Distributed Data Intensive Computing on Graph

by

UPA GUPTA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

Summer 2016

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of some people.

Firstly, I like to express my sincere gratitude to my advisor, Dr. Leonidas Fegaras who gave me the chance for doing my doctoral under his guidance. It would not have been possible without his expertise, critique, support and understanding. He gave me the freedom to find my own research interest. it did not matter how much i have struggled with my work, he supported me and guided me on my work. Today whatever I know about research, its all becauseof him. He taught me how to understand a research papers, find a research problem, formulate it technically, work on it with patience and concluding the results with brevity and clarity.

Besides my advisor, I would like to thank the rest of my committee: Prof Ramez Elmasri, Prof. Gautam Das, Dr. Chengkai Li and Mr. David Levine, for their insightful comments and encouragement, but also for the hard questions which incented me to widen my research from various perspectives.

My sincere thanks to Dr Ramez Elmasri for letting me joining Mining and Analysis of Spatio-Temporal (MAST) lab and included in their research group and helped me widen my research areas.

Next, I would like to thank my papa, Dr. Krishna Kumar Gupta who inspired me to go for doctoral studies. He has always taught me science and maths from the books and of the life. It is because of him, that i today have a strong foundation for my career.

Now, I would like to say about my mother, Mrs. Rekha Gupta who has been my emotional support through out my life. She has always been there to listen to me patiently for long hours whether i am happy, excited, confused sad, frustrated and angry.

Next, Dr. Mahashweta Das who has guided and inspired me throughout my doctoral. She has been as my big sister as Di. She showed me how to be a doctoral student, work hard and play harder. She has been perfect example in my life. In addition to these, she also became my emotional support and career advisor. Without her, Its hard for me to imagine that I could have completed my work.

I would specially like to thank Mausa and Mausa ji (Mr. Kamal Gupta and Mrs. Ranjana Gupta) who have always helped me in living and adjusting to new place and culture far away from home.

Lastly but not the least, I would like to thank my brother, Priyank Gupta, my friends, Chandeep Singh, Sujoy Bhattacharya, Kulsawasd Jitkajornwanich, Daipayan Sarkar, Somdutta Chowdhury who were there for me during my initial years. Nilabh Ohol, Nirzari Iyer, Mitali Gogate, Pooja Keskar, Ajinkiya Gadave and Ahmed Ulde were their in my last years of my doctoral studies. They were there to support me as a friend and shared my happiness and celebrated my achievements. They have always believed in me.

July 29, 2016

ABSTRACT

Distributed Data Intensive Computing on Graph

UPA GUPTA, Ph.D.

The University of Texas at Arlington, 2016

Supervising Professor: Leonidas Fegaras

Distributed frameworks, such as MapReduce and Spark, have been developed by industry and research groups to analyze the vast amount of data that is being generated on a daily basis. Many graphs of interest, such as the Web graph and Social Networks, increase their size daily at an unprecedented scale and rate. To cope with this vast amount of data, researchers have been using distributed processing frameworks to analyze these graphs extensively. Most of these graph algorithms are iterative in nature, requiring repetitive distributed jobs. This dissertation presents a new design pattern for a family of iterative graph algorithms for the distributed framework. Our method is to separate the immutable graph topology from the graph analysis results. Each compute 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 to the DFS. Our algorithm requires one job for pre-processing the graph and the repetition of one map-based job for the actual analysis. Unfortunately, in most of these iterative algorithms, such as for Page-Rank, if the graph is modified with the addition or deletion of edges or vertices, the Page-Rank has to be recomputed from scratch. We improved our previous design approach and to handle continuous updates, an update function collects the changes to the graph and applies them to the graph partitions in a streaming fashion. Once the changes are made, the iterative algorithm is resumed to process the new updated data. Since a large part of the graph analysis task has already been completed on the existing data, the new updates require fewer iterations to compute the new graph analysis results as the iterative algorithm will converge faster.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

1.1 What is Big Data?

Industry and Researchers generate data everyday and these data need to be analyzed for knowledge discovery and other applications. For example, Google creates web indexes and use them to give the relevant search results to the users, CERN researchers generate petabytes of data everyday which are being analyzed by the researchers all around the world. Facebook is the biggest social network with 1.5 billion active users generates data that are of great interest to the researcher for analysis [REFERENCES]. The size of these data are in terabytes or petabytes scales and hence they can't be stored on a single system and can't be analyzed by a single system also. As the data are generated at higher rates than the rate of data analysis, there is a need to analyze the large amount of data efficiently.

Big data are data sets whose sizes is so large that commonly used software tools are not able to capture, curate, manage, and process. The processing time using these tools is so long that the by the time the data is processed, it has lost its value. Big data "size" is a constantly moving target, as of 2012 ranging from a few dozen terabytes to many petabytes of data. Big data requires new forms of integration to uncover large hidden values from large datasets that are diverse, complex, and of a massive scale.

In a 2001 research report(1) and related lectures, META Group (now Gartner) analyst Doug Laney defined data growth challenges and opportunities as being three-dimensional, i.e. increasing volume (amount of data), velocity (speed of data in and out), and variety (range of data types and sources). The 3V model became the standard model to describe the Big Data. In 2012, Gartner updated this definitions as follows: Big data is high volume, high velocity, and \or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization. (wiki19) Additionally, a new V "Veracity" is added by some organizations to describe it.(wiki20)

Big data can be described by the following characteristics

Volume: The amount of data shows its value and potential and whether it can be considered as big data or not. The name big data itself contains a term related to size, and hence the characteristic.

Variety: The type of content for example, is it a hospital data or social graph data? This helps the analysts to effectively analyze the data to find any pattern or important feature of the data.

Velocity: the speed at which the data is being generated and processed to meet the demands of the industries and researchers.

Variability: The inconsistency the data can show at times.

Veracity: The quality of captured data, which can vary greatly. Accurate analysis depends on the veracity of source data.

Complexity: Data management can be very complex, especially when large volumes of data come from multiple sources. Data must be linked, connected, and correlated so users can grasp the information the data is supposed to convey.

1.2 The Need for Distributed Frameworks

As the big data's size is in petabytes, the datasets are distributed across multiple systems networked together. It became too expensive to develop a system that can handle the emerging requirements of the big data management and analysis. Also, the cost of commodity servers has decreased dramatically and companies started to use them to store data. This led to the development of computing architecture which can easily store data on multiple systems and also analyze and read data stored on multiple systems. This is called a distributed computing architecture. In distributed computing, the computers are networked together and each computer has their own processor and memory, the goal of a distributed system is to solve a large computational problem which is divided among the computers. The characteristics of a distributed systems are: 1. There are several independent system with their own processor and memory connected together on a network 2. These systems communicate with each other using message passing. 3. Failures can occur on the independently on these systems and have to be resolved. 4. The distributed systems are scalable i.e. any number of computational node can be added or removed from the distributed system anytime without affecting the operation of this system.

1.3 Google's MapReduce and Page-rank Analysis

Google was founded by Sergrey Brin and Larry Page in 1998 for giving search results by ranking terms by the order of relevance using an algorithm called PageRank. Because of the large size of the web data, there was a need for a system which can process data at a very large scale. This led to the development of MapReduce model for the processing of petabytes of data.

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. Analysis of large graphs is a data intensive process, which motivates the use of the MapReduce paradigm to analyze these graphs.

1.4 Big Graph Analysis

Google's pagerank has become one of the most important algorithm for the web graph analysis. It is an iterative algorithm with no changes in the topology of the graph between the iterations. Most Graph algorithms like Breadth First Traversal, Depth First Traversal, Connectivity of the Graph are iterative in nature with information being passed from one vertex to the other vertices through the edges connecting them. Such graph analysis is a data intensive process which motivates the use of distributed computing frameworks such as MapReduce framework.

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 [2] 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.

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 to 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.

