Presto: Complex and Continuous Analytics with Distributed Arrays

Shivaram Venkataraman; Indrajit Roy; Robert S. Schreiber; Alvin AuYoung

HP Laboratories
HPL-2011-198

Keyword(s):

R; Distributed execution engine; Continuous processing

Abstract:

Presto is a distributed programming model for continuously analyzing data. Continuous analytics, where applications constantly refine their predictive models as data arrives, is useful in many applications such as user recommendation systems, link analysis, and financial modeling. Unlike batch processing, continuous analytics requires partial re-execution, low-latency turnaround, and transitive propagation of changes to dependent tasks. Current distributed systems like MapReduce~\cite{mapreduce+dean} and its enhancements lack general purpose support for continuous analytics. Presto extends the freely available R software with language primitives for scalability, distributed parallelism and continuous analytics. Presto constructs, 'darray' and 'onchange', are used to express parts of algorithms that should be executed when data items change. Even though the data is dynamic, Presto ensures that algorithms see a consistent snapshot of the data. Our experiments on four applications show that Presto is an order of magnitude faster than Hadoop while providing the added feature of continuous processing.

Internal Posting Date: October 6, 2011 [Fulltext] - HP Restricted

Presto: Complex and Continuous Analytics with Distributed Arrays

Shivaram Venkataraman* Indrajit Roy Robert S. Schreiber Alvin AuYoung
UC Berkeley *HP Labs*
shivaram@eecs.berkeley.edu {*indrajitr, rob.schreiber, alvin.auyoung*}*@hp.com*

Abstract

Presto is a distributed programming model for continuously analyzing data. Continuous analytics, where applications constantly refine their predictive models as data arrives, is useful in many applications such as user recommendation systems, link analysis, and financial modeling. Unlike batch processing, continuous analytics requires partial re-execution, low-latency turnaround, and transitive propagation of changes to dependent tasks. Current distributed systems like MapReduce [13] and its enhancements lack general purpose support for continuous analytics.

Presto extends the freely available R [3] software with language primitives for scalability, distributed parallelism and continuous analytics. Presto constructs, `darray` and `onchange`, are used to express parts of algorithms that should be executed when data items change. Even though the data is dynamic, Presto ensures that algorithms see a consistent snapshot of the data. Our experiments on four applications show that Presto is an order of magnitude faster than Hadoop while providing the added feature of continuous processing.

1 Introduction

We address the problem of complex analysis of large amounts of continuously changing data. In continuous analytics if a programmer writes $y = f(x)$, then y is recomputed automatically whenever x changes. Continuous analytics allows programmers to write spreadsheet-like computations; updates to data trigger automatic recalculation of only those parts that transitively depend on the modified data.

Continuous analytics is important in both business and scientific computing. User recommendations 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. These applications have three characteristics. First, they analyze large amounts of data– from ratings of millions of users to processing links for billions of Web pages. Second, they continuously refine their mathematical models by analyzing newly arriving data (Netflix [31], Twitter [16], financial options pricing [11], astronomy [17]). Third, they implement complex algorithms– matrix decomposition, eigenvalue calculation, PageRank– on data that is incrementally appended or updated.

Continuous analytics implies that processing is *always on*: results calculated and models refined with low latency. Continuous analytics imposes additional challenges compared to simply scaling analytics to a cluster and processing terabytes of data. First, only a few portions of the input data may change; hence only the affected parts of the algorithm should be re-executed. It is hard to express such partial computations in current systems. Existing systems, such as MapReduce [13] and Dryad [15], are primarily batch processing systems that scan and compute on the whole data. Second, since the data is dynamic it is difficult to express and enforce that distributed algorithms must run on a consistent view of the data. Finally, we want to provide programming primitives that support continuous analytics without exposing low-level programming details like callbacks or message passing.

Research in analytics software has focused on either handling the complexity of analysis or scaling the infrastructure to handle very large data. Statistical tools such as R and MATLAB provide a high-level programming model in which it is easy to express complex algorithms, but these tools do not scale to large datasets or handle continuous analytics. Data parallel models like MapReduce support scalable, fault tolerant computations but make it cumbersome to write complex algorithms. Previous work has shown that MapReduce is ill-suited and even inefficient for applications that require low-latency, iterative computations or dynamic task generation [30, 8, 20]. Recent data-parallel systems (Piccolo [25], Ciel [20]) ameliorate many of these problems but do not support continuous analytics and force users to learn a new programming model. Google's Percolator [22] supports incremental processing but is tightly coupled to Bigtable [9], is specialized for Web indexing, and does not have a general programming interface.

We present Presto, a distributed system that extends R to run continuous in-memory analytics. Most complex analytics are transformations on multi-dimensional arrays, hence Presto supports a variety of analytics by providing a scalable solution to distributed array operations. For example, PageRank and anomaly detection calculate eigenvectors of large matrices [7, 16], recommendation systems implement matrix decomposition [31], even genome sequencing and financial applications primarily

*Work done during internship at HP Labs

involve array manipulation. Presto builds on R since it is a popular programming model for array manipulation, and many of these algorithms have already been written in R, albeit for small datasets. With over $2,000$ open source add-on packages, R is one of the most widely used tools in academia and industry.

While R has numerous packages for complex analytics, it is primarily used as a single threaded, single machine installation. R supports neither scalability, nor distributed parallelism, nor continuous analytics. Presto solves all three challenges by extending R and implementing the infrastructure needed to remove these limitations.

For transparent scalability, Presto provides the abstraction of distributed arrays, `darray`. Distributed arrays store data across multiple machines and give programmers the flexibility to partition data in different ways, e.g., by rows, columns or blocks. Programmers write analytics code treating the `darray` as a normal array, without worrying that it is mapped to different physical machines.

For continuous analytics, we need a safe way to handle dynamic updates and to propagate the updates to different parts of the algorithm. To handle dynamic updates, distributed arrays in Presto are inherently versioned. Presto introduces the lexically scoped construct `onchange` to express data dependence and `update` to control when array values are propagated to dependent tasks. Programmers have the flexibility to choose the granularity of data on which partial computations occur. By combining `onchange` and `update`, programmers can express distributed R algorithms that run on a consistent view of dynamic data.

The Presto runtime executes R programs on a master node and several worker nodes, and is resilient to node failures. We have implemented several algorithms in Presto including PageRank, Netflix recommender system, Twitter anomaly detection and DNA sequencing. We compare Presto with Hadoop to show that analytics written using R are easy to write, can scale and have good performance. Our results are promising: for each of these applications, Presto is more than $10\times$ faster than Hadoop.

The main contributions of this paper are:

- Design and implementation of Presto, the first system to extend R to run on a cluster and to support continuous analytics. Presto maintains the brevity of R programs. Users modify only a modest amount of lines to run their R code on a cluster.
- The introduction of language primitives–`darray`, `onchange`, and `update`– that make it easy for programmers to express continuous analytics.
- Evaluation of five applications on real world data, where we show that Presto is significantly faster than the state of the art.

