Supporting Bulk Synchronous Parallelism in Map-Reduce Queries

Leonidas Fegaras

University of Texas at Arlington, CSE
Arlington, TX 76019
fegaras@cse.uta.edu

***Abstract*—One of the major drawbacks of the Map-Reduce (MR) model is that, to simplify reliability and fault tolerance, it does not preserve data in memory across consecutive MR jobs: a MR job must dump its data to the distributed file system before they can be read by the next MR job. This restriction imposes a high overhead to complex MR workflows and graph algorithms, such as PageRank, which require repetitive MR jobs. The Bulk Synchronous Parallelism (BSP) programming model, on the other hand, has been recently advocated as an alternative to the MR model that does not suffer from this restriction, and, under certain circumstances, allows complex repetitive algorithms to run entirely in the collective memory of a cluster. We present a framework for translating complex declarative queries for scientific and graph data analysis applications to both MR and BSP evaluation plans, leaving the choice to be made at run-time based on the available resources. If the resources are sufficient, the query will be evaluated entirely in memory based on the BSP model, otherwise, the same query will be evaluated based on the MR model.**

I. Introduction

Recently, the Map-Reduce (MR) programming model [7] has been emerged as a popular framework for large-scale data analysis on the cloud. In particular, Hadoop [14], the most prevalent implementation of this framework, has been used extensively by many companies on a very large scale. Currently, the majority of these MR jobs are specified in declarative query languages, such as HiveQL [30] and PigLatin [22]. One of the major drawbacks of the MR model is that, to simplify reliability and fault tolerance, it does not preserve data in memory between the map and reduce tasks of a MR job or across consecutive MR jobs. Consequently, using the original implementation of the MR model, to pass data to the next job in a MR workflow, a MR job must dump its data to the distributed file system (DFS), to be read by the next MR job. This restriction imposes a high overhead to complex MR workflows and graph algorithms, such as PageRank, which require repetitive MR jobs. The Bulk Synchronous Parallelism (BSP) programming model [32], on the other hand, which precedes the MR model by more than a decade, has been recently advocated as an alternative to the MR model that does not suffer from this restriction, and, under certain circumstances, allows complex repetitive algorithms to run entirely in the collective memory of a cluster.

A BSP computation consists of a sequence of supersteps. Each superstep is evaluated in parallel by every peer participating in the BSP computation. A superstep consists of three stages: a local computation, a process communication, and a barrier synchronization. Each peer maintains a local state in memory, which is accessible by this peer only across all supersteps. In addition, during the local computation stage of a superstep, each peer has access to the messages sent by other peers during the previous superstep. It can also send messages to other peers during the process communication stage, to be read at the next superstep. The barrier synchronization stage synchronizes all peers to make sure that they have received all the sent messages before the next superstep. To cope with peer failures, each peer may use checkpoint recovery, to occasionally flush the volatile part of its state to the DFS, thus allowing to rollback to the last checkpoint when a failure occurs. In many graph analysis problems a large part of the state is used for storing the graph, which is immutable and does not require checkpointing. Therefore, the BSP model is better suited to graph analysis than the MR model, provided that the entire graph can fit in the collective memory of the cluster. Because of their importance and their repetitive nature, graph analysis programs are the most likely candidates to benefit from the BSP model, which explains why the most important BSP implementation to date is Google's Pregel [20], which has been used by Google as a replacement for the MR model for analyzing large graphs exclusively. In addition, Apache's Giraph [12] is an open-source alternative to Pregel and Apache's Hama [26] is a general BSP computation platform built on top of Hadoop that can process any kind of data.

When coding a parallel data analysis application in BSP, one must make sure that the local state of each peer does not exceed its memory capacity. This means that, since the graph structure is needed through the BSP computation, one has to use a sufficient number of peers to make sure that the graph partition processed by each peer can fit in its memory. That is, the entire graph, as well as the auxiliary data needed for processing the graph, must fit in the collective memory of the cluster. This imposes a minimum for the cluster size. Then, an important question is what happens if the cluster is not large enough to process a certain BSP application over a larger amount of data. One solution would be to rewrite the BSP code to process the graph in batches, but then this program would not be that different from a MR job, since it would have to dump the state to secondary storage before it would be able

to process the next batch from the graph. Furthermore, this change would require a major rewriting of the BSP code. We believe that, to be effective, a data analysis application should not depend on the cluster size. Instead, if the cluster is large enough, the code should be able to be evaluated entirely in memory in BSP mode. Otherwise, the same unchanged code should be able to be evaluated in MR mode. Supporting both modes can be done if the application is coded in a declarative query language, so that queries expressed in this language are translated into both a MR workflow and a BSP job, leaving the choice to be made at run-time based on the available resources.

Another reason for using a common declarative language for both MR and BSP computations is that, currently, most programmers prefer to use a higher-level query language for their MR applications, such as HiveQL, instead of coding them directly in an algorithmic language, such as Java. This, we believe, will also be the trend for BSP applications because, even though, in principle, the BSP model is very simple to understand, it is hard to develop, optimize, and maintain non-trivial BSP applications coded in a general-purpose programming language. Currently, there is no widely acceptable BSP query language. Existing MR query languages, such as HiveQL and PigLatin, provide a limited syntax for operating on data collections, in the form of relational joins and group-bys. Because of these limitations, these languages enable users to plug-in custom MR scripts into their queries for those jobs that cannot be declaratively coded in their query language. This nullifies the benefits of using a declarative query language and may result to suboptimal, error-prone, and hard-to-maintain code. More importantly, these languages are inappropriate for complex scientific applications and graph analysis, because they do not directly support iteration or recursion in declarative form and are not able to handle complex, nested scientific data, which are often semi-structured. But there are other query languages for distributed data processing, such as MRQL [10] and AQL [2], that are expressive enough to capture complex data analysis applications in declarative form.

In this paper, we give semantics to BSP computations and present translation rules from MR computations to BSP computations. These rules allow MR workflows, produced by compiling MR queries, to be transformed to a sequence of BSP jobs. Consecutive BSP jobs are fused into a single BSP job by chaining the supersteps from these jobs, thus yielding only one BSP job for each query. Our goal is to support both evaluation modes: if the input data and state fit in memory, then a query is evaluated in memory using BSP; otherwise, it is evaluated using MR. We have implemented our MR-to-BSP translations for MRQL ([10], [9]). MRQL is a novel query language for MR computations that is more powerful than existing query languages, such as Hive and PigLatin, since it can operate on more complex data, such as nested collections and trees, and it supports more powerful query constructs, thus eliminating the need for using explicit MR procedural code. Our MRQL system can store local data collections (bags) in three ways. If the bag is accessed only once (a property inferred statically by our type-inference system), then the bag elements are lazily accessed using stream iterators. Otherwise, the bag is materialized to a memory vector. When the vector size exceeds a threshold, it is spilled to a local file. Consequently, when memory is not sufficient, our BSP evaluations may spill data to local files, thus deteriorating performance.