1.5 Processing Dynamic Graph

Furthermore, many graphs of interest, such as the Web graph and Social Networks, are very dynamic, with millions of nodes and edges added and updated on a daily basis. For example, the Web is evolving at an enormous rate with new Web pages, content, and links added daily. Web graph analysis tools, such as page-rank, which are used extensively by search engines, need to recompute their Web graph measures very frequently since they become outdated very fast. There is a recent interest in incremental Big Data analysis, where data are analyzed in incremental fashion, so that existing results are reused and merged with the results of processing the new data. Incremental data processing can generally achieve better performance than batch processing.

We have introduced an efficient design pattern to handle a family of iterative graph algorithms in a distributed framework, such as Map-Reduce, that avoids the shuffling and sorting of the graph topology. It requires just one map stage, but no reduce stage, at each iteration step. We improved our design pattern for graph analysis that handles graph updates in an incremental fashion. As in our earlier work [9], we separate the graph topology from the graph analysis results. The graph topology remains unchanged across the iteration steps of the graph analysis but is updated when new incremental updates arrive. At each iteration step, each node participating in this graph analysis task, in addition to reading the unchanged single graph partition, it reads all the current analysis results from the DFS. These results are correlated with the local graph partition using a special merge-join and the new improved analysis results are calculated and stored in the DFS, one partition from each worker node. More specifically, we introduce a prepossessing stage before iteration, in which the graph is partitioned on the edge destination, such that edges with the same destination go to the same partition, and each partition is sorted on the edge source. Furthermore, the graph analysis results (such as, the page-rank table) are kept sorted by the node in the form of a sorted Sequence file in DFS. Hence, at each iteration step, the new page-rank table is calculated from the incoming page-rank contributions during a single map task in which a worker joins its graph partition with the entire page-rank table using a merge join.

One problem in our design pattern was that each worker has to scan the entire page-rank table simultaneously and in the same order at each iteration step, thus making the reading of this table the bottleneck of this approach. In this paper, we have resolved this bottleneck by having each worker scan the page-rank table starting from a different partition and process the other partitions in a round-robin fashion, thus assigning a single worker to each page-rank partition each time. This improvement requires that the graph partition is reorganized differently so that graph edges in a graph partition are clustered into groups that are ordered in the same way the worker node scans the page-rank partitions, thus allowing to perform the merge-join without backtracking.

In this improved design, we are also addressing the problem of incremental graph analysis by introducing a novel design pattern that extends our previous work. We take advantage of the fact that an iterative graph analysis will converge to a result with a certain accuracy faster if it uses the previous analysis results as its starting point. To handle continuous batches of updates, an update process collects the changes to the graph and applies them to the graph partitions in a streaming fashion. Once the changes are made, the iterative algorithm is resumed to process the new updated data. Since a large part of the graph analysis task has already been completed on the existing data, the new updates require fewer iterations to compute the new graph analysis results and the iterative algorithm will converge faster. We have implemented our framework on Apache Spark [10] and we have used Spark Streaming [11] to update the graph topology and graph partitions. We have also evaluated the performance of our methods by comparing their efficiency to compute 5 iterations of the page-rank algorithms.

The rest of the thesis is described as follows: Chapter 2 gives description of related work which contains the description of MapReduce model, Bulk Synchronous Model, Apache Spark and some other distributed frameworks. It also contains the description of previous design patterns for the analysis of the graph. Chapter 3 defines the iterative graph algorithms, PageRank and Connectivity of Graph Algorithm. Chapter 4 describes our proposed map-based design pattern for the graph analysis.

Chapter 5 describes the our improved incremental graph analysis. Finally, Chapter 6 evaluates the performance of our graph analysis using various data sets and compares it with the Schimmy approach.

CHAPTER 2

Related Works

2.1 Map Reduce Programming Model

MapReduce is a distributed processing framework that enables data intensive computations. The framework, inspired by the functional programming paradigm, has two main components, a mapper and a reducer. A mapper works on each individual input record to generate intermediate results, which are grouped together based on some key and passed on to the reducers. A reducer works on the group of intermediate results associated with the same key and generates the final result using a result aggregation function. The processing units of the MapReduce framework are key-value pairs. An instance of the MapReduce framework with 3 mappers and 2 reducers is shown in Fig 2.1.

Developers can develop MapReduce applications by providing the implementations for the mapper and the reducer methods. The MapReduce framework handles all the other aspects of the execution on a cluster. It is responsible for scheduling tasks, handling faults and sorting and shuffling the data between the mappers and the reducers, where the intermediate key-value pairs are grouped by key. The MapReduce framework works on the top of a distributed file system, which is responsible for the distribution of the data among all the worker nodes of the cluster.

After each mapper finishes its task, its intermediate generated results are passed to the reducers. a process known as shuffling. Each reducer is assigned a subset of the intermediate key space, called a partition. To control the assignment of the key-value pairs to reducers, the MapReduce framework uses a partitioning function. The

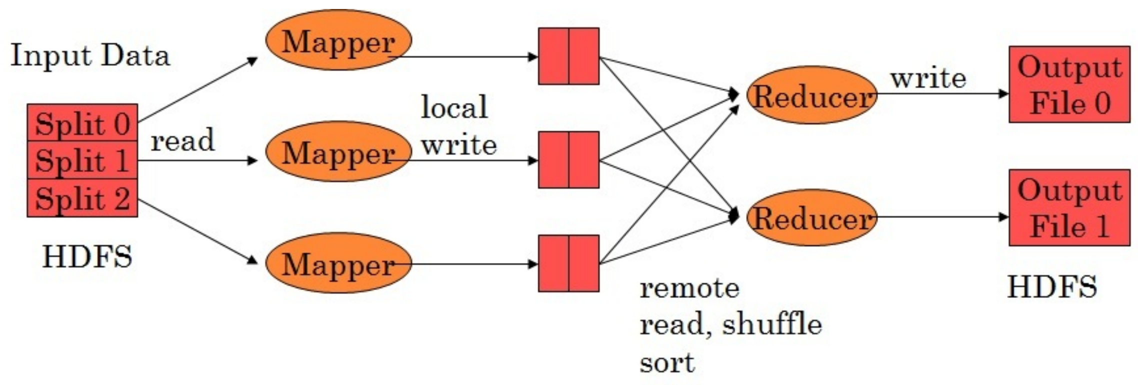

Figure 2.1. An Example of a MapReduce Job Execution.

intermediate values, after being grouped by key, are sorted by the reducers. The sort order can be controlled by a user-defined comparator function.

The MapReduce framework also allows developers to specify a function, called the combiner, to improve performance. It is similar to the reducer function but it runs directly on the output of the mapper. The combiner output becomes the input to the reducer. As it is an optimization, there is no guarantee on the number of times it will be called. When there is a large amount of shuffling of data between the map and the reduce phases, combiners can be used to aggregate the partial result at the map side to reduce the network traffic.

2.2 Apache Spark

Apache Spark is another distributed processing framework designed for data intensive computations. It is based on two main abstractions: the Resilient Distributed Datasets (RDDs) and parallel operations on these datasets. An RDD is a read-only collection of objects partitioned across the cluster. Often, it is stored in-memory, leaving only a small amount of meta-data related to the RDD to be stored on disk, which is used to reconstruct the RDD whenever a node in the cluster fails.