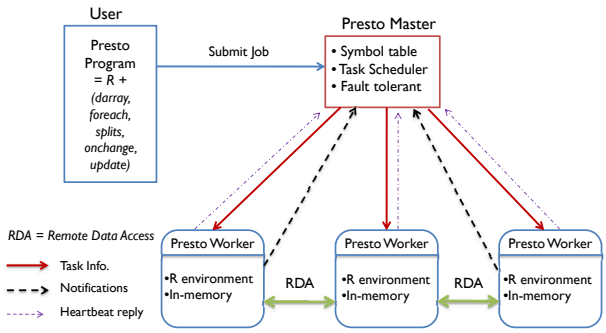

Figure 1: Presto architecture

2 Programming Model

Presto is R with the language extensions shown in Table 1. The extensions add data distribution, parallel execution and handling of dependencies in dynamic data. As shown in Figure 1, a *user* uses these extensions to write a Presto program and then submits it to a *master* node. The runtime at the master node is in charge of the overall execution. It runs the Presto program in a distributed manner across *worker* nodes and ensures fault tolerant execution. In this section we explore the extensions to R.

2.1 Background

R [3] is a freely available implementation of the S programming language [6]. R is widely used for statistical computations such as linear and nonlinear modeling, clustering, and classification. R has interpreted conditional execution (`if`), loops (`for`, `while`, `repeat`), and commonly uses procedures written in C, C++ and FORTRAN for better performance. Here we provide details on how arrays are handled in R.

R's compact notation for numeric and array operations makes it convenient for writing complex analytics. Numeric sequences can be easily generated by specifying their limits. Line 1 in Figure 2 shows how $1:5$ generates the sequence $(1,..,5)$. Line 2 shows how a 3×3 matrix can be created. The argument `dim` specifies the shape of the matrix. R uses the sequence $1:3$ to fill the matrix, repeating values since the sequence has fewer elements. For notational convenience one can refer to the entire subarray by omitting specific index values along a dimension. For example in line 4 the first row 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 in-

Functionality	Description
`darray(dim=, blocks=)`	Creates a distributed array. The array has dimensions specified by `dim`, and is partitioned by blocks of size `blocks`.
`splits(A,i)`	Returns the index vector for the i^{th} partition of the distributed array `A`. When i is not specified then a list of all the index vectors are returned.
`foreach(v, A, wait=){F(..)}`	Execute function `F` for each element v of the array A. Each iteration is executed in a process of its own. Implicit barrier at the end of the loop unless `wait` is set to false. `F()` is side-effect free.
`update(A)`	Creates a handle for the current version of `A`. Notifies and passes the handle to tasks that subscribed to changes in `A`.
`onchange(A, always=){}`	Embedded statements are executed whenever `A` is updated. Statements are triggered after each change unless `always` is set to false. `A` can be list of arrays.

Table 1: New programming language constructs in Presto

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

Figure 2: Example array use in R.

dex to another array. For example, to extract the diagonal of `A` we create an index vector `idx` in line 6 whose elements are $(1,1)$,$(2,2)$ and $(3,3)$. In line 8, `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. In line 9 we use $\%*\%$ to multiply two matrices `A` and `idx`.

2.2 Scalability with distributed arrays

R is primarily run on a single machine. It does not scale. The first challenge is to provide a scalable solution to handle large data.

Presto introduces the construct `darray` to support distributed arrays in R. Distributed arrays provide a shared memory view of multi-dimensional data stored across multiple machines. They are a natural extension to how R programmers use arrays in a single machine. We explain distributed arrays via the matrix multiplication example, `C=A*B`, in Figure 3. In lines $1-3$ we declare three new distributed arrays of dimensions $N \times N$. Similar to the existing R arrays, the `dim` argument specifies the dimension of the distributed array. The argument `blocks` is used to specify how the array is partitioned. A *partition* is a subarray, all of whose elements are guaranteed to be placed on the same machine. Distributed arrays can be partitioned into chunks of rows, columns or blocks. Partitions help programmers specify coarse grained parallelism. For example, in an embarrassingly parallel algorithm a programmer may break an array into N partitions that are automatically placed on N different machines, and each partition is processed by an independent task. In the code example, we partition `A` by rows, `B` by columns and `C` into $s \times s$ blocks. To simplify the programming model, placement of partitions on machines is hidden from the programmer. Depending upon machine availability, the runtime may locate array partitions on one or more machines. The number of machines on which the Presto program will execute is a configuration option.

Apart from scaling arrays, we also need a convenient way to refer to array partitions. Our solution is to reuse R's concept of *index vectors*. Presto provides a `splits` function that returns index vectors for each array partition in the distributed array. For array `A`, `splits(A,i)` returns the index vector for the i^{th} partition of `A` while `splits(A)` returns a list of index vectors for all partitions. Partitions are numbered in a block-cyclic column order. Each index vector can be used to extract the corresponding partition of `A`. For example, in line 5 of Figure 3, `splits(A,i)` returns the i^{th} index vector and `A[splits(A,i)]` refers to the i^{th} partition of `A`.

R functions, like row or column concatenation, that

```
#Partitioning:- A:rows, B:columns, C:blocks
1 : A<- darray(dim=c(N,N), blocks=(s,))
2 : B<- darray(dim=c(N,N), blocks=(,s))
3 : C<- darray(dim=c(N,N), blocks=(s,s))

#Populate array partitions in parallel
4 : foreach(i, 1:length( splits(A))) {
5 :   A[ splits(A,i)]<-read.table(FileA[i])
6 :   B[ splits(B,i)]<-read.table(FileB[i])
   }
#Distributed multiply
7 : num <- N/s
8 : foreach(i, 1:length( splits(C))) {
9 :   C[ splits(C,i)]<-A[ splits(A,((i-1)%%num)+1)]
                    %*%B[ splits(B, ((i-1)/num)+1)]
   }
10: print(C)
```

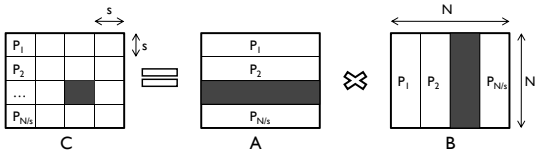

Figure 3: Distributed matrix multiply in Presto. New programming primitives are in bold. Array indices start from 1. Illustration shows the partitions, P_i.

act on arrays also work on distributed arrays. Operations such as creating and copying distributed arrays occur atomically. Distributed arrays also have version numbers which are used to run continuous analytics. The version numbers are, however, not visible to users. We describe versioning in more detail in Section 3.

2.3 Distributed parallelism

The second scalability challenge is to run R computations across machines. Presto provides programmers with the `foreach` construct to express functions that run in parallel. The function must be pure, i.e., side-effect free, and the output must depend on only the input arguments. Pure functions are easy to parallelize without the need for synchronization. The `foreach` construct is used to iterate over elements and is similar to parallel loops found in languages like FORTRAN 95. The Presto runtime creates tasks on worker nodes for parallel execution of the loop body. Unlike sequential loops, the order of execution of iterations is not defined in `foreach`. Programmers have to ensure that loop iterations are independent of each other. We will show later how the `onchange` construct can be used to expose dependences across loops. 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 parameter `wait` to false to remove the barrier.