Consider, for example, the following MRQL query that calculates the k-means clustering algorithm (Lloyd's algorithm), by deriving k new centroids from the old:

```
repeat centroids = ...
  step select < X: avg(s.X), Y: avg(s.Y) >
         from s in Points
       group by k: (select c from c in centroids
                    order by distance(c,s))[0]
  limit 10;
```

where Points is the input data set of points on a plane, centroids is the current set of centroids (k cluster centers), and distance is a function that calculates the distance between two points. The initial value of centroids (the ... value) can be a bag of k random points. The select-query in the group-by part assigns the closest centroid to a point s (where [0] returns the first tuple of an ordered list). The repeat step query clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the average values of its points. The step query repeats 10 times. Most other SQL-like query languages do not allow subqueries in group-by nor they allow arbitrary operations over groups and, therefore, have insufficient expressive power to capture any arbitrary MR computation. MRQL, on the other hand, has been proven to be MR-complete [10]. The MRQL query optimizer generates a single map-combine-reduce job for the select-query in the repeat-step, which is evaluated 10 times. More specifically, for each data point, the mapper of this job finds the closest centroid from the bag of the current centroids (which resides in memory) and uses it as the combine/reduce key. The mapper of each node uses an in-memory hash table to group the points by the key (the closest centroid) and partially aggregates the groups incrementally from the incoming points using the combine function. That is, the hash table has k entries, where each entry has 4 values, which are the partial counts and sums of avg(s.X) and avg(s.Y). Finally, the reducers perform the final reductions from the partial aggregated results shuffled from the nodes. Our system translates this query to a single BSP job, since the repeat-step query is translated to a BSP job, the repetition is done with a second BSP job, and the two BSP jobs are fused into one. That is, the resulting BSP translation reads the input points once and calculates the final centroids entirely in memory. We will show in Section X that the BSP evaluation of this query can be an order of magnitude faster than its MR evaluation.

There is a number of recent proposals for improving the execution of MR workflows and repetitive MR jobs, such as, HaLoop [3], Twister [8], and SystemML [13]. There also some proposals for cloud computing that use distributed memory instead of DFS for inter-node communication, such as, the main memory MR (M3R [28]), Spark [34], Piccolo [25], and

distributed GraphLab [19]. The closest approach to ours is the Shark [27] (Hive on Spark) sub-project of Spark [34], which evaluates SQL-like queries using in-memory evaluation. To the best of our knowledge, Shark does not provide a comprehensive query optimization framework yet, although there are plans for doing so in the future. Our work is also related to the Datalog query language used by the Asterix project [4], which is used as an intermediate language for translating and optimizing Pregel-like and iterative map-reduce-update specifications to Hyracks physical operations. None of these systems provide any query language, optimization technique, or query evaluation on top of an existing BSP platform.

In summary, the key contribution of this work is in the design of a framework for translating complex declarative queries for scientific and graph data analysis applications into both MR and BSP evaluation plans, leaving the choice to be made at run-time based on the available resources. If the resources are sufficient, the query will be evaluated entirely in memory based on the BSP model, otherwise, the same query will be evaluated based on the MR model. Leveraging on our earlier work on translating complex MRQL queries into MR workflows, this work makes the following contributions:

- We present a novel BSP algebra that captures the essence of most BSP computations. Contrary to some BSP implementations, this algebra separates BSP supersteps explicitly as functional arguments to a BSP operator, thus making easier to chain together supersteps from consecutive BSP jobs.
- We present transformation rules to map any workflow of MR jobs into a workflow of BSP operations, thus making possible to translate any MR query into a BSP workflow.
- We present rules for fusing cascading BSP jobs into a single BSP job, thus deriving a single BSP job for the entire query.
- We report on a prototype implementation of the BSP evaluation of MRQL queries on top of Apache Hama. We show the effectiveness of our method through experiments that compare the BSP to the MR evaluation plans for two analytical task queries: K-means clustering and PageRank.

The rest of this paper is organized as follows. Section II compares our approach with related work. Section III summarizes our earlier work on a MR algebra, which is used in the implementation of the MRQL language. Section IV describes our BSP algebra and its implementation on Apache's Hama. Section V gives the transformation rules from the MR algebra to the BSP algebra, thus making possible the translation of MR queries to BSP jobs. Section VI describes the normalization rules for fusing cascading BSP jobs into a single BSP job, thus deriving a single BSP job from a MR query. Section VII describes how MRQL physical plans are translated to BSP plans. Section VIII addresses total aggregation in BSP, which requires to summarize results across all peers. Section IX describes an optimization technique for merging consecutive supersteps within the same BSP job, thus reducing the number of supersteps. Finally, Section X presents experiments comparing the BSP to the MR evaluation plans, produced by MRQL for two analytical task queries: K-means clustering and PageRank.

II. Related Work

The map-reduce (MR) model was first introduced by Google in 2004 [7]. Several large organizations have implemented this model, including Apache Hadoop [33] and Pig [22], Apache/Facebook Hive [30], Google Sawzall [24], and Microsoft Dryad [16]. The most popular MR implementation is Hadoop [14], an open-source project developed by Apache, which is used today by Yahoo! and many other companies to perform data analysis. There are also a number of higher-level languages that make MR programming easier, such as HiveQL [30], PigLatin [22], SCOPE [5], and Dryad/Linq [17]. Hive ([30], [31]) is an open-source project by Facebook that provides a logical RDBMS environment on top of the MR engine, well-suited for data warehousing. Using its high-level query language, HiveQL, users can write declarative queries, which are optimized and translated into MR jobs that are executed using Hadoop. Yahoo!'s Pig [11] resembles Hive as it provides a user-friendly query-like language, called PigLatin [22], on top of MR, which allows explicit filtering, map, join, and group-by operations. SCOPE [5], an SQL-like scripting language for large-scale analysis, does not support sub-queries but provides syntax to simulate sub-queries using outer-joins.

The BSP model was introduced by Leslie G. Valiant in 1990 [32] and has been since improved and extended by many others (eg, [29]). The best known implementations of the BSP model for data analysis on the cloud are Google's Pregel [20] and Apache's Giraph [12] and Hama ([26], [15]). Although Pregel has already been used extensively by Google for large graph analysis, it has not reached the popularity of MR yet. Its open-source counterpart, Giraph, and the general BSP platform, Hama, are still in their very early stages of development and are not used in the same degree as Pregel. The BSP model is compared with the MR model in [23], which discusses how to map any MR computation to a BSP, and vice versa.

III. Background: The MR Algebra

In our earlier work ([10], [9]), we have presented an algebra for MR computations, which is summarized in this section (in a more simplified form). It consists of the following operators:

- source(file,tag): a tagged dataset from a DFS file.
- $S_1 \cup S_2$: the bag union of the datasets S_1 and S_2.
- mapReduce$(m,r)\ S$: a MR job.
- repeat$(f)\ S$: a repetition of MR jobs.