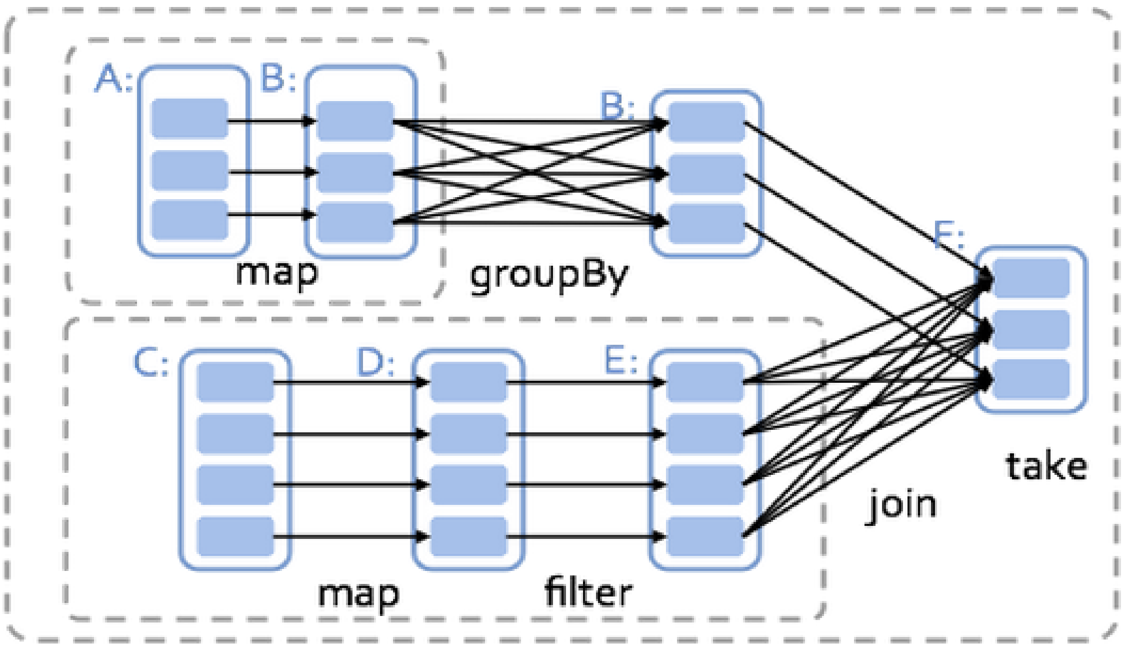

Figure 2.2. An Example of a Spark Job Execution [10].

An RDD can be created from a file stored in HDFS, by dividing it into partitions and distributing these partitions across the worker nodes, by transforming an existing RDD, and by changing the persistence level of an RDD. RDDs are lazy and transient, which means they are constructed on demand when they are used in a parallel operation or being stored on the disk. When RDD operations applied to an RDD, they are queued until they are forced to be applied when the results of these operations need to be stored on disk or sent to other nodes. Spark achieves fault tolerance through lineage, which means that if an RDD partition is lost, there is enough information stored on how the RDD is derived to help rebuild the lost partition.

Spark supports many parallel operations on RDDs, such as reduce, collect and foreach. The reduce operation combines the data in a dataset using an associative function and produces a result. The collect operation collects all the data from partitions and sends it back as one to the main program. The foreach performs a function on each of the RDD partitions and produces a new RDD. In addition to these, there are also simple parallel operations, such as map and filter.

Spark also provides two types shared variables: broadcast variables and accumulators. Broadcast Variables are large read-only data needed by multiple worker nodes, which are distributed across the cluster once before the start of the driver program. Accumulators are generally used to implement counters in a Spark program. Worker nodes can only add to the accumulators using some associative function and the driver program is the only node that can read these accumulators.

Since Spark is an in-memory distributed processing framework, the worker nodes in the cluster require high memory. Hadoop on the other hand, stores results on secondary storage and then reads them again for the next computation step. Being in-memory, the Spark outperforms Hadoop by 10x.

2.3 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) Model [5] is the abstract model for the distributed computing proposed by Leslie Valient in 1990. It consists of: 1. Components performing processing and/or memory functions. 2. Router to deliver messages between any pair of components. 3. A synchronization system to synchronize all or subset of components at regular intervals. Google's Pregel, Apache Hama and Apache Giraph were inspired by the bulk synchronous model. BSP model is designed specially for large matrix computations and graph algorithms.

A BSP algorithm has a different programming paradigm consisting of sequence of supersteps. Each superstep is composed of three ordered phases, as shown in Figure 2.3

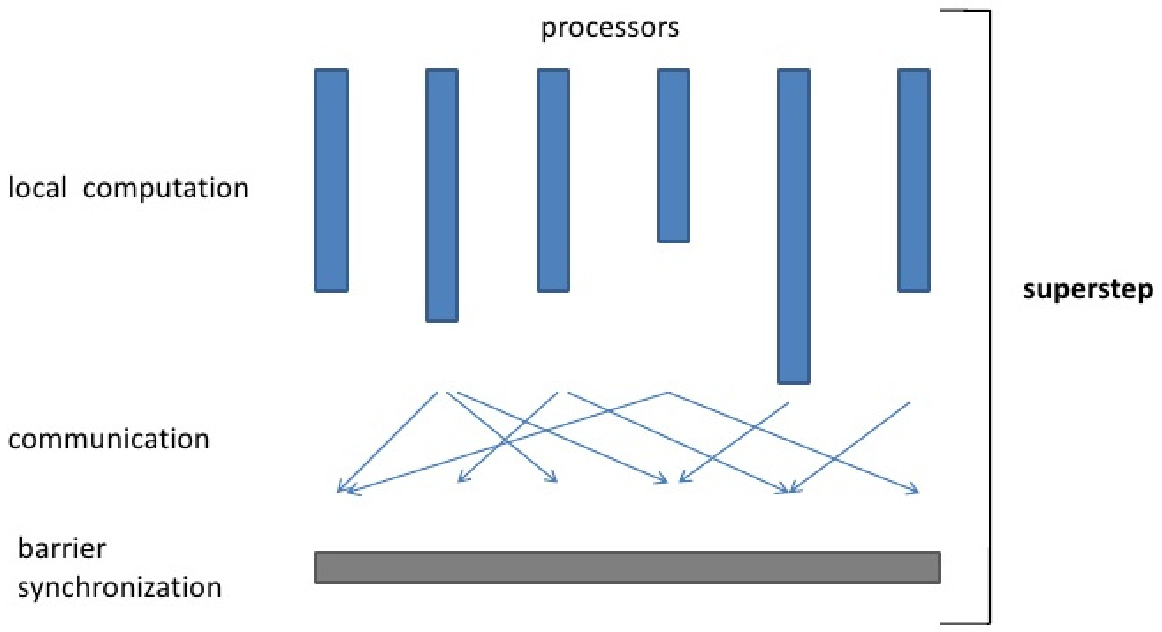

Figure 2.3. Architecture of Bulk Synchronous Parallel Model [5] [8].

1. A Local Computation Phase: Each computer process on the local data available to it and make communication requests such as remote memory reads and writes. The computations occur asynchronously of all the others but may overlap with communication.
2. A Global Communication Phase: Data is exchanged between computers according to the requests made in the computation phase.
3. A Barrier Synchronization: It waits for all data transfers to complete and makes the transferred data available to the computers for use in the next superstep.

The computation and communication actions do not have to be ordered in time. The barrier synchronization ends the superstep. It ensures that all data have been properly communicated across the system. Systems based on two-sided communication include this synchronization cost implicitly for every message sent. To remove the synchronization cost, one-sided communication is used. The method for barrier synchronization relies on the hardware facility of the BSP computer.

The key idea in the BSP model is the synchronization happening in the last. It makes BSP algorithms easier to implement and analyze. This also guarantees that the process being processed by the individual computers are mutually independent.

There has been a rising interest in the BSP model because Google adopt this technology for mainly for graph analytics at massive scale. This gives rise to next generation Hadoop-like distributed frameworks decoupled from MapReduce model. Examples are Apache Hama [7] and Apache Giraph [8].

2.4 Percolator

Percolator [12] is specially designed for incremental processing of a large amount of data. Percolators primary application in Google is to index the web pages for the search queries and this indexing application is implemented as incremental application so as to process individual documents as they are crawled. This reduces the latency in processing document by 50%. Percolator also reprocesses the documents when there is a change in the resources on which web pages depends. Percolator was not designed to replace any exiting solutions for data processing and the also computations which cannot be broken down into small incremental updates are better handled by MapReduce.

There are two main functionality of the Percolator: 1. ACID transactions over a random access repository. 2. Organizing incremental computations with the help of Observes.