By combining `darray` and `foreach`, programmers can express distributed *data-parallel* and *task-parallel* programs without writing low level code for message passing. In the data-parallel case each iteration of the loop acts on separate partitions of distributed arrays. The runtime tries to schedule tasks on workers such that there is location affinity for data. Loading of arrays in the matrix multiply code (lines $4-6$) is an example of data parallelism. In task-parallel programs, same or different tasks running on each worker may read and write to multiple distributed arrays. Remote portions of the distributed arrays are first copied locally before the task is executed. In the next section, we will describe the PageRank example (Figure 4) where one task continuously ingests the updates to the graph while other concurrent tasks fetch portions of the graph and refine its PageRank.

Concurrent tasks interact only at the start or after the task ends, otherwise executing in isolation. Writes to distributed arrays are buffered and not visible to other workers unless the task ends. Any write by the worker task to a subarray (array) creates a new version of the subarray (array). The runtime automatically associates an array variable with its latest version.

Revisiting matrix multiply. Let us revisit the matrix multiply example in Figure 3. As already mentioned, in lines $1-3$, matrices `A`, `B`, and `C` are created and partitioned by rows, columns and blocks respectively. Therefore, `A` and `B` have $\frac{N}{s}$ partitions while `C` has $(\frac{N}{s})^2$ partitions. In lines $4-6$ the partitions are populated in parallel. The parallel multiplication occurs in line 8 and 9, where a task is started to calculate each block of `C` by multiplying the corresponding rows in `A` and columns in `B`. Different iterations may perform remote fetches of partitions of `A` and `B` in parallel. Since there is an implicit barrier at the end of the `foreach` loop, all blocks of `C` would be calculated before the runtime executes the print statement in line 10. At line 10, the Presto runtime automatically fetches all the partition s of `C` from worker nodes and displays `C` on the terminal of the master node.

2.4 Continuous analytics

The third challenge that we solve is to provide support for continuous analytics. Continuous analytics requires two features. First, a programming construct to specify reactions to events. Second, reactions to events should result in mutations on consistent views of dynamic data.

Presto introduces `onchange` and `update` to run algorithms that continuously execute on a consistent view of dynamic data. In continuous analytics, only parts of an algorithm may need to execute in reaction to events. Events can be *external*, like data entering the system (PageRank, Figure 4) or *algorithmic*, where one task depends on the other (sequence alignment, Figure 5). In Presto, both types of event dependences are expressed as data flows using the `onchange` language construct.

In Presto, programmers express dependences by wait-

ing on updates to distributed arrays. For example, `onchange(A){...}` implies that the embedded statements would be executed whenever the array `A` is updated. `A` can also be a list of distributed arrays or just a sub-part of a larger array. Thus distributed arrays not only hold data, they also have meta-data about tasks that should be triggered when changes occur.

Using the `update` construct a programmer can propagate information when a change occurs. Programmers have the flexibility to determine what constitutes a change. For example, a PageRank calculation may batch multiple changes to the whole Web graph matrix and then call `update(adjMX)`, or it may absorb changes corresponding to only the top 20 Websites and call update on the submatrix, `update(adjMX[vec])`. In both cases the runtime will invoke the corresponding tasks that are waiting for the changes. Invocations adhere to the superset relation. Tasks that are waiting for changes to the whole matrix, `onchange(adjMX)`, would also be notified even if only a sub-part of the matrix changed, e.g., when `update(adjMX[vec])` occurs.

By calling `update` the programmer not only triggers the corresponding `onchange` tasks but also binds 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 the waiting tasks. Each task fetches the input data corresponding to the version vector. Therefore, each task executes on a programmer-defined, consistent view of data.

Presto's dependence mechanisms are different from *condition variables* that are used in multi-threaded programs. In a distributed setting, dependences require mechanisms to handle events occurring on multiple remote machines. While condition variables only provide event notification, Presto's dependence mechanisms also act as a wrapper for the snapshot of data that should be processed.

PageRank. Figure 4 shows the code to calculate PageRank of a dynamic Web graph. The PageRank of a Web page measures its relative importance in the Web graph. The graph is represented as an adjacency matrix `adjMX`. PageRank is the principal eigenvector of the matrix and is calculated in line $8-12$ using the power method [7]. The code in lines $2-5$ creates a worker process that periodically reads the modified graph and then calls `update` on `adjMX`. The PageRank calculation code is embedded inside the `onchange` clause in line 7. Therefore, every time `update(adjMX)` is executed, the current version vector is propagated to the PageRank task waiting on the `onchange` clause.

To simplify the presentation we have hidden two details. First, in line 9 we imply a local matrix multiplication using the $\%*\%$ operator. The actual implementation uses a `foreach` loop to perform distributed matrix multiply and addition, similar to the example in Figure 3. Second, PageRank calculation occurs on the transition matrix and not the adjacency matrix. In the implementation, changes to the adjacency matrix triggers recalculation of the transition matrix, which in turn causes recomputation of PageRank.

```
1 : adjMX<- darray(dim=c(N,N), blocks=(s,))
#Start a continuous task to update adj. matrix
2 : foreach(j,[1],wait=false){
3 :    while(TRUE){
       #Periodically read & publish modified graph
        ...
4 :    update(adjMX)
    }
5 : }

#Dependent task to calculate PageRank vector
#w and z are two vectors of constants
6 : xold<-w * (1/N)
7 : onchange (adjMX){
8 :  repeat{
      #Matrix operations
9 :   pgr<-(adjMX %*% xold)+(w %*% z)
10:   if(norm(pgr-xold)>1e-9) break
11:   xold<-pgr
12:  }
13:  print(pgr)
14: }
```

Figure 4: Calculating PageRank of a dynamic Web graph using the power method.

Sequence alignment. Figure 5 shows an implementation of the Smith-Waterman sequence alignment algorithm [26]. Unlike the PageRank example, this code shows how `onchange` can be used to express algorithmic dependences across parallel `foreach` iterations. Sequence alignment is solved using dynamic programming; each element $\mathtt{A}_{i,j}$ depends on three other elements $\mathtt{A}_{i,j-1}$, $\mathtt{A}_{i-1,j}$ and $\mathtt{A}_{i-1,j-1}$. After the calculation, the optimal alignment can be constructed by starting at the highest value cell in the matrix and then tracing backward. In line 1, `A` is partitioned into $s \times s$ blocks. In line 4 parallel worker tasks are created to process each block. However, each task depends upon the completion of three other tasks before it can proceed. This dependence is made explicit in line 5 via `onchange`. After processing a chunk the corresponding task calls `update` in line 7 to propagate the new values to waiting tasks. The illustration in Figure 5 shows the intra-block and inter-block dependences. After $\mathtt{A}_1$ is processed, tasks to process $\mathtt{A}_2$ and $\mathtt{A}_{m+1}$ can start. The worker task for block $\mathtt{A}_{m+2}$ has to wait for $\mathtt{A}_2$ and $\mathtt{A}_{m+1}$ to finish processing and call `update` before it can start execution.

```
1 : A<- darray(dim=c(N,N), blocks=(s,s))
2 : m<-N/s
3 : chunks<- splits(A)
4 : foreach(i, 1:length(chunks)) {
5 :   onchange (A[chunks[i-1]],A[chunks[i-m]],
                              A[chunks[i-m-1]]){
        #Invoke alignment function
6 :     A[chunks[i]]<-align(A[chunks[i-1]],
            A[chunks[i-m]],A[chunks[i-m-1]])
7 :     update(A[chunks[i]])
8 :   }
9 : }
```

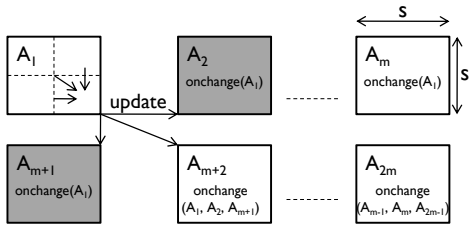

Figure 5: Dynamic programming solution to sequence alignment using the Smith-Waterman algorithm. Boundary conditions in the code are not shown.