These algebraic operators can capture most nested relational operations, including equi-join, selection, projection, group-by with aggregation, unnesting, intersection, etc. They cannot capture general θ-joins and total aggregation. We will discuss total aggregation in Section VIII.

The operation source(file,tag) reads an input file from DFS, which can be a raw text or binary file, and breaks it into data fragments, based on some specification. Every fragment is annotated with an integer tag to identify the data source. The result is a dataset of type $\{ (\text{int}, \alpha) \}$, which is a bag of pairs, where each pair contains an integer tag and a fragment of type α.

The operation 'mapReduce$(m, r)\, S$' specifies a map-reduce job. It transforms a dataset S of type $\{\alpha\}$ into a dataset of type $\{\beta\}$ using a map function m and a reduce function r with types:

$$\begin{aligned} m: &\ \alpha \rightarrow \{ (\kappa, \gamma) \} \\ r: &\ (\kappa, \{\gamma\}) \rightarrow \{\beta\} \end{aligned}$$

for the arbitrary types α, β, γ, and κ. The map function m transforms values of type α from the input dataset into a bag of intermediate key/value pairs of type $\{(\kappa, \gamma)\}$. The reduce function r merges all intermediate pairs associated with the same key of type κ and produces a bag of values of type β, which are incorporated into the mapReduce result. The semantics of mapReduce can be given by the following equation:

$$\text{mapReduce}(m, r)\, S \ = \ \text{cmap}(r)\, (\text{groupBy}(\text{cmap}(m)\, S))$$

where cmap and groupBy are defined as follows: Given two arbitrary types α and β, the cmap$(f)\, s$ operation (also known as concat-map or flatten-map in functional programming languages) applies the function f of type $\alpha \rightarrow \{\beta\}$ to each element of the input bag s of type $\{\alpha\}$ and collects the results to form a bag of type $\{\beta\}$. Given two types κ and α, for an input bag of pairs s of type bag((κ, α)), the groupBy(s) operation groups the elements of s by the key of type κ to form a bag of type bag($(\kappa, \{\alpha\})$). For example, groupBy({ (1,"A"), (2,"B"), (1,"C") }) returns the bag { (1,{"A","C"}), (2,{"B"}) }. The implementation of a dataset in a MR platform, such as Hadoop [14], is the path name of a DFS file that contains the data. Then, mapReduce is a MR job that reads a DFS file as input and dumps the output into a DFS file. The union operation is a no-op since it simply concatenates the DFS path names.

In our earlier work [10], we have used an explicit join algebraic operation to capture several equi-join algorithms for the MR framework, such as reduce- and map-side joins ([18], [33]), as well as cross products and θ-joins using distributed block-nested loops. Here, to simplify our translations from MR to BSP, we have used only one equi-join algorithm, the reduce-side join, which can be expressed as a mapReduce operation over the union of the two join inputs. For example, the join

select X.C, Y.D **from** X, Y **where** X.A=Y.B

can be evaluated using:

```
mapReduce( λ(n,z). if n=1 then {(z.A,(n,z))} else {(z.B,(n,z))},
           λ(k,s). { (x.C,y.D) ‖ (1,x)∈s, (2,y)∈s } )
         ( source(X,1) ∪ source(Y,2) )
```

where an anonymous function $\lambda(x, y).e$ specifies a binary function f such that $f(x, y) = e$. In this join specification, the MR input is the combination of the two inputs X and Y, under different tags, 1 and 2 respectively. For each input tuple (either from X or Y), the MR mapper emits the join key along with the tuple. The reducer separates the X tuples from the Y tuples (based on their tag) and performs their cross product in memory (since they already match the join condition).

Finally, 'repeat$(f)\, S$' transforms a dataset S of type $\{\alpha\}$ to a dataset of type $\{\alpha\}$ by applying the function f of type $\{\alpha\} \rightarrow \{(\alpha, \text{bool})\}$ repeatedly, starting from S, until *all* returned boolean values are false. The implementation of repeat in Hadoop does not require any additional MR job (other than those embedded in the function f) as it uses a user-defined Java counter to count the true values resulting from the outermost physical operator in f. These counts are accumulated across all Hadoop tasks assigned to this outermost operator. The repeat operator repeats the f workflow until the counter becomes zero.

There are other operations required to capture SQL-like queries, which are all described in our earlier work [10]. Our earlier work also describes general methods for translating any MRQL query into a MR algebraic plan and for optimizing and translating this plan into a MR job workflow.

IV. The BSP Algebra

In this section, we present our BSP algebra that captures BSP computations. It has been implemented on Apache Hama. The domain of our BSP algebra is $\{(I, V)\}$, where I is the peer id type and V is a snapshot type. More specifically, the BSP domain is a map from I to V, which, for each peer participating in the BSP computation, returns its local snapshot of type V. This snapshot contains all the local data of the peer I that reside in its local memory. The BSP algebra consists of the following operators:

- source(file,tag): a tagged dataset from a DFS file.
- $S_1 \cup S_2$: the bag union of the datasets S_1 and S_2.
- bsp(tag, superstep, state) S: the BSP operation.

Note that, as it was the case for the MR algebra, this BSP algebra cannot capture total aggregation, which is addressed separately in Section VIII.

As it was for the MR algebra, the operation source(file,tag) reads an input file from DFS and breaks it into data fragments, based on some specification. Every fragment is annotated with an integer tag that identifies the data source. This operation returns the map $\{(I, \{(\text{int}, \alpha)\})\}$, which associates each peer I with a dataset of type $\{(\text{int}, \alpha)\}$, which is a bag of pairs, where each pair contains the tag and a fragment of type α. This dataset is stored locally in the snapshot of peer I.

The BSP operation 'bsp(tag, superstep, initstate) S' maps a dataset S of type $\{(I, \{(\text{int}, V)\})\}$ into a dataset of type $\{(I, \{(\text{int}, V)\})\}$, using arguments of the following types:

$$\begin{aligned} \text{tag}: &\ \text{int} \\ \text{superstep}: &\ (\{M\}, V, K) \rightarrow (\{(I, M)\}, V, K, \text{boolean}) \\ \text{initstate}: &\ K \end{aligned}$$

The tag is the output value tag. The superstep operation is performed by every peer participating in the BSP computation and can modify the peer's local snapshot V by returning a new

```
for I in peers do {
    terminate[I] = false;
    snapshot[I] = input[I];
    state[I] = initstate [I];
};
msgs = { };
do {
     new_msgs = { };
     exit = true;
     for I in peers do {
          (m,v,k,b) = superstep( { m || (i,m) ∈ msgs, i=I },
                                 snapshot[I], state[I] );
          snapshot[I] = v;
          state[I] = k;
          new_msgs = new_msgs ∪ m;
          terminate[I] = terminate[I] ∨ b;
          exit = exit ∧ b;
     };
     msgs = new_msgs;
} while (¬exit);
return snapshot;
```

