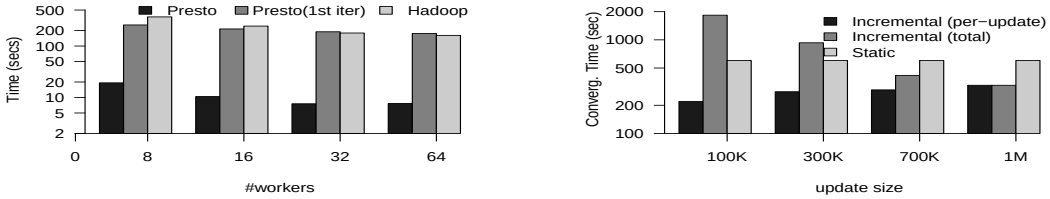

Figure 4: PageRank experiment (a) Comparison with Hadoop (b) Convergence time as the Web links are updated. Lower is better. Y-axis is log scale.

1.2B edge ClueWeb09[1] graph. With 64 workers, each iteration in Hadoop takes about 161 seconds while Presto takes only 8 seconds on average. The first iteration in Presto takes 175 seconds because it incurs the overhead of disk loading and has data imbalance; the first partition has one-sixth of the total data.

Figure 4(b) shows the convergence time when $1M$ Web links (0.1% of the Web graph) are updated in a day. We compare the time to incrementally compute PageRank as the number of updates vary versus running one calculation from scratch at the end of a day (*Static*). The plot shows that convergence time increases with number of updates; from 220 seconds for $100K$ updates to 327 seconds for $1M$ updates. Processing each update is considerably faster than performing the *Static* computation, which takes 602 seconds. However, incremental processing means computations occur more often: if the batch size is $100K$ then 10 re-computations are triggered and the total processing time (the middle bar) exceeds that of *Static* computation. Thus, there is a tradeoff between freshness of results and overall resource usage.

7 Related work

Programming models such as MapReduce, DryadLINQ, and others do not export a global view of data. They are inefficient to express linear algebraic formulations, and don't support incremental processing [5, 6, 16, 11]. Piccolo is closest to Presto in terms of programming model as it is based on partitioned tables but uses a key-value interface instead of matrices and does not focus on linear algebra [14]. Piccolo currently does not support incremental processing. Parallel MATLAB and certain extensions of R execute distributed array programs but don't handle faults, load imbalance or incremental processing. MadLINQ [15] provides a platform on Dryad specifically for matrix computations. Similar to Presto, MadLINQ can reuse existing matrix libraries on local partitions, is fault tolerant and distributed. However MadLINQ does not efficiently handle sparse datasets or support dynamic partitioning.

Incoop uses memoization for incremental computations but works only for MapReduce jobs thus inheriting its inefficiencies [2]. DryadInc supports a limited form of incremental processing and does not handle dynamic data or arbitrary task dependences [13]. Percolator [12] is Google's incremental processing system that applies multi-row transactional updates to Bigtable [4]. Presto can reuse Percolator's storage abstractions such as atomic multi-row updates and notifications for HBase and HP Vertica. Unlike Percolator, Presto also provides the language abstractions and mechanisms to handle fine-grained incremental processing such as dynamic partitioning, consistent updates, and memory management.

[1] http://lemurproject.org/clueweb09.php

8 Conclusion

This paper advocates the use of array-based languages, such as R, for large-scale machine learning and graph processing. We list the challenges of using R and propose abstractions to extend it for such computations.

Acknowledgements. We thank the anonymous reviewers whose comments helped us improve this paper.

References

[1] The R project for statistical computing. http://www.r-project.org.

[2] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proc. SOCC'11*, Cascais, Portugal, 2011. ACM.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the seventh international conference on World Wide Web*, WWW7, pages 107–117, 1998.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of OSDI 2006*, pages 205–218, Nov. 2006.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, pages 59–72, 2007.

[7] U. Kang, B. Meeder, and C. Faloutsos. Spectral Analy-

sis for Billion-Scale Graphs: Discoveries and Implementation. In *PAKDD (2)*, pages 13–25, 2011.

[8] R. Kannan. Algorithms: Recent highlights and challenges. In *Proc. of ISCA'11*, San Jose, USA, 2011. ACM.

[9] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Fundamentals of Algorithms. SIAM, 2011.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD '10*, pages 135–146, 2010.

[11] D. G. Murray and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *NSDI '11*, Boston, MA, USA, 2011.

[12] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI '10*, Vancouver, BC, Canada, 2010.

[13] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *HotCloud'09*, San Diego, California, 2009.

[14] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI '10*, Vancouver, BC, Canada, 2010. USENIX Association.

[15] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 197–210, Bern, Switzerland, 2012.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*, Boston, MA, 2010.

[17] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM '08*, pages 337–348, Shanghai, China, 2008.