2.5 Synchronization

There are no explicit synchronization primitives like locks or transactions in Presto. Programmers can use `onchange` to express dependences, arrays have implicit versions and `foreach` loops have barriers at the end. While we could implement distributed synchronization [5, 22], the absence of explicit synchronization is a conscious choice. Programs that require synchronization for isolation, i.e., concurrent reads and writes may occur on shared data, are hard to express correctly and difficult to parallelize. Unlike other languages, R was specifically developed for statistical computing. Most R programs act on arrays, have a very regular communication pattern, and do not require irregular synchronization. These programs expose dependence structure and do not need isolation if the data is static. Therefore, we have focused on adding primitives for dependence– `onchange` and `update`– and versioning to handle dynamic data. In Presto, data races may occur for incorrectly written programs. Data races between tasks do not cause data corruption since arrays are versioned and writes by tasks go to new versions of the array. In future one may use static analysis or runtime techniques to detect races.

3 System Details

The Presto runtime executes the analytics program across a master and multiple worker processes (Figure 1). The master takes part in both program execution and in overseeing worker progress even in the presence of failures. Both the master and the workers have full access to the R environment. In this section we describe the system support to run Presto programs.

3.1 Program execution

Program execution starts at the master. The master acts as the control thread for the program execution. New tasks are created whenever `foreach` loops are encountered in the program. The master schedules these tasks on workers. The master keeps a *symbol table* that maps the variable names to the actual data locations. For distributed arrays the symbol table contains meta-data about the number of partitions and the location of each partition across the machines. The master attempts to place tasks on workers close to the data that the task requires as input.

Workers execute tasks in parallel. The master gives workers symbol information about the input parameters of the task. Workers use the symbol information to fetch the required data before executing a task. Workers exchange data with each other using pairwise communication instead of going through the master. The division of work between the master and worker is hidden from the user. For example, in matrix multiply (Figure 3) each block of the resulting matrix `C` is calculated in parallel at the workers (lines 4-6). The workers fetch the corresponding rows and columns of the distributed arrays `A` and `B` before performing the local multiplication. However, after the calculation `C` is printed on the master side even though blocks of `C` are distributed across machines. The runtime automatically fetches all blocks of `C` at the master without any explicit annotation from the programmer, thus hiding the complexities of distributed programming.

3.2 Supporting language semantics

To support continuous analytics Presto implicitly handles versioning and notifications.

Versioning. 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 occur on a new version of the partition. This ensures that concurrent readers of previous versions still have access to data. For example, the adjacency matrix `adjMX` in the PageRank code (Figure 4) starts with version $\langle 0, 0, .., 0 \rangle$. In line $1-3$ the matrix may change. It is possibly that only Web page links present in the first partition of `adjMX` change. A new version of the first partition will become available, and the new matrix version becomes $\langle 1, 0, .., 0 \rangle$. The versions and data of unaffected partitions remain the same.

By versioning arrays Presto can safely execute multiple concurrent `onchange` tasks. In the PageRank example, the first `onchange` task may still be refining the PageRank of version $\langle 0, 0, .., 0 \rangle$ of the adjacency matrix when the second `onchange` task is triggered on version $\langle 1, 0, .., 0 \rangle$.

Previous versions of each partition are garbage col-

lected only when they are not being used by any task. The latest version of each partition exists unless the whole array is garbage collected. Presto uses reference counting for garbage collecting arrays.

Dependence notifications. Presto uses task queues to store tasks that will be invoked when data dependences have been satisfied. Task queues store the handle to functions embedded in the `onchange` clause. These functions can execute on either the master or the workers. 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.

In the limit, tasks can wait for changes to individual elements of the array. Update notifications are propagated in a hierarchical fashion to all listeners. For example, consider three tasks waiting for changes to a matrix M using `onchange`. Assume M is partitioned into four quadrants with initial version $\langle 0,0,0,0\rangle$. Task T_1 waits for changes to the first element of M, T_2 to the first quadrant, and T_3 to the whole matrix. If the first element is updated then all three tasks are notified, but if the second quadrant is updated then only T_3 will be notified. Since arrays are versioned by partitions, if only the first cell changes, then the notifications will include $\langle 1,0,0,0\rangle$ which is the new version of M.

In Presto it is possible that update to an array occurs before the onchange handler for that array is registered. This asymmetry may occur because of asynchrony in the distributed system or because of program logic, e.g the onchange construct is inside a conditional statement. To minimize chances of lost updates, Presto saves a history of update notifications with the array. The history is bounded in size, N_h, and its elements are garbage collected in FIFO order. All `onchange` handlers are guaranteed to receive N_h of the most recent updates destined for the handler.

Multiple 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. As in the sequence alignment case (Figure 5), Presto allows `update` notifications that may emanate from tasks on different machines.

Presto does not support logical disjunction of dependences, e.g. `onchange(A|B)`, to keep the programming model and the implementation simple. Otherwise,

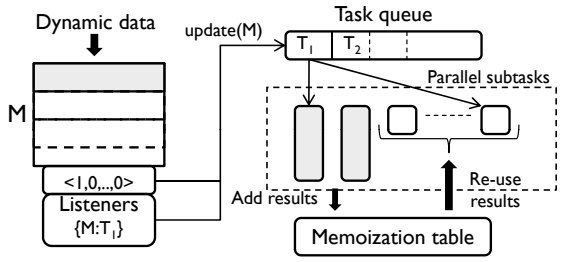

Figure 6: Example control flow in a Presto program. Shaded portions show partitions and tasks affected by new data.

due to distributed processing if a task is notified of versions A_0, B_0 and A_1, then the programmer's intent is not clear whether `onchange` should be invoked on array A_0 or B_0 or (A_0, B_0) or (A_1, B_0). By supporting only logical conjunction, Presto will invoke the task on (A_0, B_0) and wait for B_1 before invoking the next task on (A_1, B_1). This programming model reduces nondeterminism and makes it easier to reason about the interactions in the code. Additionally, failure recovery becomes simpler since we do not need to checkpoint what notifications are consumed at the worker while resolving a dependence.

Memoization. Presto reuses results from previous computations. In continuous analytics input data may change only partially. In the matrix multiply $C = A \times B$, if only a few rows of A change then only the corresponding blocks in C would be affected. It is wasteful to perform the complete multiplication if the changes are a small fraction of a very large matrix. Presto supports memoization at the granularity of tasks. The runtime stores a map of each task invocation and input versions to the corresponding outputs of that task. Before each task invocation, the map is checked to see if any entry matches the task and the inputs. On a match the output reference is substituted from the entry. As we explain in the next section, the memoization table is also helpful in handling failures.