Fig. 1. Sequential Evaluation of 'bsp(tag, superstep, initstate) input'

```
class IM { I peer; M message; }
class Result { Bag<IM> messages; Bag<V> snapshot;
               S state; Boolean exit; }

void bsp ( BSPPeer<K,V,K,V,M> peer ) {
    Bag<V> snapshot = readLocalSnapshot(input);
    S state = initstate ;
    Result result;
    Bag<M> msgs = new Bag<M>();
    M msg = null;
    boolean exit;
    do {
        result = superstep(msgs,snapshot,state);
        snapshot = result.snapshot;
        state = result.state;
        exit = synchronize(peer,result.exit);
        for ( IM m: result.messages )
            peer.send(m.peer,m.message);
        peer.sync();
        msgs = new Bag<M>();
        while ((msg = peer.getCurrentMessage()) != null)
            msgs.add(msg);
    } while (!exit);
    writeLocalSnapshot(tag,snapshot);
}
```

Fig. 2. Implementation of 'bsp(tag, superstep, initstate) input' on Hama

```
boolean synchronize ( BSPPeer<K,V,K,V,M> peer,
                      Boolean exit ) {
    if (exit == allTrue)
        return true;
    if (exit == allFalse)
        return false;
    if (!exit)
        for ( I i: peer.getAllPeerNames() )
            peer.send(i,"not ready");
    peer.sync();
    return peer.getNumCurrentMessages() == 0;
}
```

Fig. 3. Peer Synchronization

local snapshot V. Since we are going to use a single BSP job to capture a workflow of multiple, repetitive MR jobs, each superstep must be able to execute multiple operations, one for each MR job, and be able to switch between the jobs in the MR workflow. This is done using a finite state machine (DFA) embedded in the superstep logic, which controls the superstep evaluation, as we will show in Section V. More specifically, the DFA state is of type K, which may contain data particular to this state. Then, a superstep evaluation makes a transition from the current state K (in the superstep input) to a new state K (in the superstep output). The initial state is initstate. The call superstep(ms, v, k) takes a bag of messages ms of type $\{M\}$, the current snapshot v, and the current state k, and returns the triple (is, v', k', b), which contains the new messages is to be sent to the other peers (a bag of (I, M) pairs), the new snapshot v', the new state k', and a flag b. This flag indicates whether the peer wants to terminate this BSP computation ($b =$ true) or use barrier synchronization to continue by repeating the superstep under the new snapshot, state, and messages ($b =$ false). Note that, only if *all* peers agree to exit (when they all return $b =$ true), then every peer should exit the BSP computation. Otherwise, if there is at least one peer who wants to continue, then every peer must do barrier synchronization and repeat the superstep.

The meaning of the BSP operation, when simulated on a sequential machine, is given in Figure 1. In this simulation, each peer I, from the set of all peers, is associated with a snapshot snapshot[I], a DFA state state[I], and a flag terminate[I]. The result is stored in the map ResultState. At each step, the superstep operation is evaluated by every peer, even when the peer has requested to terminate. The input to the superstep consists of the messages sent to this peer during the previous step, along with the current snapshot and state. Its result contains the messages to be sent to other peers m, the new snapshot v, the new state k, and the termination flag b. The BSP operation repeats until all peers want to terminate.

Figure 2 gives the implementation of the BSP operation on Apache Hama [15]. As it was done for our MR implementation, the domain of the physical operations that correspond to our BSP algebraic operators is a DFS file. More specifically, each peer reads all those blocks from the input file that are local to the peer (ie, those that reside in its local disk) and stores them entirely in its memory. This is done with the function readLocalSnapshot. At the end of the BSP operation, the new state is dumped to DFS using the function writeLocalSnapshot. Although the input/output is done through DFS, the rest of the BSP computation is done entirely in memory (if we ignore checkpointing). We will see in Section IX that using the DFS for passing snapshots across BSP operations is not important because consecutive BSP operations are fused into one, thus requiring the use of DFS

only at the beginning and the end of a program. The classes IM and Result in Figure 2 are used to store a message sent to peers and a superstep result, respectively. The body of the Hama's bsp method is evaluated at each peer. The snapshot is the local snapshot of the peer, accessible only by this peer throughout its BSP computation. In Hama, there is no clear separation of supersteps. Instead, when a peer calls peer.sync(), it waits for the next barrier synchronization. That is, this call signals the end of a superstep. If the bsp method exits without a call to peer.sync(), the peer terminates. Hama requires that all peers exit at the same time. Before the new messages that are derived from a superstep are sent to peers, we need to check whether all peers agree to exit. This is done by calling the synchronize method, shown in Figure 3. The result.exit flag returned by the superstep is a Boolean object that can take four values: If it is == to the prespecified Boolean object allTrue, it indicates that all peers want to exit right now, so there is no need to poll the peers; if it is == to the prespecified Boolean object allFalse, it indicates that all peers need to continue; otherwise, if its value is equal to true/false, it indicates that this particular peer wants to exit/continue, but we still need to poll the other peers to check if they can all exit. In the latter case, if all peers agree to exit, then we exit. This is done in synchronize by having those peers that are not ready to exit send messages to other peers: if there is at least one message after sync, then at least one peer is not ready to exit. As we will see in Section V, in all but one case, peers agree to simultaneously exit or continue and this decision is hardwired in the logic of the superstep. This means that in most cases the synchronize method will immediately return true or false without having to poll the peers. The only case where polling is required is at the BSP superstep of a repeat operation that tests the termination condition of the loop (explained in Section V). In such a case, some peers may want to finish the loop earlier because they reach the termination condition sooner. Based on synchronize, these peers will continue their supersteps until all peers agree to terminate. The shortcut of using allTrue/allFalse eliminates unnecessary message exchanges and syncs in most cases.

V. MR-to-BSP Transformation

It has been noted in related work [23] that any MR job can be evaluated using a BSP job that has two supersteps: one for the map and one for the reduce task. In this section, we elaborate this idea by transforming our MR algebra into our BSP algebra. First, we transform a mapReduce operation:

mapReduce(m, r) S
= bsp(tag,
  $\lambda(ms, as, b)$.
    if b
    then (cmap($\lambda(k, c).\{$(shuffle(k),(k, c))$\}$)
      (cmap($\lambda(k, c).m(c)$) as),
      { }, false, allFalse)
    else ({ }, cmap(r) (groupBy(ms)), false, allTrue),
  true) S

where tag is a unique tag assigned to this operator. The BSP snapshot assigned to a peer is a bag of type $\{(\text{int}, \alpha)\}$, which contains the input partition local to the peer. The DFA state is a boolean flag b that indicates whether we are in map or reduce mode. When the superstep is called with $b =$ true, then it evaluates the map function m over each element in the input partition, as. Then, it shuffles the map results to the reducers, using the function shuffle(k) over the key k, which is returned by the mapper m. Shuffle must be a function that returns the same peer for the same key. An obvious implementation of shuffle(k) in Hama distributes the mapped values equally to the peers using hash partitioning:

peer.getPeerName(k.hashValue() % peer.getNumPeers())

The first superstep returns the new state $b =$ false, which directs bsp to perform the reduce stage over the received messages, which contain the shuffled data. Then, it simply groups the data by the key and applies the reduce function (based on the mapReduce semantics in terms of cmap and groupBy, given in Section III). Note that, the cmap and groupBy operations are evaluated entirely in memory. Note also that the exit condition of the map step is allFalse, while that of the reduce step is allTrue. They indicate that all peers agree to continue after map and exit after reduce.