Percolator is designed on the top of BigTable [13]. BigTable is the multidimensional table where each cell is identified by the key and cell contains the arbitrary strings of data. It provides atomic read/write on individual rows. Programmer also

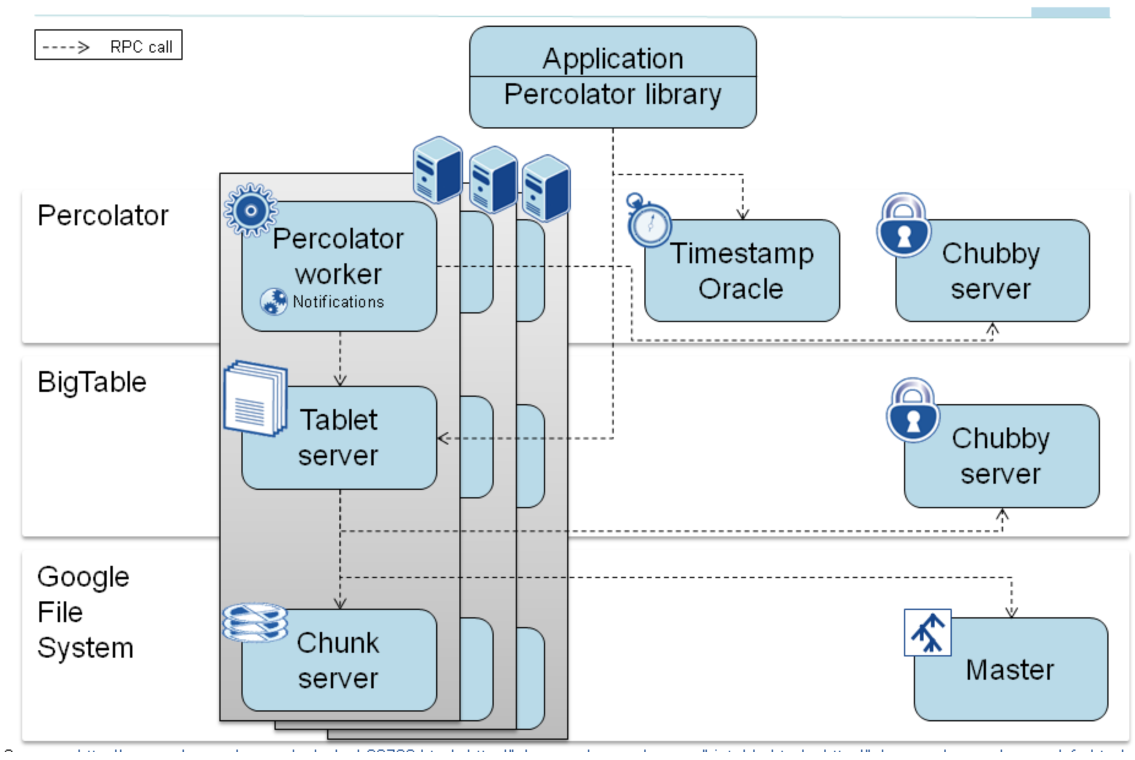

Figure 2.4. Architecture of Percolator [12].

have the capability to group a set of columns into a locality group which makes the scanning less expensive since data is other columns need not be scanned.

As percolator is built over BigTable, it runs three binaries on each system of the cluster. These are: 1. Percolator Worker 2. BigTable Tablet Server 3. GFS Chunk Server The function of Percolator works is to scan the BigTable [13] for any changes and invokes the corresponding observer as a function. These observers perform transactions by sending Read/Write RPCs to BigTable tablet server which in turn send R/W RPCs to GFS chunk server. Percolator performs computations and updates the results by performing transactions and provides cross row, cross table transactions with ACID snapshot isolation semantics.

2.5 Deviations from MapReduce for Incremental Processing

2.5.1 HaLoop

HaLoop [4] is developed from Hadoop for supporting iterative data analysis in a MapReduce style architecture. It mainly focuses on avoiding processing of the unchanged data at each iteration. It also checks termination criteria for ending the iterations early without the need of extra mapreduce job. For achieving its goals, it provides programming interface for developing iterations and HaLoop's scheduler makes ensure that tasks are assigned to the same nodes in each iteration. It keeps the invariant data in cache and hence the data doesn't need to be reloaded at each iteration. Finally, by caching the reduce tasks local output, it is possible to support comparisons of results of successive iterations in an efficient way and allow termination when convergence is identified. Figure 2.5 shows the architecture of HaLoop. It depicts the new loop control module as well as the modules for local caching and indexing in the HaLoop framework. The loop control is responsible for initiating new MapReduce steps (loops) until a user-specified termination condition is fulfilled. HaLoop uses three types of caches: the map task and reduce task input caches, as well as the reduce task output cache. In addition, to improve performance, cached data are indexed. The task scheduler is modified to become loop-aware and exploits local caches. Also, failure recovery is achieved by coordination between the task scheduler and the task trackers.

2.5.2 MapReduce Online

MapReduce online [14] proposes to overcome the built-in feature of materialization of output results of map tasks and reduce tasks, by providing support for pipelining of intermediate data from map tasks to reduce tasks. This is achieved by applying significant changes to the MapReduce framework. In particular, mod-

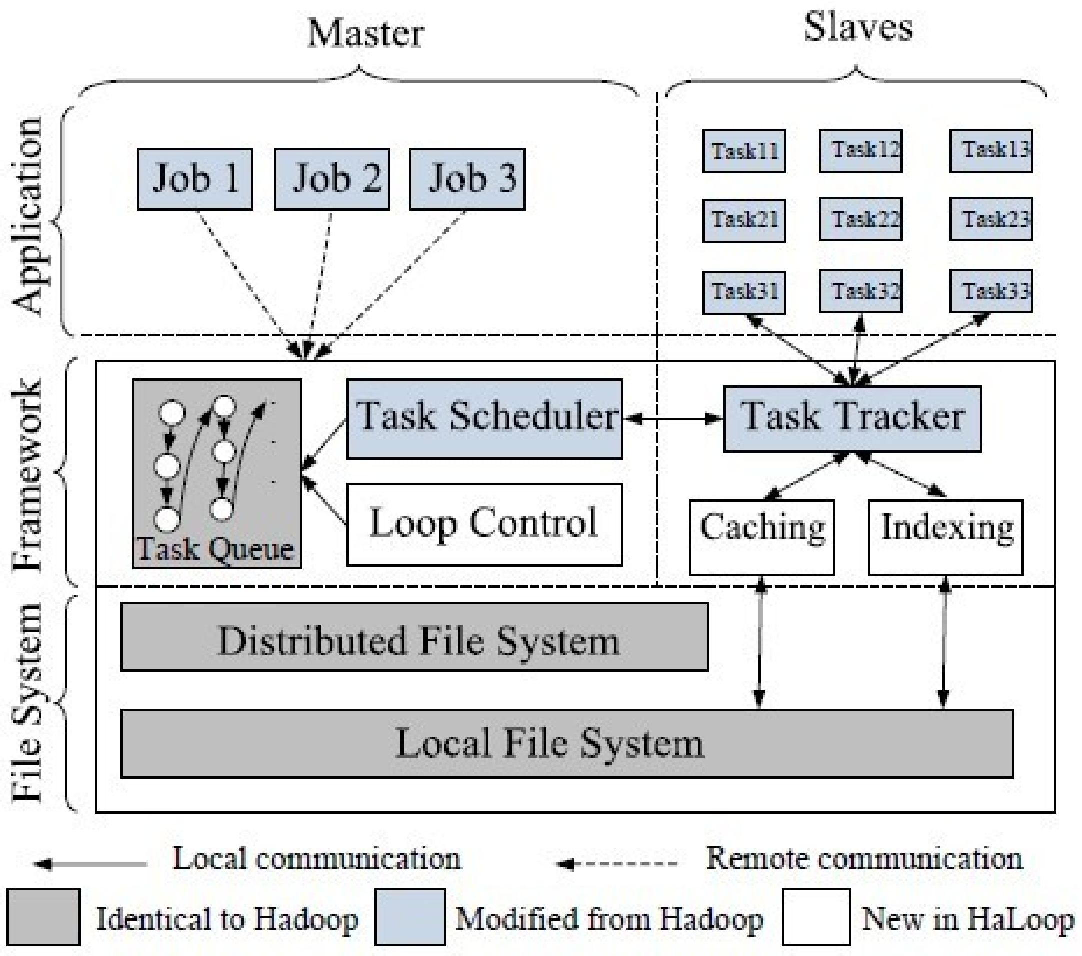

Figure 2.5. Architecture of Haloop [4].

ification to the way data is transferred between map task and reduce tasks as well as between reduce tasks and new map tasks is necessary, and also the TaskTracker and JobTracker have to be modified accordingly. An interesting observation is that pipelining allows the execution framework to support continuous queries, which is not possible in MapReduce. Compared to HaLoop, it lacks the ability to cache data between iterations for improving performance.

