Abstract—The MapReduce framework has become the de-facto framework for large-scale data analysis and data mining. One important area of data analysis is graph analysis. Many graphs of interest, such as the Web graph and Social Networks, are very large in size with millions of vertices and billions of edges. To cope with this vast amount of data, researchers have been using the MapReduce framework to analyze these graphs extensively. Unfortunately, most of these graph algorithms are iterative in nature, requiring repetitive MapReduce jobs. We introduce a new design pattern for a family of iterative graph algorithms for the MapReduce framework. Our method is to separate the immutable graph topology from the graph analysis results. Each MapReduce node participating in the graph analysis task reads the same graph partition at each iteration step, which is made local to the node, but it also reads all the current analysis results from the distributed file system (DFS). These results are correlated with the local graph partition using a merge-join and the new improved analysis results associated with only the nodes in the graph partition are generated and dumped in the DFS. Our algorithm requires one MapReduce job for preprocessing the graph and the repetition of one map-based MapReduce job for the actual analysis.

Index Terms—MapReduce, Distributed Computing, Graph Algorithms

I. INTRODUCTION

Recently, the MapReduce programming model [1] has been emerged as a popular framework for large-scale data analysis on the cloud. In particular, Hadoop, the most prevalent implementation of this framework, has been used extensively by many companies on a very large scale. Many of the data being generated at a fast rate take the form of massive graphs containing millions of nodes and billions of edges. One graph application, which was one of the original motivations for the MapReduce framework, is page-rank [2], that calculates the relative importance of web pages based on the web graph topology. Analysis of such large graphs is a data intensive process, which motivates the use of the MapReduce paradigm to analyze these graphs.

The execution time of a MapReduce job depends on the computation times of the map and reduce tasks, the disk I/O time, and the communication time for shuffling intermediate data between the mappers and reducers. The communication time dominates the computation time and hence, decreasing it will greatly improve the efficiency of a MapReduce job. Previous work required the whole graph to be shuffled to and sorted by the reducers, leading to the inefficient graph analysis. This problem becomes even worse given that the most of these algorithms are iterative in nature, where the computation in each iteration depends on the results of the previous iteration. To improve the efficiency of graph analysis, some earlier work has been done on reducing the size of the input so that graph partitions are small enough to fit in the memory of a single cluster node. In addition, the Schimmy design pattern has been introduced to avoid passing the graph topology across the network. Unfortunately, this method still requires the partial results computed for each node to be shuffled among the nodes. There is also earlier work on optimizing iterative MapReduce jobs, such as Twister [3] and HaLoop [4]. Furthermore, work has been done on implementing graph analysis in other parallel programming paradigms, such as the bulk synchronous parallel [5] paradigm, such as Pregel [6] by Google, and Hama [7] and Giraph [8] by Apache.

In this paper, we introduce a new design pattern for a family of iterative graph algorithms. Our method is to separate the immutable graph topology from the graph analysis results. Each MapReduce node participating in the graph analysis task always reads the same graph partition at each iteration step, which is made local to the node, but it also reads all the current analysis results from the distributed file system (DFS). These results are correlated with the local graph partition using a merge-join and the new improved analysis results associated with only the nodes in the graph partition are generated and dumped in the DFS. Our method requires that the partial analysis results associated with only those nodes that belong to the local graph partition be stored in memory, which is usually far smaller than the graph partition itself since the number of nodes is usually far less than the number of edges. Our algorithm requires one MapReduce job for preprocessing the graph and the repetition of one map-based MapReduce job (ie, a job without a reduce phase) for the actual analysis.

The rest of the paper is described as follows: Section II gives description of related work. Section III describes the previous design patterns for graph analysis. Sections V describes our proposed map-based design pattern for the graph analysis. Finally, Section VI evaluates the performance of our graph analysis using various data sets and compares it with the Schimmy approach.

II. RELATED WORK

The MapReduce model was first introduced by Google in 2004 [1]. Several large organizations have implemented this model including Apache Hadoop [9], Google Sawzall [10], and Microsoft Dryad [11]. Hadoop, an open source project developed by Apache, is the most popular MapReduce implementation. It is being used by many companies to perform...
data analysis on large scale.

Since most graph processing algorithms, such as breadth-first-search, are iterative in nature, they require repetitive map-reduce jobs. Earlier work [12] [13] on graph processing based on MapReduce used algorithms that read and shuffle the whole graph among the participating nodes at each iteration step, resulting in inefficient graph analysis.