The repetition repeat(f) S is translated to BSP by first translating the MR repetition step function f into a BSP repetition F by recursively applying the MR-to-BSP transformation rules. Then, the translation of f should take the form:

$F(x)$ = bsp$(tag, s, k0)$ x

for some tag tag, superstep function s, and initstate $k0$, since all MR-to-BSP transformations return a bsp operation. Then, repeat(f) S is mapped to BSP as follows:

repeat(f) S
= bsp(tag,
  $\lambda(ms, vs, k)$.let $(ts, bs, k, b) \leftarrow s(ms, vs, k)$
    in if b
      then ({ },
        cmap($\lambda(t, (v, b)).\{(t, v)\}$) bs,
        $k0$,
        $\forall(t, (v, b)) \in bs : \neg b$)
      else (ts, bs, k, allFalse),
  $k0$) S

The state of this BSP is the same as the state of the repeat step, which is initially $k0$. This BSP evaluates the superstep function of the repeat step, s, multiple times. When the s call returns $b =$ false, we are in the middle of evaluating a superstep s, so we must return allFalse to continue until we finish evaluating one complete repeat step. When $b =$ true, we have finished a single repeat step and, if the termination condition is false, then we should proceed to the next repeat iteration starting from scratch, with state equal to $k0$. Otherwise, we should exit. The termination condition is checked by evaluating $\forall(t, (v, b)) \in bs : \neg b$, given that the dataset returned by a repeat step contains the source tag t, the value v, and the flag b, which is true if v satisfies the termination condition. This termination condition is the only case where a value not == to allFalse/allTrue is returned. As explained in Section IV, this is handled in the BSP call by polling all peers to check the condition across all peers.

VI. BSP Normalization

As we have seen in Section V, any MR workflow that consists of MR algebraic forms can be mapped to a sequence of cascading bsp operations. Unfortunately, each bsp operation reads its input state from DFS and dumps its final state to DFS, which makes the resulting BSP workflow plan not that different from the original MR workflow in terms of wasting resources. Our goal is to translate any such plan into a single BSP job that uses the DFS to read the initial input and dump the final answer only, while performing the rest of the computation entirely in memory. More specifically, we will give a constructive proof to the following theorem by providing a normalization rule for fusing BSP operations in pairs.

Theorem 1: Any algebraic term in the BSP algebra can be normalized into a term with just one BSP job that takes the following form:

$$\mathsf{bsp}(t, s, k)\ (\mathsf{source}(f_1, t_1) \cup \cdots \cup \mathsf{source}(f_n, t_n))$$

for some t, s, k, f_i, and t_i.
Consequently, since any MR query can be translated to a MR algebraic term (from our previous work [10]), and since any MR algebraic operator can be translated into a BSP algebraic operator (Section V), this theorem indicates that any MR query can be translated into a single BSP operation.

Proof: The normalization is done by recursively applying a single rule that fuses two cascading bsp operations into one. Essentially, it chains together the supersteps of the two cascading bsp operations into one superstep, which uses a flag to remember which superstep is evaluating each time. The bsp fusion law is the following:

$$\begin{array}{l}
\mathsf{bsp}(t_2, s_2, k_2)\ (\ S_1 \cup \cdots \cup S_{i-1} \\
\qquad\qquad \cup\ \mathsf{bsp}(t_1, s_1, k_1)(S'_1 \cup \cdots \cup S'_m) \\
\qquad\qquad \cup\ S_{i+1} \cup \ldots \cup S_n\ ) \\
= \mathsf{bsp}(\ t_2, \\
\qquad \lambda(ms, as, (c, k)). \\
\qquad\quad \mathsf{if}\ c \\
\qquad\quad \mathsf{then\ let}\ (ts, bs, k', b) \leftarrow s_1(ms, as, k), \\
\qquad\qquad\quad \mathsf{exit} \leftarrow \mathsf{synchronize}(b) \\
\qquad\qquad \mathsf{in}\ (\ ts, bs, \\
\qquad\qquad\quad (\ \neg\mathsf{exit}, \mathsf{if\ exit\ then}\ k_2\ \mathsf{else}\ k'\ ), \\
\qquad\qquad\quad \mathsf{allFalse}\ ) \\
\qquad\quad \mathsf{else\ let}\ (ts, bs, k', b) \leftarrow s_2(ms, as, k) \\
\qquad\qquad \mathsf{in}\ (\ ts, bs, (\ \mathsf{false}, k'\ ), b\ ), \\
\qquad (\mathsf{true}, k_1)\ ) \\
\quad (\ S_1 \cup \cdots \cup S_{i-1} \cup S'_1 \cup \cdots \cup S'_m \cup S_{i+1} \cup \ldots \cup S_n\ )
\end{array}$$

The state of the resulting bsp is extended with a flag c that identifies which of the two supersteps (s_1 or s_2) to evaluate each time. While c is true, the first superstep s_1 is evaluated, until it returns $b = \mathsf{true}$, in which case it synchronizes with the other peers to check if they all agree to exit from step s_1. If they do, exit becomes true and the BSP computation switches to $c = \mathsf{false}$ mode. Then, while c is false, the second superstep s_2 is evaluated until it terminates. The call to synchronize is necessary since the inner BSP computation may correspond to a repeat expression, which may not terminate at the same time for all peers. In all other cases, synchronize will immediately return true or false without having to poll the peers. ■

VII. Translating MRQL Plans to BSP Plans

In Section V, we presented transformation rules from the MR algebra to the BSP algebra that make possible the translation of MR queries to BSP jobs. The MRQL query evaluation system though provides many specialized MR physical plans that implement the MR algebraic operator. For example, the Map physical operator implements the MR operation without a reduce function. This can be trivially implemented using a BSP operation with only one superstep. In this section, we show how two important MR physical plans used in MRQL are mapped to BSP plans: the mapCombineReduce and the reduce-side join, mapReduce2.