2.5.3 Twister

Twister [3] introduces an extended programming model and a runtime for executing iterative MapReduce computations. It relies on a publish/subscribe mechanism to handle all communication and data transfer. On each node, a daemon process is initiated that is responsible for managing locally running map and reduce tasks, as well as for communication with other nodes. The architecture of Twister differs substantially from MapReduce, with indicative examples being the assumption that input data in local disks are maintained as native files (differently than having a distributed file system), and that intermediate results from map processes are maintained in memory, rather than stored on disk. Some limitations include the need to break large data sets to multiple files, the assumption that map output fits in the distributed memory, and no intra-iteration fault-tolerance.

2.6 Other Distributed Frameworks

2.6.1 GraphLab

It is a non-hadoop framework for parallel machine learning. It has two main functions: 1. update: its the same as Map function of the MapReduce but they are allowed to access and modify overlapping contexts in the graph. 2. Sync: its the same as reduce function but they can run concurrently with the update function. GraphLab is based on shared-memory architecture. It maintains a shared data table to support globally support variables.

2.6.2 Pegasus

A system is developed on Hadoop for Graph Mining purposes. They find the connected components, diameter and PageRank of very large graphs using the Generalized Iterated Matrix-Vector Multiplication (GIM-V). The method has three main steps: 1. Combine2: multiply $m_{i,j}$ and v_j, where M is the n x n and v is the vector of size n. 2. combineAll: sum n multiplication results for node i. 3. assign: overwrite previous value of v_i with new result to make v_i. GIM-V is also implemented based on block multiplication where elements of input matrix is grouped into blocks or sub-matrices. This reduces the time to sort the number of items at the shuffling stage.

2.7 Incremental PageRank

In an attempt to reduce the number of iterations, Desikan et al. [6] introduced incremental method to compute Pagerank over evolving graphs. The idea behind this work is the changes in web graphs is slow, with large parts of it remaining unchanged, so we can develop algorithm to calculate PageRank incrementally. In the initial stage, the PageRank must be calculated in normal way; using power method. Later when there are new links, the old graph will be compared with the new graph. The key idea behind Incremental PageRank is to partition the new graph into two main parts. The first part consists of unchanged nodes from previous graph and the second part is contained with changed nodes together with affected nodes. The PageRank value for the first part will be re-scaled while PageRank value for second part will be recalculated in iterative method. The number of iterations to calculate the updated graph is reduced by eliminating unchanged nodes from going to recalculation with power method.

As a working example we will start from Figure 2.6 where we have 8 vertices. Then after some times, the graph changed as in Figure 2.7 where we added vertices 9 and 10 and delete vertex 8. In Incremental PageRank we will tag the node as in Figure 2.7 efore we partition it. We will start from new and deleted node and find all descendant of it using Breadth-First Search algorithm. Vertex 6 and 7 are categorized

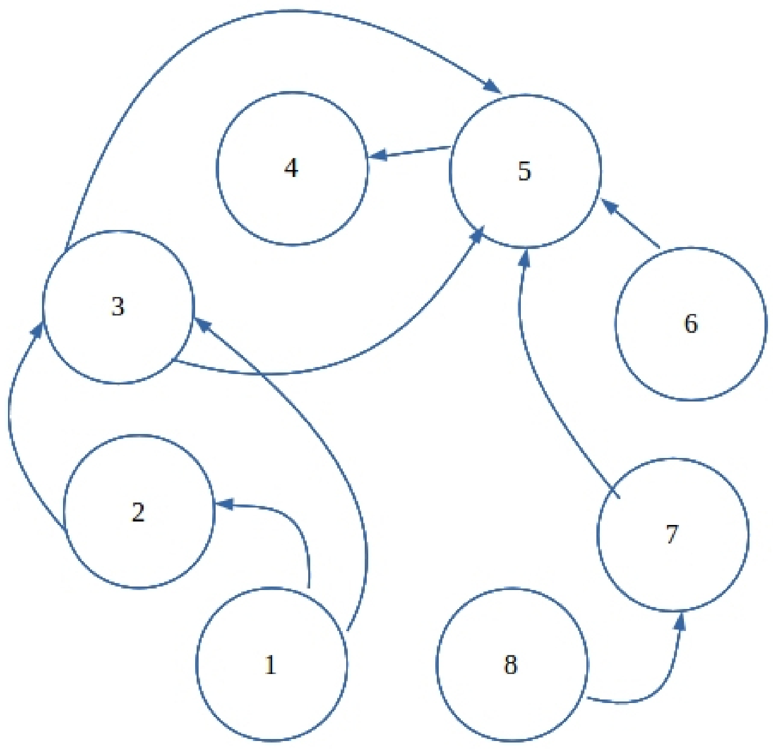

Figure 2.6. Example of Graph for Incremental Analysis.

as changed node because vertex 6 added new in-link while vertex 7 lost an in-link. We also will find all descendant for these two nodes. Vertex 5 and 4 are unchanged node but it will be recalculated because their PageRank value is contributed by changed nodes, vertex 6 and 7. Vertex 3 will go to recalculation stage but will not be recalculated. Vertex 3 needed in recalculation stage because vertex 3 contributed to PageRank value for vertex 5. Vertex 1 and 2 will just be scaled and will not go to iterative recalculation stage because they are unchanged and not descendant of changed node.

Incremental PageRank was evaluated by Desikan et al. with a small dataset which is just a departmental university website that only need at most 12 iterations if the PageRank vector is calculated in naive way. The result of their experiments

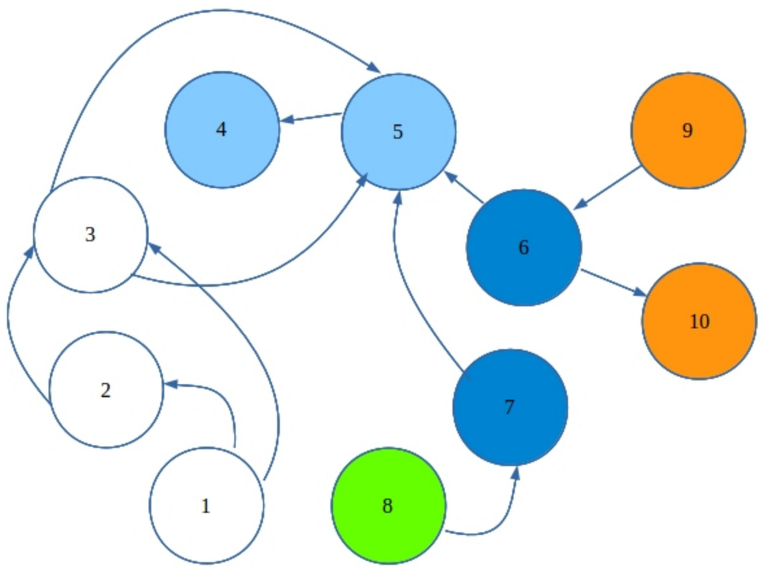

Figure 2.7. Example of changes to graph for incremental analysis.

show that if the changes between the old graph and the new graph is less than 5% the number of iterations needed to recalculate is about 30% - 50% less than naive way. However, if the changes is more than 50%, it only can save 2 iterations out of 12 iterations. The good thing about this method is that the PageRank value is exactly the same compared to the naive update. On the other hand, the problem with this method is that if there is a new node pointing to a node with large number of descendants, it will create a ripple effect and will bring all descendants to iterative stage. Let say the nodes have a million connected to it and later on it just add one new node. The addition will not affect that much on current PageRank but the effort taken to recalculate the PageRank is big.