We briefly describe some related work that addresses the problem of inefficiency of graph processing in the MapReduce framework. Twister [3] and HaLoop [4], as stated before, are developed to support iterative computations in the MapReduce framework. The main feature of Twister is the long running mapper and reducer tasks with a cacheable distributed memory, which was used to avoid the repeated data loading from disks. Twister is also a stream-based system where the output of mappers is directly streamed to reducers. HaLoop caches the loop-invariant data structures, thus reducing the cost of loading and shuffling them in subsequent iterations. Neither Twister nor HaLoop are scalable, because they both require memory cache for storing and indexing the data, which is proportional to the size of data.

The Schimmy design pattern [14], introduced by Lin and Schatz, uses the notion of the parallel merge join to avoid the shuffling of the graph topology in MapReduce but it still requires the shuffling of the partial results generated for each vertex or edge of the graph. MapReduce was also used to filter data and reduce the data size in a distributed fashion [15], so that the data analysis can be done in a single machine. This method is not applicable to some graph algorithms, such as the PageRank, which requires the whole graph topology and the metadata of all vertices to compute the popularity of all the vertices of the graph.

There has been a recent development in incremental processing systems, such as Incoop [16], a Hadoop-based incremental processing system, and Google’s Percolator [17], an incremental processing system based on BigTable. These systems target applications that process their data in the form of small updates. They do not target general graph processing algorithms.

Another programming paradigm for parallel graph processing is the bulk synchronous parallel (BSP) [5] programming paradigm. Google’s Pregel [6], Apache Hama [7], and Apache Giraffe [8] are systems based on the BSP model. The disadvantage of the BSP model is that it requires that the whole graph be stored in the collective memory of the cluster, which limits the size of graphs that can be analyzed.

III. Graph Algorithms

A graph is defined as \( G(V, E) \), where \( V \) is a set of vertices and \( E \) is a set of directed edges. A directed edge from \( v_i \) to \( v_j \) is represented as the pair of nodes \((v_i, v_j)\), where \( v_i \in V \) and \( v_j \in V \). Each vertex may have some information associated with it (such as, a node label, a page-rank value, the number of out-links, etc) and, similarly, each edge may have some information associated with it (such as, edge label and relationship type).

The focus of this paper is on iterative graph algorithms on directed graphs where partial results associated with nodes can be improved at each iteration. Such graph algorithms can be formulated as follows:

```
repeat
   for all \( v_n \in V \) do
      \( R_n \leftarrow F_n \)
      \( F_n \leftarrow f(\{F_m| (v_m, v_n) \in E\}) \)
   end for
until \( \forall v_i \in V : \rho(R_i, F_i) < \theta \)
```

where \( F_n \) and \( F_m \) are the partial results at vertex \( n \) and \( m \), respectively, \( f \) is a function to compute the partial result for each vertex of the graph, \( \rho \) is the function to compute the result improvement between iterations, and \( \theta \) is the threshold determining the stopping condition. The algorithm given above repeats until the termination criterion is met.

Page-Rank, a well-known algorithm for computing the importance of vertices in a graph based on its structure, can be captured using the above algorithm. It computes the pagerank \( P_i \) for every vertex \( v_i \in V \) belonging to the graph. \( P_i \) is the probability of reaching the vertex \( v_i \) through a random walk in the graph, which is computed based on the topology of the graph. The page-rank computation often includes a random periodic jump to any other vertex in the graph with a probability \( 1 - d \), where \( d \) is the dumping factor. The page-rank of a vertex \( v_i \in V \) of the graph is calculated iteratively as follows:

\[
P_i = \frac{1 - d}{|V|} + d \sum_{(v_j, v_m) \in E} \frac{P_j}{|\{v_m| (v_m, v_j) \in E\}|}
\]

where \( v_i, v_j \) and \( v_m \) are the vertices of the graph, \( P_j \) is the page-rank of node \( v_j \) from the previous iteration, and \( P_i \) is the new page-rank of the vertex \( v_i \). The page-rank equation can be compared to the general iterative algorithm where calculating the page-rank of the vertex \( v_i \) in single iteration is the function \( f \) and the page-rank calculated for all the vertices can be seen as a partial result which will be used to calculate the page-rank of all the vertices in the next iteration.