As noted in the k-means example in Section I, a map-combine-reduce operation applies a combine function to the data generated at each map task, thus partially reducing the data at the map-side before they are shuffled and shipped to the reducers. This technique, which is generally known as an in-mapper combiner, is very effective when the reducer performs aggregations only. The $\mathsf{mapCombineReduce}(m, c, r)\ S$ operation, in addition to the map m and reduce r functions, provides a combine function c. Then, this operation is mapped to BSP as follows:

$$\begin{array}{l}
\mathsf{mapCombineReduce}(m, c, r)\ S \\
= \mathsf{bsp}(\ \mathsf{tag}, \\
\qquad \lambda(ms, as, b). \\
\qquad\quad \mathsf{if}\ b \\
\qquad\quad \mathsf{then}\ (\ \mathsf{cmap}(\lambda(k, s).\, \mathsf{cmap}(\lambda x.\, \{(\mathsf{shuffle}(k), (k, x))\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (c(k, s))) \\
\qquad\qquad\qquad\quad (\mathsf{groupBy}(\mathsf{cmap}(\lambda(k, x).\, m(x))\ as)), \\
\qquad\qquad\quad \{\ \}, \mathsf{false}, \mathsf{allFalse}\ ) \\
\qquad\quad \mathsf{else}\ (\ \{\ \}, \mathsf{cmap}(r)\ (\mathsf{groupBy}(ms)), \mathsf{false}, \mathsf{allTrue}\ ), \\
\qquad \mathsf{true}\ )\ S
\end{array}$$

that is, the mapper groups its results by the shuffle key and partially reduces them using the combiner c.

To join data from multiple data sources, MRQL supports various physical join operators. The best known join algorithm in an MR environment is the reduce-side join [33], also known as partitioned join or COGROUP in Pig. It mixes the tuples of two input data sets S_1 and S_2 at the map side, groups the tuples by the join key, and performs a cross product between the tuples from S_1 and S_2 that correspond to the same join key at the reduce side. We translate the reduce-side join to the following BSP operation:

$$\begin{array}{l}
\mathsf{mapReduce2}(m_1, m_2, r)\ (S_1, S_2) \\
= \mathsf{bsp}(\ t_1, \\
\qquad \lambda(ms, as, b). \\
\qquad\quad \mathsf{if}\ b \\
\qquad\quad \mathsf{then}\ (\ \mathsf{F}(m_1, m_2, t_1, as), \{\ \}, \mathsf{false}, \mathsf{allFalse}\ ) \\
\qquad\quad \mathsf{else}\ (\ \{\ \}, \mathsf{G}(r, ms), \mathsf{false}, \mathsf{allTrue}\ ), \\
\qquad \mathsf{true}\ )\ (S_1 \cup S_2)
\end{array}$$

where t_1 is the tag of S_1, m_1 and m_2 are the map functions for S_1 and S_2, respectively, and r is the reduce function. The code for F and G is as follows:

$$\begin{array}{l}
\mathsf{F}(m_1, m_2, t_1, as) \\
= \mathsf{cmap}(\lambda(t, c). \\
\qquad\quad \mathsf{if}\ t = t_1 \\
\qquad\quad \mathsf{then}\ \mathsf{cmap}(\lambda(k, z).\, \{(k, (k, (1, z)))\})\ (m_1(c)) \\
\qquad\quad \mathsf{else}\ \mathsf{cmap}(\lambda(k, z).\, \{(k, (k, (2, z)))\})\ (m_2(c)))\ as
\end{array}$$

```
G(r, ms)
= cmap(λ(k, s).r( cmap(λ(t, x).if t = 1 then {x} else {}) s,
                  cmap(λ(t, y).if t = 2 then {y} else {}) s ))
        (groupBy(ms))
```

VIII. Handling Total Aggregations

Total aggregations summarize collections into values. For example, the MRQL query:

```
avg(select e.salary from e in Employees)
```

returns a single number. When aggregations are executed in a distributed system, one peer must be designated as a master to collect all partial aggregations from the other peers and emit the final aggregation result. Our MR algebra includes the operation aggregate(a, S), where a is a commutative monoid $a = (\oplus, z)$. That is, $x \oplus y = y \oplus x$ and $x \oplus z = x$, for all x and y. It reduces a dataset S of type $\{T\}$ into a value T by applying $\oplus$ to the elements of S in pairs. Our BSP algebra uses the same operation aggregate(a, S). It is implemented with special code in Hama that designates one of the participating peers (the master) to collect the final result, it partially aggregates the results at each peer, which sends its partial aggregation result to the master, and, finally, the master dumps the final result to DFS.

```
mapAggregateReduce(m, r, acc, zero) S
= bsp( tag,
       λ(ms, as, b).
         if b
         then ( cmap(λ(k, c).{(shuffle(k),(k, c))})
                     (cmap(λ(k, c).m(c)) as),
                { }, false, allFalse )
         else ( { }, aggregate( acc, zero,
                                cmap(r) (groupBy(ms)) ),
                false, allTrue ),
       true ) S
```

IX. Improving the BSP Evaluation

The improvement we consider in this section is reducing the number of supersteps in a BSP job. In general, two consecutive supersteps can run as one (without a barrier synchronization) if there are no messages sent by peers during the first superstep. In most cases, it is very hard to prove that a superstep results to an empty bag of messages for all peer states. But our MR-to-BSP transformations control the superstep evaluation using a DFA state, which gets the same value across all peers, since all peers do the same state transitions at the same time. More specifically, the false branch of the mapReduce mapping and the true branch of the repeat mapping in Section V return empty messages and apply to all peers at the same time. We can mark this empty bag of messages { } as a special value, which, when received by the BSP execution engine, it skips the barrier synchronization, since it is certain that there will be no messages sent by any peer. More specifically, the Hama implementation of the bsp operator given in Figure 2 will not call peer.sync() if the result.messages is equal to this special empty bag (this can be tested in Java using object equality). This is a very effective evaluation speedup that results to one superstep per MR job for a chain of MR jobs, since the reduce superstep of one MR job is combined with the map superstep of the next MR job.

```
type point = < X: double, Y: double >;

function distance ( x: point, y: point ): double {
   sqrt(pow(x.X−y.X,2)+pow(x.Y−y.Y,2))
};

repeat centroids = { < X: 0.0, Y: 0.0 >,
                     < X: 10.0, Y: 0.0 >,
                     < X: 0.0, Y: 10.0 >,
                     < X: 10.0, Y: 10.0 > }
  step select ( < X: avg(s.X), Y: avg(s.Y) >, true )
       from s in source(binary,"points.bin")
       group by k: (select c from c in centroids
                    order by distance(c,s))[0]
  limit 10;
```

Fig. 4. K-means Clustering Expressed as an MRQL Query

X. Performance Evaluation

MRQL is implemented on top of Hadoop and Hama. It is available at http://lambda.uta.edu/mrql/. It is an open source project available at GitHub https://github.com/fegaras/mrql, where other people can contribute and request changes. MRQL can execute MRQL queries in two modes: using the MR framework on Apache Hadoop or using the BSP framework on Apache Hama. The MRQL query language is powerful enough to express most common data analysis tasks over many forms of raw data, such as XML and JSON documents, binary files, and line-oriented text documents with comma-separated values.

The platform used for our evaluations was a small cluster of nine Linux servers, connected through a Gigabit Ethernet switch. The cluster is managed by Rocks Cluster 5.4 running CentOS-5 Linux. For our experiments, we used Hadoop 1.0.3 and Hama 0.5.0. The cluster frontend was used exclusively as a NameNode/JobTracker, while the rest 8 compute nodes were used as DataNodes/TaskTrackers for Hadoop MR and as Groom servers (BSP nodes) for Hama. Each server has 4 Xeon cores at 3.2GHz with 4GB memory. That is, there were a total of 32 cores available for MR and BSP tasks. For our experiments, Hama's checkpointing was turned off.