CHAPTER 3

Iterative Graph Algorithms

3.1 Definition of Graph Analysis

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

The focus of this paper is on iterative graph algorithms on directed graphs where partial results are used to compute the results of the next 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 the function to compute the partial result for each vertex of the graph, ρ is the function to compute the distance between the results of the current iteration and the previous iteration and θ is the threshold determining the stopping condition. The algorithm described above repeats until the termination criterion is met.

3.2 Page-Rank Algorithm

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. PageRank calculation is based on power method which need iterative implementation. This method is very slow to reach convergence. The problem with computing PageRank for very big graph in traditional way is we need machine with high processing power and huge memory. Now we can use distributed frameworks to calculate the PageRank in parallel computation and distribute the processing and memory usage across cluster with cheaper machine.

It is the link-based analysis algorithm used by Google to rank webpages was at first present as citation ranking [15]. Another popular link-based analysis is HITS by Kleinberg which is used by Teoma search engine later acquired by Ask.com search engine which is based on natural language [16]. Long before PageRank, the web graph was largely an untapped source of information. Page et al. used the new innovative way which is not to based solely on the number of inlinks but also consider the importance of page linked to it has made PageRank a popular way to do ranking. In a nutshell, a link from a page to another is understood as a recommendation and status of recommender is also very important.

Page-Rank computes the value P_i for every vertex $v_i \in V$ belonging to the graph where P_i is the probability of reaching the vertex v_i through a random walk in the graph. The probability of reaching a vertex is computed using the topology of the graph but the computation also includes a damping factor d which allows a random periodic jump to any other vertex in the graph. Page-Rank of a vertex $v_i \in V$ of the graph is calculated iteratively as shown below:

$$P_i = \frac{1-d}{|V|} + d \sum_{(v_j,v_i)\in E} \frac{P_j}{|\{v_m|(v_j,v_m) \in E\}|} \tag{3.1}$$

where v_i, v_j and v_m are the vertices of the graph and P_i is the page-rank of the vertex v_i and V is the set of all the vertices of graph G. 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.

CHAPTER 4

Design Pattern Proposed by Others

4.0.1 Basic Implementation

The graph is represented as the set of directed edges where each edge is represented as a key-value pair with source vertex as the key and the destination vertex as the value. Each vertex contains the identifier of the vertex and its corresponding meta-data which includes the current page-rank value of the vertex and number of outgoing edges from the vertex.

Algorithm 1 The Mapper for a Basic Implementation of Page-Rank

```
1: function MAP((Vertex $from$, Vertex $to$))
2:     Emit ($from.id$, ($from, to$))
3:     $p \leftarrow from.pageRank / from.numOfOutlinks$
4:     Emit ($to.id$, $p$)
5: end function
```

We first describe the basic approach to apply the MapReduce to the graph algorithms described in section IV. The basic approach is to make the mappers map over the key-value pairs comprising the graph structure and compute the partial results for each vertex using vertex's meta-data. The partial results computed in mappers are passed to each of the respective vertex. This can be achieved by emitting key-value pairs with the respective vertex identifier as the key and the intermediate partial result computed as the value. The sort and shuffle phase of MapReduce framework sorts the intermediate partial results. The reducer takes all the values corresponding to one of the key i.e. belonging to the single vertex and then aggregates all the values to get the final partial result for that vertex.

Algorithm 2 Reducer for Basic Implementation of Page-Rank

```
1: function REDUCER(id $m$, $[p_1, p_2, ..]$)
2:     $s \leftarrow 0$
3:     $M \leftarrow null$
4:     ListOfDestinationVertex $N \leftarrow null$
5:     for all $p \in p_1, p_2, .., p_n$ do
6:         if IsPair $(p)$ then
7:             $M \leftarrow p.from$
8:             $N.add(p.to)$
9:         else
10:            $s = s + p$
11:        end if
12:    end for
13:    $M$.PageRank $\leftarrow s$
14:    for all $n \in N$ do
15:        Emit $(M, n)$
16:    end for
17: end function
```

One of the important points to note is that, along with the intermediate partial result at a vertex is passed, then the edge associated with the same vertex as the source is also passed, with source vertex being the key and edge being the value. This step is necessary for the preservation of the graph topology. As a result, two types of messages are being passed from mapper to reducer, one is the partial computations for the vertex and the other is the incoming edges to the vertex. The second type of message passes the topology of the graph to reduce phase so that the graph can be updated.

Taking page-rank as an example, the pseudo-code for the basic implementation in the MapReduce framework is provided in Algorithm 1 (Mapper) and Algorithm 2 (Reducer). The pseudo-code does not take the damping factor and dangling nodes into account. The mapper maps over key-value pairs with source vertices acting as a key. It computes the page-rank contribution of the source vertex to the destination vertex and emits the destination vertex's id as the key and its corresponding fraction of page-rank as the value. In addition to this, mapper also emits source vertex's id as the key and the whole edge as the value to pass the graph structure. Here, the reducer gets the page-rank contribution from each of the incoming edge for a vertex and the graph topology associated with the vertex. In a single reducer task, 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.

4.1 Schimmy Design Pattern

The basic implementation of a graph algorithm in map-reduce framework 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. Lin and Schatz managed to reduce works per iteration by 69% using Hadoop framework and implemented design pattern called Schimmy for calculating PageRank on Carnegie Mellon University ClueWeb09 collection of web graph with 1.4 billion edges in [2]. Schimmy is a combination of the authors name, Schatz and Jimmy. The idea behind this Schimmy design pattern is message passing. The message passing design pattern is claimed to address issue in existing best practices for MapReduce graph algorithms that have significant shortcomings which limit performance, especially with respect to partitioning, serializing, and distributing the graph. Typically, such algorithms iterate some number of times, using graph state from the previous iteration as input to the next iteration, until some stopping criterion is met.

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 $T_1, \ldots, T_n$, respectively, based on their join key and by sorting each partition on the join key. Then, each pair S_i/T_i 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 3).

The implementation of the page-rank based on the Schimmy design does not need to shuffle the graph structure and hence the mapper remains the same as in Algorithm 1 but without line 2. In the reducer (Algorithm 3), 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 [2]

Algorithm 3 The Reducer for the Schimmy Implementation

```
1: function Initialize
2:     P.OpenGraphPartition()
3: end function

4: function Reduce(ID $m$, List $[p_1, \ldots, p_n]$)
5:     $s \leftarrow 0$
6:     for all $p \in [p_1, \ldots, p_n]$ do
7:         $s \leftarrow s + p$
8:     end for
9:     repeat
10:        $(from, to) \leftarrow$ P.Read()
11:        if $from.id \neq m$ then
12:            Emit($from$, $to$)
13:        else if $from.id = m$ then
14:            $from.pageRank \leftarrow s$
15:            Emit($from$, $to$)
16:        end if
17:    until $from.id > m$
18: end function
```

CHAPTER 5

Map-Based Graph Analysis

5.1 Methodology

The schimmy design pattern improved the efficiency of the implementation of the graph algorithms by removing the need to shuffle and sort the graph topology. But it still requires shuffling and sorting of the partially computed results and then sending it to the respective reducer for updating the internal states of the graph structures. As earlier stated, in the execution of MapReduce job, the network traffic dominates the computation time and also for sorting and shuffling, all the vertices or edges or both with their meta data needs to be shuffled and sorted to perform the single iteration. To remove the need for the sorting and shuffling of the graph and its partial results, we introduce a map-based design pattern for the analysis of the graph.