IV. Earlier Work

A. Basic Implementation

Although they can apply to other graph algorithms too, we describe the earlier work on graph analysis based on MapReduce in terms of the page-rank algorithm. A graph in a MapReduce framework is typically represented as a set of directed edges, where each edge is represented as a key-value pair with the source vertex as the key and the destination vertex as the value. Each vertex \( p \) contains the identifier of the vertex \( p.id \) and its meta-data, which includes its current page-rank value \( p.pageRank \) and the number of outgoing edges \( p.numOfOutLinks \) from the vertex.

We first describe the basic approach of applying MapReduce to the graph algorithms described in Section IV. The mapper function applies to each key-value pair, with the source vertex serving as a key. It computes the page-rank contributions
from the source vertex to the destination vertex and emits
the destination vertex id as the key and its corresponding
fraction of page-rank as the value. In addition to the page-
rank contributions, the mapper regenerates the graph structure
by emitting the source vertex id as the key and the whole edge
(a pair) as the value.

The reducer gets the page-rank contributions from each of
the incoming edges to a vertex, along with the graph topology
associated with the vertex. These page-rank contributions are
aggregated to get the updated page-rank value of the vertex.
The reducer also updates the page-rank value of the source
vertex and the revised edge is written back to the disk. This
completes an iteration of the page-rank computation and the
output is then fed again to the mapper to begin the next
iteration. Note that along with the intermediate partial result
of the vertex \( M \), the set \( N \) of all incoming edges to \( M \) is
also passed. This is necessary in order to preserve the graph
topology to be used in the next iteration steps. As a result, two
types of data are being passed from mappers to reducers, one
is the partial computation of the vertex value and the other is
the incoming edges to the vertex, which passes the topology of
the graph to the reduce phase. Note that this pseudo-code does
not take the damping factor and dangling nodes into account.

B. The Schimmy Implementation

The basic implementation of a graph algorithm passes
two types of data from mappers to reducers. One is the
partial result computed for the vertex and the other is the
graph topology itself. After receiving the partial results for a
vertex and the graph topology associated with it, the reducer
aggregates the partial results and updates the metadata of
the nodes. The shuffling of the graph structure between the
mapper and reducer has high overhead, especially in the case
of iterative algorithms.

To address the inefficiency of the basic implementation, the
Schimmy design pattern was introduced [14]. The Schimmy
design pattern is based on the concept of the parallel merge
join. A merge join between two given relations \( S \) and \( T \) is
done by first sorting both relations on their join keys
and then by simultaneously scanning them, joining the rows
having the same join key. This merge join can be processed
in parallel by partitioning \( S \) and \( T \) into small files \( S_1, \ldots, S_n \)
and \( T_1, \ldots, T_m \), respectively, based on their join key and by
sorting each partition on the join key. Then, each pair \( S_i/T_j \) is
processed by a single node that performs a local merge join
and the node results are combined.

In the Schimmy design pattern, the graph \( G \) is partitioned
into \( m \) partitions, so that each reducer \( R_i \) is assigned a
different partition \( G_i \) and the edges of each partition are sorted
by the ID of the source vertex. The reducer \( R_i \) works on
the intermediate partial results corresponding to the vertices
in partition \( G_i \) and uses a merge-join between these results
and the partition \( G_i \) to calculate new improved results for the
vertices (Algorithm ??).

The implementation of the page-rank based on the Schimmy
design does not need to shuffle the graph structure. In the
reducer the corresponding graph partition file is opened (line
2). The reducer reads through this file until it finds the edge to
be updated, then updates the page-rank of the source vertex of
the edge, and then advances to the next edge. It updates all the
edges with the same source vertex. Once an edge is updated,
it is written back to the distributed file system. In addition to
the design pattern, the Schimmy approach introduced various
improvements, such as using a regular MapReduce combiner
or an in-mapper combiner, which was found to perform better
than a regular combiner. For more details, refer to [14]

V. Map-Based Graph Analysis

Although the Schimmy design pattern improves the effi-
ciency of graph algorithms by avoiding the shuffling and
sorting the the graph topology, it still requires shuffling and
sorting of the partially computed results. To avoid this, we
introduce a map-based design pattern for the analysis of the
graph. In contrast to the Schimmy approach, which requires
both a map and a reduce stage at each iteration of the graph
analysis, our method requires just a map stage. As in the case
of Schimmy, our method too uses a parallel merge-join. In
Schimmy, the merge join happens in the reduce stage between
a partition of the graph and the intermediate partial results
generated from the mappers. In our case, the merge-join is
done at the map stage between a partition of the graph and a
local file (stored in DFS) that contains all the partial results.
Figure 2 illustrates this idea.