In the rest of this section, we present experiments comparing the BSP to the MR evaluation plans, produced by MRQL for two analytical task queries: K-means clustering and PageRank.

A. K-means Clustering

As a first example, the MRQL query in Figure 4 calculates the k-means clustering algorithm, by deriving k new centroids from the old, where the input data set is of type {<X:double,Y:double>}, centroids is the current set of centroids (k cluster centers), and distance calculates the distance between two points. The query in the group-by assigns the closest centroid to a point s. This query clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the mean values of its points.

The dataset used in our experiments consists of random (X, Y) points in 4 squares: $X \in [2 \ldots 4, 6 \ldots 8]$ and $Y \in$

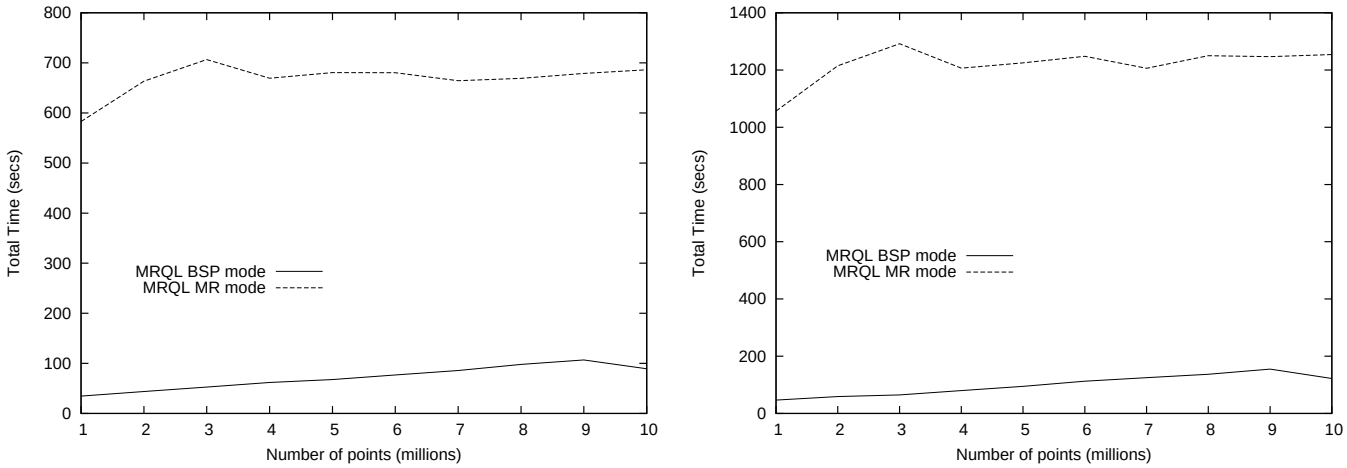

Fig. 5. K-Means Clustering Using MR and BSP Modes for 5 Steps (Left) and 10 Steps (Right)

$[2 \ldots 4, 6 \ldots 8]$. Thus, the 4 centroids were expected to be $(3, 3)$, $(3, 7)$, $(7, 3)$, and $(7, 7)$. As we can see from the query, the initial centroids were $(0, 0)$, $(10, 0)$, $(0, 10)$, and $(10, 10)$. Figure 5 shows the results of evaluating the K-means query in Figure 4 using MR and BSP modes for limit (number of iterations) equal to 5 and 10, respectively. We can see that the BSP evaluation outperforms the MR evaluation by an order of magnitude.

B. PageRank

The second MRQL query we evaluated was PageRank over synthetic datasets. The complete PageRank query is given in Figure 6. Given that our datasets represent a graph as a flat list of edges, the first query in Figure 6 groups this list by the edge source so that each tuple in the resulting graph contains all the neighbors of a node in a bag. We only measured the execution time of the last query in Figure 6, which calculates the PageRank of the graph (this is done by the repeat MRQL expression) and then orders the nodes by their rank. Recall from Section III that, for the repeat to converge, the condition abs((n.rank-m.rank)/m.rank) $>$ 0.1 must become false for all graph nodes. The optimized query requires one MapReduce per iteration. The inner select-query in the repeat step reverses the graph by grouping the links by their link destination and it equally distributes the rank of the link sources to their destination. The outer select-query in the repeat step recovers the graph by joining the new rank contributions with the original graph so that it can be used in the next iteration step. The repeat step, if evaluated naively, requires two MR jobs: one MR job to group the nodes by their destination (inner query), and one MR job to join the rank contributions with the nodes (outer query). Our system translates this query to one MR job by using the following two algebraic laws: The first rule indicates that a group-by before a join can be fused with the join if the group-by attribute is the same as the corresponding join attribute. The resulting reduce-side join nests the data during the join, thus incorporating the group-by effects. The second rule indicates that a reduce-side self-join (which joins a dataset with itself) can be simplified to one MR job that traverses the dataset once. In essence, the map function of this MR job sends each input element to the reducers twice under different keys: under the left and under the right join keys. Consequently, the group-by operation in the repeat step is fused with the join, based on the first rule, deriving a self-join, which, in turn, is simplified to a single MR job, based on the second rule.

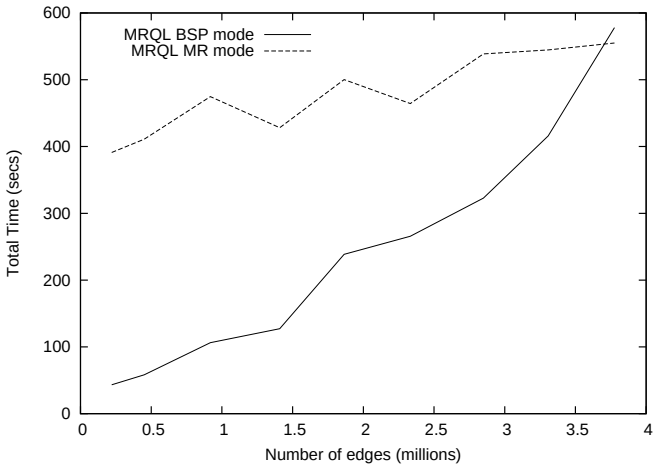

Fig. 7. PageRank Evaluation Using MR and BSP Modes

We evaluated PageRank over synthetic data generated by the R-MAT algorithm [6] using the parameters a=0.57, b=0.19, c=0.19, and d=0.5 for the Kronecker graph generator. The number of distinct edges generated were 10 times the number of nodes. We used 9 different datasets with the following number of edges: 0.25M, 0.5M, 1M, 1.5M, 2M, 2.5M, 3M, 3.5M, and 4M. PageRank required 5-6 steps in MR mode and 19-29 BSP supersteps to converge. Figure 7 shows our results. We can see that, although BSP evaluation outperforms the MR evaluation for small datasets, when the datasets cannot fit in memory, they are spilled to local files and the BSP performance deteriorates quickly.