The notion of our idea is also parallel merge join as in the case of schimmy. In schimmy, the merge join happens between partition of a graph and the intermediate partial results sent for that partition but in our case, merge join is done between a partition of the graph and a global table storing all the partial results. As earlier described, the parallel merge join between two relations S and T can be done by partitioning both relations S and T, then sorting the partitions on the join key and sequentially parsing the two respective partitions of each relation S and T for joining. It can also be achieved by partitioning only one relations e.g. S into $S_1, S_2, S_3 ... S_n$ and then merge joining each of them with T in a parallel fashion. All the partitions

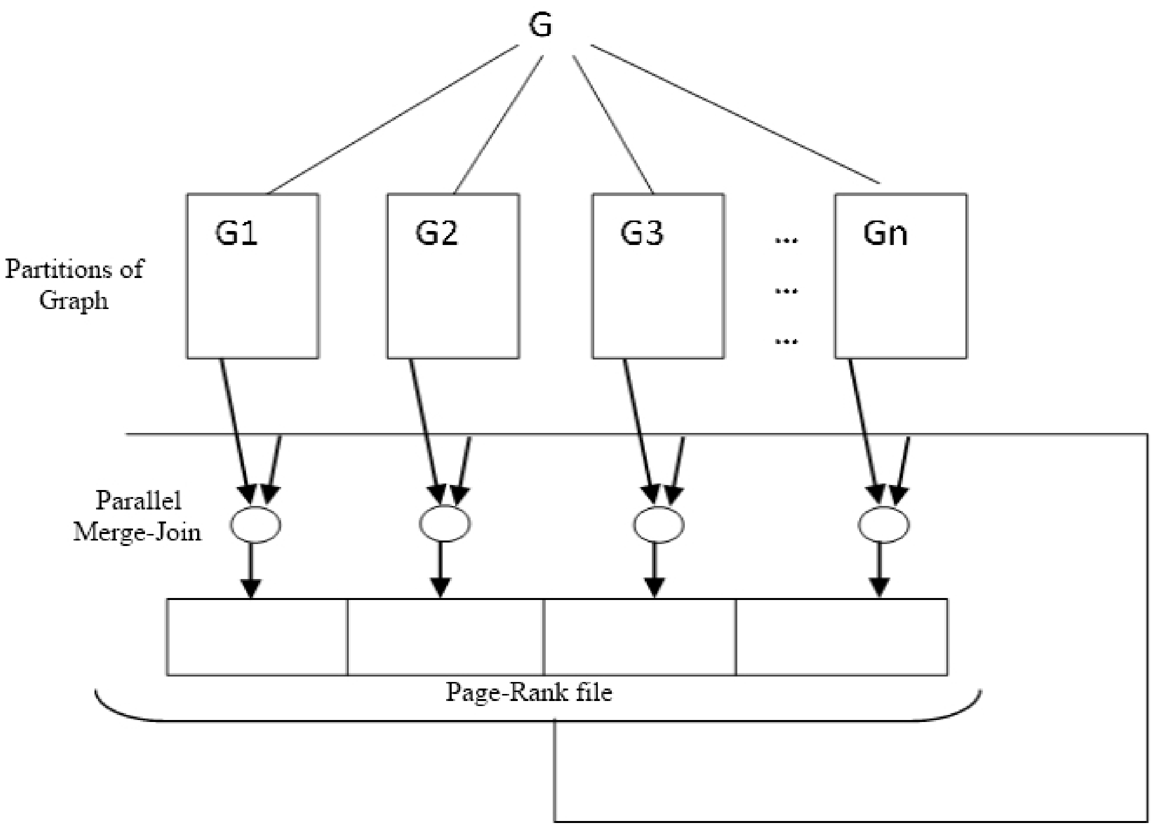

Figure 5.1. Page-Rank Computation using Parallel Merge Join..

$S_1, S_2, S_3 ... S_n$ and T are also sorted on same join key. Fig. 2 shows the notion of the idea

We perform the graph analysis by doing parallel merge join between the partition of graph and a global table where the partial results are stored. Taking page-rank as the example, we describe the approach as follows. Graph G is partitioned into G_1, G_2, G_3 in such a way that $G = G_1 \cup G_2 \cup G_3 \cup ... \cup G_n$ and partitioning is done on the edges such that no two partitions contain the edges with the same destination vertices i.e. the edges with the same destination vertices will be in one partition. Also, each of the partitions will be sorted on the source vertex of the edges. A global table in the form of flat binary file is initialized which will contain the partial results of each node after the end of iteration and also the global table is sorted over vertex's id. To analyze the graph, a merge join is done between the partitions of the graph and the global table during every iteration. The important point to note here is that graph is partitioned on the destination vertices of the edges and each partition is sorted on the source vertices of the edges.

It can be implemented in MapReduce as following. A MapReduce job, a pre processing step, is first made to partition the graph on the basis of destination vertices of each edge by making use of Partitioner class which is using the hash partitioning method on the destination vertices. We can also make sure that each of these partitions are sorted on source node of each edge by making use of Key Comparator class and Grouping Comparator class. Once all the partitions are made, they are saved on the file system. Also we initialize the page-rank of each of the node in the same MapReduce job through its reducer phase and save them in a sequence files. These files all together acts as the global table and hence are saved in the same directory.

Then a mapper is made to parse through each of the partitions and also it advances through the global page-rank table. Whenever it finds a edge with source vertex as same as the current vertex as it reads from the page-rank table, it will calculate the contribution of the page-rank from that source to the corresponding destination vertex of the edge. These contributions are aggregated together and saved in a dictionary. In the end this dictionary will contain the updated page-rank of the vertices belonging to that partition. The dictionary saves the vertices in the sorted manner and hence after calculating the page-rank, the dictionary is flushed out to the disk in the sequence file format to make a single section of the new global page-rank table which is used by next iteration. The mapper will not be writing anything to the disk other than the new page-rank values. This makes the single iteration of the page-rank. The pseudo-code of the mapper is shown in the Algorithm 5 .

Algorithm 4 Mapper for Map Based Parallel Merge Join for Computing Page-Rank

```
1: function INITIALIZE
2:     static id $n$
3:     static double $rank$
4:     P.OpenPageRankFile()
5:     Dictionary D $\leftarrow$ $null$
6:     ($n$, $rank$) $\leftarrow$ P.Read()
7: end function

8: function MAP(()$from$, $to$)
9:     if $from$.id $= n$ then
10:        D[to] $+=$ $rank/from$.NumOfOutLinks
11:    end if
12:    if $from \geq n$ then
13:        repeat
14:            (id $n$, value $rank$) $\leftarrow$ P.Read()
15:        until $from \geq n$
16:    end if
17: end function

18: function CLOSE
19:     P.WriteNewPageRank(D)
20: end function
```

Since, the MapReduce framework ensures that all the vertices are processed in the sorted order and we no longer need to pass the graph again as there is no change in the structure of the graph resulting in no change in the partitions of the graph. Hence once the partition is made by the pre-processing MapReduce job, there is no need to pass the graph again and make the partition. Also map phase of the job make sure the consistency of the global page-rank table i.e. if first section of the global page-rank table contains page-rank of the vertices from identifier 1 to 10, first section of all the other global page-rank table will contain the revised page-rank values of the same vertices. It should also be noted that to decrease the File Read/Write execution, we change the parameter for data replication in MapReduce to 1 while writing the global page-rank table and hence no replica of global page-rank table is created which decreases the HDFS write time.

5.2 Problems in Map Based Analysis

Our map-based page-rank algorithm given in Algorithm 5 represents a map in a map-reduce framework that applies to every edge $(from, to)$ of the graph G. In addition, T is the old page-rank (stored in an HDFS Sequence file) and D is the partition T_i' of the new page-rank table stored in a local Dictionary and dumped to HDFS at the end. Note that every worker W_i generates a single partition T_i' of the new page-rank table but reads the entire current page-rank table sequentially, one-partition-at-a-time. Since both the graph partition data and the page-rank files are sorted by the join key, and the key is unique in page-rank, the merge join never backtracks. The drawback of this work, which is resolved in the next section, is that each worker W_i must scan the entire page-rank table T simultaneously and in the same order (ie, all workers must read T_1 first, then T_2, etc). This makes the reading of the global table T the bottleneck of this approach.

5.2.1 Improvement of the Map-Based Graph Analysis

We have improved the parallel merge join to resolve the bottleneck problem that occurs when multiple workers try to scan the global table in the same way. More specifically, we have changed Algorithm 5 so that each worker node W_i will scan the global pagerank table starting from the ith partition, T_i. This is done by replacing Statement 5 with T.OpenPageRankPartition(i), where i is the worker number. In addition, each graph partition G_i is reorganized differently so that edges whose source nodes are joined with T_i will appear first. That way, the parallel merge is done in a round-robin fashion join across workers, such that, first W_1 reads T_1, W_2 reads T_2, $\ldots$, W_m reads T_m, then W_1 reads T_2, W_2 reads T_3, $\ldots$, W_m reads T_1, etc, until all workers read all T partitions. We have used two ways to partition the graph and global page rank tables: range partitioning and hash partitioning.