Example. Figure 6 shows the control flow for the first iteration of the PageRank program. The changes in the Web graph affect only the first partition of `adjMX`, and its version number becomes $\langle 1,0,...,0\rangle$. Task T_1 listens to the changes to the whole matrix. When `update` is called, T_1 is notified of the version vector of `adjMX` that it should process. T_1 starts many parallel worker tasks, some of which multiply `M` with a vector to update the PageRank. Since only the first partition of `M` has changed, many of the worker tasks simply reuse the results from the memoization table.

3.3 Fault tolerance

Presto can withstand machine failures during the computation. We use primary-backup replication for the master. Only the meta-data information like the symbol table, program execution state, and worker information is replicated. The state of the master is reliably updated at the backup before a statement of the program is considered complete. R programs are generally a couple of hundred lines of code, but most lines perform a compute intensive task. The overhead of checkpointing the master state after each statement is low compared to the time spent to execute the statement.

The master sends periodic heartbeat messages to determine the progress of worker nodes. When workers fail they are restarted. Similar to systems like MapReduce and Dryad we assume that tasks are deterministic, hence we do not checkpoint worker state. In Presto each worker receives the versioned input data, executes a deterministic function on the input, and then writes the output value to versioned arrays. The output value and version is completely determined by the versioned input data and the function. We assume that the function execution on workers is idempotent, which is true if the functions are deterministic and do not trigger external events like output to the screen. Thus, we only need to guarantee that workers have *execute at least once* semantics; if a worker fails it is restarted with the appropriate input data. Stronger guarantees like *execute exactly once* semantics will reduce the current restriction of idempotent functions but would require checkpointing worker state including some of the messages. However, our experience with applications like PageRank, recommendation systems, and anomaly detection shows that *execute at least* once semantics is sufficient for a wide variety of analytics.

At worker restart, the runtime needs to fetch the input data for the task. If the input data is stored on a failed worker then, recursively, the corresponding versions of the input data are recreated. The information on how to recreate the input is present in the memoization table which keeps track of what input data versions and functions result in specific output versions. The master oversees the recovery of data through task re-execution. We assume reliable storage for the first version of the input data and all externally generated updates. In practice, arrays should periodically be made durable for faster failure recovery.

4 Implementation

Presto is implemented as an R add-on package. Packages are a simple way to extend R and have flexibility similar to browser plugins. One may dynamically load a package whenever needed. After installing the Presto package, programmers can use the new language constructs like `darray`, `foreach` and `onchange`. The current prototype has been tested on R-2.13.0.

In Presto, distributed array is a class in R whose objects can be passed to the `splits` function. Similarly, `foreach`, `onchange` and `update` are functions written in R. Dense matrices are stored using the native R matrix format. Sparse matrices are stored in the compressed sparse column format using the R Matrix[1] library. Our current prototype supports a limited set of operators, load, save, and concatenation for distributed arrays. We plan to implement all array operations for distributed arrays. The Presto package contains 300 lines of R code to support the new language constructs.

The Presto package is backed by our C++ infrastructure for communication between the master and workers. We use the publicly available Rcpp[2] library to interface between R and C++ code. Presto master and workers are Apache Thrift servers [4]. Both the master and the worker use thread pools for efficient RPC dispatch. Control messages, like starting the loop body in a worker, are serialized and sent using Thrift's RPC protocol. Data transfers, which primarily involve copying remote arrays, occur directly via TCP connections. Using the RInside[3] package we embed R instances inside the Presto master and worker. Therefore, both the master and workers have access to the R environment and can call R functions from C++. We wrote approximately $5,000$ lines of code for the C++ infrastructure.

Execution of Presto programs start at the master. The master makes RPC calls to send workers the information on what functions to execute and where the inputs are located. The symbol table of objects and their locations is a C++ map present at the master. The master also makes explicit RPC calls to workers to garbage collect arrays.

5 Evaluation

Presto allows programmers to express a variety of analytics that are either difficult or inefficient to write in current systems. Our goal is to evaluate Presto using three metrics:

Performance and scalability. How does Presto compare to existing systems like Hadoop?

Expressiveness. What programs are easier to express in Presto? How much effort does it take to convert sequential R programs to run on a cluster?

Benefits of continuous analytics. What are the performance benefits? Does accuracy of predictive models increase?

Table 2 summarizes the five applications and the datasets that we use to answer these questions. Four are

[1] http://cran.r-project.org/web/packages/Matrix/index.html
[2] http://cran.r-project.org/web/packages/Rcpp/index.html
[3] http://cran.r-project.org/web/packages/RInside/index.html

Application	Main algorithm	Input data	R LOC	Presto LOC
PageRank	Power method	nodes=100M, edges=1.2B	20	41
Twitter anomaly detection	Iterative Lanczos [16]	nodes=30M, edges=280M	65	132
Netflix recommender	Alternating-least-squares [31]	ratings=100M, movies=17K, users=480K	78	126
Sequence alignment	Smith Waterman	seq. length=20K	59	75
Matrix multiply	Multiplication	A=4.4M×16K, B=16K×87	5	7

Table 2: Characteristics of applications used for evaluation. We also note the lines of code (LOC) required to express the sequential version in R.

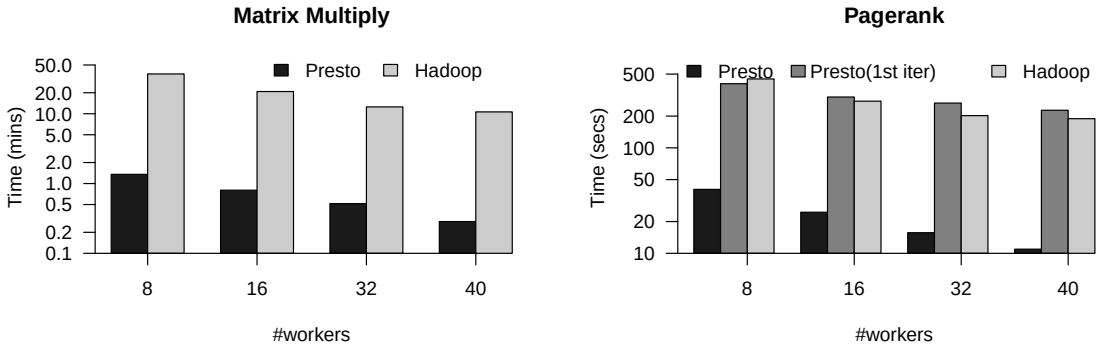

Figure 7: Execution time and scalability results for (a) Matrix multiply and (b) PageRank (per iteration). Lower is better. Y-axis is log scale.

real world applications that exhibit complex processing of large datasets and continuously refine their predictive models. The fifth application is matrix multiply. Even though matrix multiply is a micro-benchmark, it is a core component of many algorithms.

All experiments were performed using our local cluster consisting of 20 HP SL390 servers. Each server has two Intel Xeon X5650 processors and each processor consists of six 2.67GHz cores. The servers have 96GB of RAM and two 120GB SSD drives. The servers are connected to each other using a 10Gbps interface. All servers have Ubuntu 11.04 as the operating system with Linux kernel 2.6.38. We use the same number of Hadoop mappers as the number of Presto workers. We use Hadoop algorithms that are part of the Apache Mahout machine learning library [1]. Each iterative experiment was run at least 5 times and we present the average of the runs.