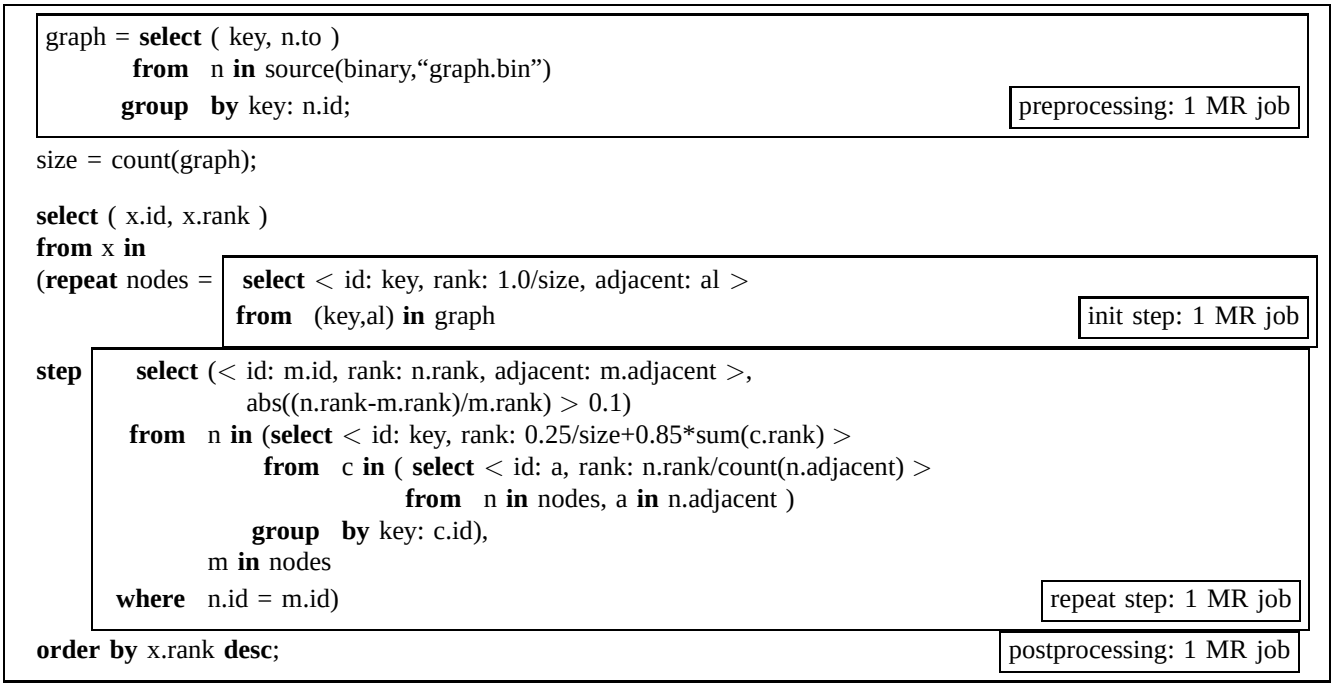

Fig. 6. The PageRank Expressed as an MRQL Query

XI. Conclusion and Future Work

We have presented a framework for translating MRQL queries to both MR and BSP evaluation plans, leaving the choice to be made at run-time based on the available resources. This translation to BSP plans is performed after MRQL queries have been translated and optimized to MR physical plans. There are many improvements that we are planning to add to our system. Our BSP normalization method assumes that fusing two BSP jobs into one BSP job avoids materialization of the intermediate data on DFS and, consequently, improves BSP execution. This is true, provided that the combined state of the resulting BSP job fits in memory. This may not be necessarily true for a three-way join, which corresponds to two BSP jobs, since it may not be possible to fit the three input sources in memory, while it could be possible if we do the join using two BSP jobs. Determining whether the state of two fused BSP jobs can fit in memory requires estimating the size of the resulting state, which, in turn, requires data statistics. In addition, our implicit assumption was that, if there were enough resources, the BSP implementation beats the MR one since the former can run entirely in memory. This may not be necessarily true and may depend on how frequently we do checkpointing and what amount of data we checkpoint. We plan to look at the cost of both MR and BSP plans under various conditions, resources, and checkpointing scenarios, to make better choices. Finally, we would like to experiment with other in-memory/hybrid distributed evaluation systems as a target for MRQL, such Hyracks and Spark.

References

[1] D. Battre, *et al.* Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SOCC'10*.

[2] A. Behm, V. Borkar, M. J. Carey, *et al.* ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. Distrib Parallel Databases (2011) 29: 185–216.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB'10*.

[4] Y. Bu, *et al.* Scaling Datalog for Machine Learning on Big Data. In *CoRR'12*.

[5] R. Chaiken, *et al.* SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *PVLDB'08*.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM'04*.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*.

[8] J. Ekanayake, *et al.* Twister: A Runtime for Iterative MapReduce. In *MAPREDUCE'10*.

[9] L. Fegaras, C. Li, U. Gupta, and J. J. Philip. XML Query Optimization in Map-Reduce. In *WebDB'11*.

[10] L. Fegaras, C. Li, and U. Gupta. An Optimization Framework for Map-Reduce Queries. In *EDBT'12*.

[11] A. F. Gates, *et al.* Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *PVLDB* 2(2), 2009.

[12] Giraph. http://incubator.apache.org/giraph/.

[13] A. Ghoting, *et al.* SystemML: Declarative Machine Learning on MapReduce. In *ICDE'11*.

[14] Hadoop. http://hadoop.apache.org/.

[15] Hama. http://incubator.apache.org/hama/.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys'07*.

[17] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD'09*.

[18] J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. Book pre-production manuscript, April 2010.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *VLDB'12*.

[20] G. Malewicz, *et al.* Pregel: a System for Large-Scale Graph Processing. In *PODC'09*.

[21] MRQL. http://lambda.uta.edu/mrql/.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-Foreign Language for Data Processing. In *SIGMOD'08*.
[23] M. F. Pace. BSP vs MapReduce. In *ICCS'12*.
[24] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming* 13(4), 2005.
[25] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI'10*.
[26] S. Seo and E. J. Yoon HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *CloudCom'10*.
[27] Shark (Hive on Spark). http://shark.cs.berkeley.edu/.
[28] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs In *VLDB'12*
[29] A. Tiskin. The Bulk-Synchronous Parallel Random Access Machine. *Theoretical Computer Science, 196(1,2)*, 1998.
[30] A. Thusoo, *et al*. Hive: a Warehousing Solution over a Map-Reduce Framework. In *PVLDB* 2(2), 2009.
[31] A. Thusoo, *et al*. Hive: A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE'10*.
[32] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. of ACM*, 33:103-111, 1990.
[33] T. White. Hadoop: The Definitive Guide. O'Reilly, 2009.
[34] M. Zaharia, *et al*. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012.