More specifically, we perform the graph analysis by doing a
parallel merge-join between the partition of graph and a table
that contains the partial results associated with all nodes.
The graph \( G \) is partitioned into \( G_1, \ldots, G_m \) such that edges
with the same destination go to the same partition. Also, each
of the partitions is sorted by the source vertex of the edges.
A global table in the form of a binary DFS file is created that
contains the partial results of each node after the end of each
iteration. This file is kept sorted by the vertex’s ID. Without
loss of generality, we describe our graph analysis technique
applied to the page-rank computation.

Each mapper \( M_i \) reads the same unchanged partition \( M_i \)
and combines it with the global page-rank table using a merge-
Algorithm 1 The Mapper for Map-Based Parallel Merge-Join for Computing Page-Rank

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td><code>function INITIALIZE()</code></td>
</tr>
<tr>
<td>2:</td>
<td>P:OpenPageRankFile()</td>
</tr>
<tr>
<td>3:</td>
<td>Dictionary D ← empty</td>
</tr>
<tr>
<td>4:</td>
<td>(n, rank) ← P:Read()</td>
</tr>
<tr>
<td>5:</td>
<td><code>end function</code></td>
</tr>
</tbody>
</table>

6: `function MAP(Vertex from, Vertex to)`

7: if `from.id = n` then
8:   D[to] += `rank/from.numOfOutLinks` |
9: `end if` |
10: if `from ≥ n` then
11:   repeat |
12:     (n, rank) ← P:Read() |
13:   until `from ≥ n` |
14: `end if` |
15: `end function` |

16: `function CLOSE` |
17: P:WritePageRankFile(D) |
18: `end function` |

join between $M_i$ and the entire global table. Each mapper $M_i$ generates new page-rank values only for those nodes that are destinations of the edges in $G_i$. Note that no other mapper has edges that have the same destination as those in $G_i$ due to the way the graph $G$ is partitioned. Thus, each mapper $M_i$ is responsible for generating its own subset of the new page-rank values $P_i$, which does not overlap with anyone else’s. This is done by aggregating the incoming contributions from the current page-ranks stored in the global table. As we can see in Algorithm 1, each mapper uses a dictionary $D$ to store its own partition $P_i$ of the new page-rank values. The average size of the dictionary $D$ is $|V|/m$, where $|V|$ is the number of graph nodes and $m$ is the number of mappers. Our assumption is that $D$ can fit in the mapper’s memory. To analyze the graph, a merge-join is performed between $G_i$ and the global table during each iteration and the new page-rank values are aggregated in $D$, which contains $P_i$. After $G_i$ is processed, $D$ is flushed to DFS and becomes one of the new partitions of the global table.

This algorithm is implemented in MapReduce as follows. An initial MapReduce job, which is a pre-processing step, is first called to partition the graph based on the destination vertex of each edge by using a user-defined Partitioner method (which uses uniform hashing on the destination vertex). At the same time, each partition $G_i$ is sorted by the source node of the edges by using user-defined Key Comparator and Grouping Comparator methods. Once all the partitions are formed, they are saved on the distributed file system. In addition, each reducer node of this MapReduce job generates one partition $P_i$ of the global table that contains the initial page-rank values and saves them in a sequence DFS file. All these files generated by the reducer nodes are saved in the same DFS directory, thus forming the initial global table.

The mapper that evaluates each iteration step is shown in Algorithm 1. It evaluates a merge-join between the graph partition $G_i$ (the mapper input) and the global page-rank table, since $G_i$ is read sorted by the source of the edges and the page-rank table is read sorted by the node. During this join, it aggregates the incoming page-rank contributions for those vertices that are destinations of $G_i$ edges. When the source vertex $from$ of the input vertex has the same ID as the current vertex $n$ read from the page-rank table, it adds a page-rank contribution from the source vertex $from$ to the destination vertex $to$ of the edge. These contributions are aggregated together and saved in the dictionary $D$. After the entire $G_i$ is processed, this dictionary will contain the updated page-ranks of the vertices belonging to that partition. These values are sorted and flushed out to the disk in a sequence file format to form a single partition $P_i$ of the new global page-rank table, which is used by next iteration. The mapper does not write anything to the disk other than the new page-rank values. It should be noted that, to decrease the diskwrite overhead, we set the data replication parameter of the DFS sequence file that contains the global page-rank table to one, so that there is always a single replica of global page-rank table in DFS.

References