5.1 Performance and scalability

We compare the performance of Presto with Hadoop, which is an open source implementation of MapReduce. Our aim is to confirm that applications that are traditionally very easy to write in R can be scaled easily using Presto to get superior performance. Our results show that Presto is an order of magnitude faster than Hadoop for such applications.

Matrix multiply. Matrix multiply is used in many applications including graph algorithms [16]. It is also one of the steps in dimensionality reduction of huge sparse datasets. We compare the execution time of multiplying two large matrices in Presto with that of Hadoop, using input matrices of size $4.4M \times 16K$ and $16K \times 87$. These matrices were generated while performing singular value decomposition (SVD) of Apache Software Foundation (ASF) mail archives [1]. Presto code for distributed matrix multiply is shown in Figure 3. The plot in Figure 7(a) compares the performance of matrix multiply as we scale the number of workers. Execution time on Presto scales from 1.3 minutes with 8 workers to just 16 seconds with 40 workers. These numbers include the time to load the matrices from the solid-state disk and to save the results. In Hadoop the execution time decreases from 37 minutes to 10 minutes. Presto is $28\times - 37\times$ faster, with the performance gap increasing as the number of workers are increased. Presto performs better because of flexible data partitioning (block partitioning) and by avoiding the overhead of the MapReduce shuffle phase.

PageRank. The PageRank algorithm is widely used to determine the relative importance of Web pages. We use the iterative power method to calculate the PageRank [7]. Let M represent the transition probability matrix obtained from the Web graph. In the power method, at every iteration the PageRank vector is multiplied with the matrix M and normalized. Iterations end when a convergence criterion is met. Figure 4 shows the crux of Presto's PageRank code. We use a sample of the

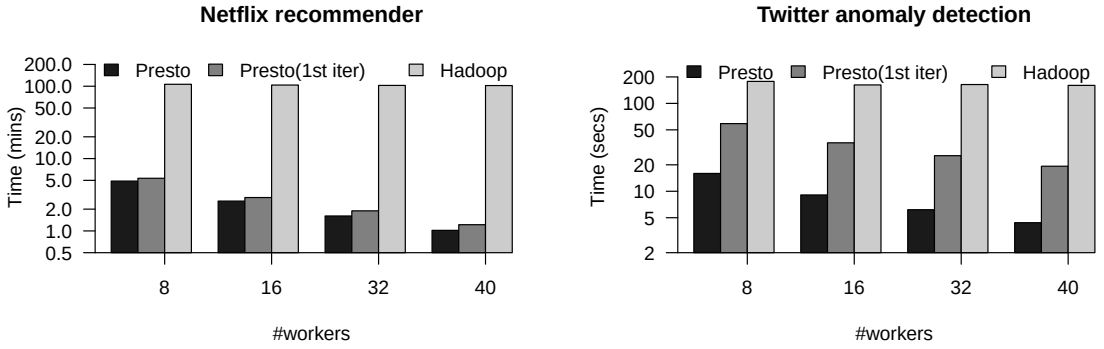

Figure 8: Per-iteration execution time and scalability results for (a) Netflix recommender and (b) Twitter anomaly detection. Lower is better. Y-axis is log scale.

ClueWeb09[4] graph with $100M$ nodes and $1.2B$ edges as our input.

Figure 7(b) compares the per-iteration performance of the PageRank in Presto with that of Mahout. In Mahout an iteration of PageRank takes 447 seconds with 8 mappers and reduces to 189 seconds with 40 mappers. The first iteration in Presto includes the load time and takes 406 seconds with 8 workers to 227 seconds with 40 workers. It is slow because the graph has an uneven edge distribution. The first partition has one-sixth of the total data and takes more time to load. Subsequent iterations are significantly faster and take from 40 seconds to 11 seconds as we increase the number of workers. In our experiments, it takes more than 52 iterations to converge. If we exclude the first iteration, then Presto is $11\times-17\times$ faster than Hadoop. [5]

Netflix recommender. The Netflix recommendation system uses aggregate ratings of a large number of users to suggest relevant movies. Similar recommendation systems are used by Amazon and Google to suggest different items. We use the dataset provided by the Netflix Prize competition which consists of $100M$ ratings given by $480K$ users on $17K$ movies [2]. The goal of the competition was to surpass Netflix's own prediction system (Cinematch) by 10%. We have implemented a parallel version of the alternating-least-squares (ALS) algorithm [31] in Presto and compare it with the same algorithm in Hadoop. ALS is an iterative algorithm based on factorizing the ratings matrix R, and approximating it with two rank-k matrices U and M. Since there are missing elements in R (unrated movies), ALS alternately refines U and M while minimizing the objective function. The algorithm ends when the improvement in the root mean squares error (RMSE) is less than a threshold.

In our experiments one iteration consists of one refinement for both U and M. Each refinement is a compute intensive task. For ALS, we set the number of features to 20 and λ to 0.065. Ideally, one should pick hundreds of features, but the Mahout code crashes if we choose more than 20. In comparison, we have successfully run ALS with more than 100 features on Presto. As shown in Figure 8(a), each Hadoop iteration takes approximately 106 minutes to complete with very little scaling as we increase the number of mappers. Scaling is limited because more than 85% percent of the work is performed by a single reducer. Using more number of reducers does not help as the reducer's task is not parallelized. Presto takes approximately 5 minutes with 8 workers to about a minute with 40 workers. The figure also shows the first iteration which is slightly slower because of initial data loading. The loading time is short and ranges from $12-27$ seconds. Thus, Presto is approximately $21\times-100\times$ faster that Hadoop for the recommendation algorithm.

Twitter anomaly detection. Many interesting insights about graphs can be obtained by looking at the connectivity of nodes. One useful metric is the number of triangles in a graph where a triangle consists of three nodes connected to each other. Kang et al. show that anomalous Twitter accounts have a very high ratio of triangle count versus node degree [16]. Anomalous accounts like adult sites in Twitter have low node degree but high number of connected, possibly fraudulent followers. Kang et al. also show that the number of triangles can be approximated by the top-K eigenvectors of the adjacency matrix. We implement the iterative Lanczos algorithm to calculate the top-K eigenvectors of large graphs.

Figure 8(b) compares the performance results of Presto and Hadoop on a Twitter graph with $30M$ nodes and $280M$ follower edges.We run 10 iterations in both

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

[5] We also experimented with Haloop [8] PageRank code which claims good iterative performance. Our initial results show that it converges slower than Mahout. We are in discussion with the Haloop authors and hence have not added results here.

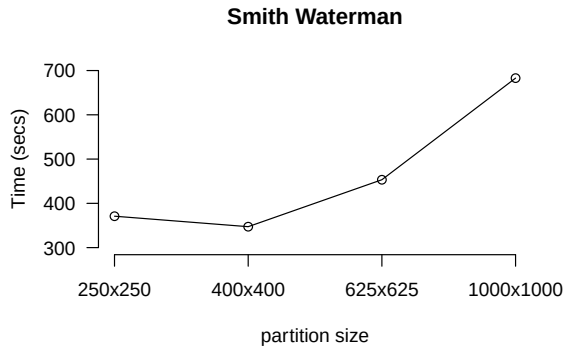

Figure 9: Effect of partition size on execution of sequence alignment. Lower is better.

the systems. Each iteration of Hadoop takes from 178 seconds to 160 seconds as we scale the number of mappers from 8 to 40. For Presto each iteration takes from 15 seconds to 4.5 seconds as we scale the number of workers. If we include the load time, which happens for the first iteration, then the execution time for the first iteration ranges from 58 seconds to 19 seconds. For subsequent iterations, Presto is $11\times - 36\times$ faster than Hadoop.

5.2 Expressiveness

Compared to existing systems, Presto has the advantage that it can easily express continuous analytics. Our experience also shows that many distributed computations, even without continuous analytics, are easier to express in Presto than Hadoop. People who are familiar with R can use Presto to run their existing programs in a cluster with minimal code changes. Since objective evaluation is difficult, we consider two examples.

Data dependence algorithms. The Smith Waterman algorithm (Figure 5) has applications in DNA sequence analysis and protein alignment. It is an example of dynamic programming which is hard to parallelize due to dependences–each element depends upon three neighboring elements. The program cannot be converted into efficient code in the MapReduce paradigm. If the matrix is partitioned by rows then there is no parallelism during execution. Rows will be computed sequentially in each MapReduce job. Even if we could partition the matrix in blocks, the lack of dependence primitives in MapReduce means one cannot trigger the runnable tasks without manually managing each mapper. In contrast the distributed alignment program can be expressed in 75 lines in Presto. Figure 9 shows the performance of Presto when aligning two $20K$ length sequences. We vary the size of the partitions, and execute it across 32 workers. Smaller partitions result in more tasks, lower per task execution time but higher communication overheads. The results show execution time increases both when partitions are too small or too large. In the experiment partitions of size 400×400 have better performance because this size balances the computation versus communication costs.

From R to Presto. For each of the five applications we wrote the sequential R code and then Presto code to run it in a cluster. Table 2 summarizes the number of lines of code for each application. The results show that in each application we had to add fewer than 70 lines to the original R code. Therefore, we expect that with modest changes many existing R applications can be run in Presto.

5.3 Benefits of continuous analytics

The main advantage of continuous analytics is that it reduces the latency at which processed information is available. In the PageRank application, newly added Web pages can be ranked and made part of search results more quickly. For Netflix recommendation, new movie ratings can be instantaneously processed to give better movie suggestions. In contrast, current batch processing systems resort to over-the-night re-computations.

In this section we show that (1) time to process each update is low and (2) continuous processing increases the accuracy of models but at the cost of increased overall execution time.

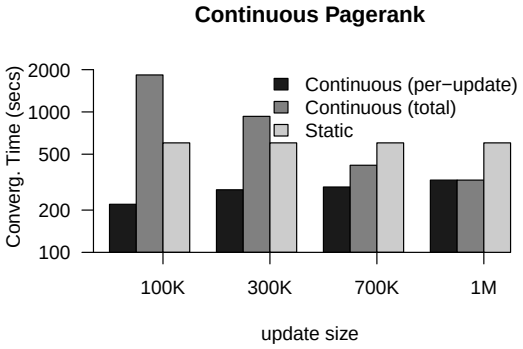

Figure 11: PageRank convergence time as the Web links are updated. Y-axis is log scale. Lower is better.

PageRank. Consider the case when $1M$ Web links (0.1% of the Web graph) are updated in a day and the PageRank has to be re-calculated for the new graph. We compare the time to update the PageRank continuously versus running a one-time calculation from scratch at the end of the day. Figure 11 shows the PageRank convergence time as we increase the update size, i.e., number of Web links in one update. The one-time calculation over the new graph, with all of $1M$ updates, is denoted as *Static*.

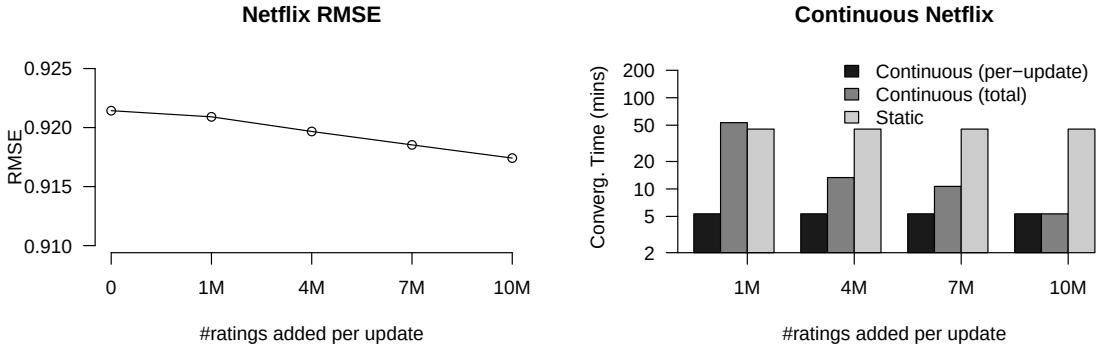

Figure 10: (a) Effect on prediction accuracy as new ratings are included (b) Netflix recommender convergence time as new ratings are added (Y-axis is log scale). Lower is better.

Convergence time increases with update size; from 220 seconds for $100K$ to 327 seconds for $1M$ update size. Processing each update is considerably faster than performing the *Static* computation from scratch, which takes 602 seconds. However, continuous analytics means computations occur more often. If the batch size is $100K$ then 10 re-computations would be triggered. In Figure 11 we also plot the total time spent in processing all $1M$ changes for each update size. The plot shows that for smaller update size, the total processing time exceeds that of *Static* computation. Our results point to the trade off that faster inclusion of new data triggers more continuous computations and may increase the overall resource usage.

Netflix. For the Netflix experiments we measure the effect on accuracy and performance as new movie ratings are added. We create a base dataset which consists of randomly chosen $90M$ of the total $100M$ Netflix Prize movie ratings. Remaining $10M$ movie ratings simulate newly arriving data. The ALS algorithm parameters consist of 50 features and $\lambda = 0.065$. Figure 10(a) shows that the accuracy of the prediction model increases (RMSE decreases) as we include new movie ratings. Thus, it is beneficial to continuously process movie ratings and update the recommender model.

In Figure 10(b) we measure the time taken for the recommender to converge as we increase the number of movie ratings in an update. Experiments show that for our dataset the per-update convergence time is approximately 5.3 minutes irrespective of the update size. In the plot, the *Static* version consists of computing the convergence on full $100M$ ratings and takes more than 45 minutes. Similar to the PageRank experiment we see that frequent computations increase the aggregate time spent to process all the updates. Thus, continuous analytics improves the recommender accuracy and decreases latency but at the cost of increased aggregate execution time.

6 Related Work

Many programming models and distributed infrastructure have been proposed for large scale data analytics. We contrast Presto with some of the closely related systems.

Dataflow models. MapReduce and Dryad are popular dataflow systems for parallel data processing [13, 15]. To increase programmer productivity high-level programming models–Dryadlinq [29], PIG [21], and Sawzall [23]–were created on top of MapReduce and Dryad. While these systems scale to hundreds of machines, they are best suited for batch processing. Modifications to MapReduce, like HaLoop [8] and Twister [14], improve the performance of iterative computations by caching data across loop iterations. Spark, though not tied to MapReduce, uses similar caching mechanisms and can reconstruct lost data using selective re-execution [30]. None of these systems provide primitives to process consistent views of dynamic data. Any re-execution as part of continuous analytics has high latency, may scan the whole dataset, and wastes work.

Incremental processing. Incremental processing, which involves recomputing only on changed data, has been studied in programming languages community. Memoization and dependence graphs are two techniques used for processing incremental changes. DryadInc can support a limited form of incremental processing by using memoization and by merging results according to user defined functions [24]. It handles append-only data, and does not support dynamic data (e.g. link updates to Web graph in the PageRank calculation) or arbitrary task dependences.

Percolator [22] is Google's incremental processing system that applies multi-row transactional updates to Bigtable [9]. It has *observer* tasks that are triggered when changes occur in the Bigtable cells. Percolator's

observers and the Presto tasks that process `onchange` statements have similar functionality. Percolator does not specify language semantics for observers or notifications, thus tying the programming model tightly with Bigtable. In contrast, Presto primitives of `onchange` and `update` can be used to specify arbitrary dependence graphs; making it easier for programmers to write a variety of continuous programs. Percolator notifications trigger waiting tasks to process any data, hence programmers must use distributed transactions for concurrency control. In Presto the users specify the arrays that a waiting task should process. Notifications include the information about the inputs, hence concurrency control is implicit via versioning. Unlike Percolator, where users have to write new code, Presto programmers can reuse and compose hundreds of freely available R packages to write different algorithms.

Other data-parallel models. Piccolo runs parallel application that can share state using distributed, in-memory, key-value tables [25]. Piccolo runs *kernel* functions in parallel that manipulate table data and if necessary store it back in the table. These kernel functions are similar to the Presto worker tasks. In Piccolo, concurrent updates to a key are resolved via user defined accumulators. Piccolo does not provide arbitrary task dependences. In contrast, CIEL can dynamically start new data-dependent tasks as the job is progressing [20]. CIEL can support both iterative and recursive algorithms transparently. Neither Piccolo nor CIEL, support continuous analytics where partial computations occur on dynamic data.

Parallel languages. High performance computing use explicit message passing models like MPI. MPI programmers have the flexibility to optimize the messaging layer over a variety of network topologies. MPI programs are difficult to write and hard to maintain because of the explicit communication. An implementation of Presto may use MPI as the messaging layer for good performance, but such a layer should not be exposed to the end user.

New parallel programming languages like X10 [10] and Fortress [27] use the partitioned global address space model (PGAS). In this model data is logically partitioned in a global address space. Most of these languages provide a plethora of features like asynchrony, locality and atomicity. While continuous analytics could be expressed in these languages, the programmer would have to deal with low level primitives like synchronization and explicit locations. For example, X10 exposes constructs like *Place* so that programmers can specify on what processors computations should occur. None of these languages are as popular as R, and the users would have to rewrite hundreds of statistical algorithms that are already present in R.

Statistical tools. MATLAB provides a parallel computing toolbox to run single programs on multiple data. It has parallel for-loops and *codistributed* arrays. Codistributed arrays are arrays partitioned across multiple MATLAB instances. Unlike Presto, MATLAB functions embedded inside parallel for loops cannot access remote portions of a codistributed array. Parallel MATLAB is not fault tolerant, and does not support dependences or continuous analytics.

While there have been individual efforts in parallelizing R, to the best of our knowledge, Presto is the first system that provides the semantics of distributed arrays and parallel loops in R. There is no support for continuous analytics in existing R infrastructure. Recent work like Ricardo extends R to distributed analytics by integrating R with Hadoop [12]. Ricardo does not scale the R infrastructure, instead users execute programs as Hadoop jobs followed by R operations on the Hadoop results. R operations run on a single machine. Ricardo inherits the inefficiencies of Hadoop and has no support for continuous analytics.

Graph based models. Pregel and GraphLab use bulk synchronous processing (BSP [28]) to process data in parallel [19, 18]. In Pregel each vertex processes its local data and communicates with other vertices using messages. GraphLab supports safe data sharing by providing locks. Both require the user to re-write their program in the BSP model. GraphLab can support localized computation but it exposes locking semantics that are difficult to get right, and it cannot support dynamic data.

7 Conclusion

Presto's main contributuons are that it provides language semantics and an implementation to express continuous analytics, and it extends R to run in a cluster.

Even though Presto is a high performance computation layer, there are multiple ways in which it can be improved. Presto will benefit greatly by reliable distributed storage systems. Reliable distributed storage will simplify fault tolerance in Presto. Especially with reliable *in-memory* storage, checkpointing state will become cheaper and Presto will be able to support many non-deterministic programs. An implementation shortfall that we noticed is that sparse matrices with non-uniform data distribution can cause load imbalance. Certain workers may receive a dense partition which takes longer to compute than others. By supporting irregular partitioning such load imbalances could be decreased.

We believe that Presto makes it easier for users to transition to distributed programming using the familiar R

language. However, the transition for programmers does have a learning curve, arguably less steep than learning a new programming paradigm. Even though many R packages can be reused in Presto, most packages are not parallel or distributed. Presto programmers will have to reuse and implement distributed version of the algorithms using the new language constructs. For example, distributed matrix multiply in Presto involves data partitioning and parallel loops, but it can reuse the R's native matrix multiplication for each local task.

R is the programming language of choice for many users to write statistical machine learning algorithms. R is concise. Thousands of open source packages for algorithms and data visualization are available. Due to the lack of scalability, programmers must rewrite the same algorithms in the Hadoop or Dryad framework. With Presto, our aim is to retain the benefits of R, add new functionality, and reduce the need to rewrite code. Our experience with the case studies in this paper show that Presto code remains concise and is significantly faster than Hadoop.

References

[1] Apache mahout. http://mahout.apache.org.

[2] Netflix prize. http://www.netflixprize.com.

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

[4] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, April 2007.

[5] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP '07*, pages 159–174, 2007.

[6] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language*. Chapman & Hall, London, 1988.

[7] 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 7*, WWW7, pages 107–117, 1998.

[8] Y. Bu, B. Howe, M. Balazinska, and M. e. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[9] 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.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*, pages 519–538, 2005.

[11] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, September 1979.

[12] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD Conference'10*, pages 987–998, 2010.

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

[14] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC '10*, pages 810–818, 2010.

[15] 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.

[16] U. Kang, B. Meeder, and C. Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In *PAKDD (2)*, pages 13–25, 2011.

[17] S. Loebman, D. Nunley, Y. Kwon, B. Howe, M. Balazinska, and J. P. Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *Cluster Computing*, pages 1–10, 2009.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *CoRR*, pages 1–1, 2010.

[19] 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.

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

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD'08*, pages 1099–1110, 2008.

[22] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI '10*, pages 1–15, 2010.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13:277–298, October 2005.

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

[25] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI '10*, pages 1–14, Vancouver, BC, Canada, 2010. USENIX Association.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, Mar. 1981.

[27] G. L. Steele, Jr. Parallel programming and code selection in fortress. In *PPoPP '06*, pages 1–1, 2006.

[28] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[29] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, pages 1–14, 2008.

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

[31] 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.