5.2.1.1 Range-Partitioning Method

We partition the graph in the same way, so that edges with the same destination go to the same partition, but sorting has been changed. The graph is again sorted on the source vertices. Graph G is partitioned into $G_1, \ldots, G_m$ on destination vertices of the edges so that all the edges to a destination goes to same partition¿ Each partition is then sorted on source vertices of the edges. This time the sorting depends on the partition number. We sort the graph partition, lets say containing destination vertices from m to n (m $\rightarrow$ n), in such a way that the first source vertex is m, then it goes till last vertex id after that it will again start with vertex id 1 till m-1. Note that, the graph partitions are sorted on source vertices and a partition of a graph may contain all the vertices of the graph being as the destination.

The global page-rank table is implemented in the same way as before containing vertex id, page-rank and out-degree of the respective vertex id and range partitioned

Algorithm 5 The Mapper for Map-Based Parallel Merge-Join for Computing Page-Rank

```
1: partition(id): the partition number of the vertex id

2: function INITIALIZE
3:     Dictionary D ← empty
4:     T.OpenPageRankPartition(0)
5:     ($n$, $rank$) ← T.Read()
6: end function

7: function MAP(Vertex $from$, Vertex $to$)
8:     repeat
9:         if !T.hasnext() and partition($n$) <partition($from.id$) then
10:            T.OpenPageRankPartition(partition($from.id$))
11:        end if
12:        if T.hasNext() then
13:            $(n, rank)$ ← T.Read()
14:        else
15:            continue
16:        end if
17:    until $n < from.id$
18:    if $from.id = n$ then
19:        D[$to.id$]$.rank += rank/from.numOfOutLinks$
20:    end if
21: end function

22: function CLOSE
23:     T.WritePageRankPartition(D)
24: end function
```

on the vertex ids. For the merge join, the first partition of graph starts with the first file of the global table and the second partition will start with the second file of the global table and last partition with the last file. As the sorting of the graph partition depends on the partition number, and partition of graph starts with source vertex same as the first vertex id found in the respective partition of the global page-rank table. Once every partition has completed the merge join with their respective file of the global table, they will move to the next one i.e. first partition of the graph will start merge join with the second file of the global table, second partition with the third file of the global table and last partition with the first file of the global table in a round robin fashion. In this way, we will be able to reduce the waiting time of a partition to get a part of global table.

5.2.1.2 Hash-Partitioning Method

As before, the graph G is hash-partitioned into $G_1, \ldots, G_m$ on the destination node of the edges. That is, an edge $(from, to)$ is sent to the partition G_i, where $i = \text{partition}(to.id)$. The edges $(from, to)$ in each partition G_i are also sorted by the following order:

- major order: $((\text{partition}(from.id) \bmod m) + i) \bmod m$
- minor order: $from.id$

That is, each G_i will contain m sub-partitions, so that the first sub-partition can be joined with T_i, the second with T_{i+1}, etc. As before, the global page-rank table T is hash-partitioned on the vertex id and each partition is sorted by the vertex id. As graph partitions are sorted individually on the source vertices in the same way as the global page-rank table, the merge join is done in the same way as in the range-partitioning method.

5.3 Graph Analysis on Updated Graphs

Most real world graphs are dynamic in nature, i.e., edges and nodes are getting deleted from and added to the graph frequently. Hence, a graph analysis has to be started from scratch every time the graph is updated, which repeats the computation on the parts of the graph that have not changed. But, many iterative analysis algorithms, including page-rank, have the property that the final data analysis results depend on the graph topology exclusively, so that they converge to the same result on different initial values. But these algorithms will converge to the final results with a certain accuracy faster if the initial values are closer to the final results. In this paper, we exploit this property by generating new results for the updated graph starting from the previous results.

We update the graph in streaming mode. We first collect a sufficient number of updates for the graph in an update file, then use this update file to update the graph topology and the global page-rank table, and then start the iterative graph algorithm to get the new results. An update to the graph can be one of the four types: 1. Insertion of a vertex 2. Deletion of a vertex 3. Insertion of an edge 4. Deletion of an edge. Two points should be noted: 1. Insertion of a vertex can only happen when there is an insertion of an edge with a new vertex. 2. Deletion of a vertex is always followed by the deletion of the one or more edges. For simplicity, we are considering addition and deletion of edges only. Addition and deletion of vertices is equivalent to addition and deletion of one or more edges. We are now describing the updates done to the graph and the global table.

The updates U being collected is a list of edges, with each edge having a flag that denotes whether the edge is to be deleted or added. The list U is being streamed and used to update the graph G and global page-rank table T. The stream is split into two different streams: 1. a Delete-Stream U_D and 2. an Add-Stream U_A. U_D

is collected over a time window and partitioned using the same partitioner that was used to partition the graph G into $G_1, \ldots, G_m$. Each partition of U_D is also sorted in the same way.

To delete the edges, each worker node W_i reads only one partition G_i and it joins it with the corresponding partition of U_{Di} using a parallel merge-join. More specifically, each worker W_i maintains a new updated partition of the graph G'_i in memory, which is dumped to HDFS at the end of an join step as a new partition G_i. At the end of the parallel merge-join, the respective edges will be deleted from the graph.

Each edge in the U_A is added to the end of the graph-partition. The graph-partition G_i in which U_A should be added is found using the partitioner that was used to partition the graph. Once all the edges from update files are sorted, each graph partitions $G_1, \ldots, G_m$ is sorted separately again.

The updates U are processed to compute the changes in the outdegree of the vertices. Once, the changes in outdegree are computed, these changes are merge-joined with the partitions of the global page-rank table T to updates the outdegrees and updating the global page-rank table in the same way as the U_A was added to G. The detailed algorithm to update the graph topology is given in Algorithm 6 and Algorithm 7 and the update of the global page-rank table is given in Algorithm 8.

Once the update process is completed, the iterative process is resumed to process the new updated data of the graph. Since a large part of the graph analysis task has already been completed on the existing data, the new updates require fewer iterations to compute the new graph analysis results since the iterative algorithm will converge faster. Figure 5.2 describes our incremental graph analysis approach using page-rank as an example.

Algorithm 6 The Streaming Function for Adding Edges to the Graph

```
partition($id$): the partition number of the vertex id
function INITIALIZE
    U.OpenUpdateFile()
    $U_A \leftarrow$ U.GetEdgeInsertions()
    (Vertex $from$, Vertex $to$) $\leftarrow U_A$.Read()
end function

function ADDEDGES(Vertex $from$, Vertex $to$)
    $partitionNo.$ $\leftarrow$ partition($to$)
    GraphPartition($partitionNo$).Write($from$, $to$)
end function

function CLOSE
    GraphPartition.Sort()
end function
```

Algorithm 7 The Streaming Function for Deleting Edges from the Graph

```
1: partition($id$): the partition number of the vertex id
2: function INITIALIZE
3:     U.OpenUpdateFile()
4:     List L $\leftarrow$ empty
5:     $U_D \leftarrow$ U.GetEdgeDeletions()
6:     (Vertex $from$, Vertex $to$) $\leftarrow U_D$.Read()
7:     $i \leftarrow$ partition($to$)
8:     $G_i$.OpenGraphPartition()
9: end function

10: function MAP(Vertex $from$, Vertex $to$)
11:     repeat
12:         if $G_i$.hasNext() then
13:             $from_G, to_G \leftarrow G_i$.Read()
14:             if $from_G$ , $to_G = from$ , $to$ then
15:                 continue
16:             end if
17:             L.Add($from_G$, $to_G$)
18:         end if
19:     until $from_G >= from$
20: end function

21: function CLOSE
22:     L.WriteGraphPartition(L)
23: end function
```

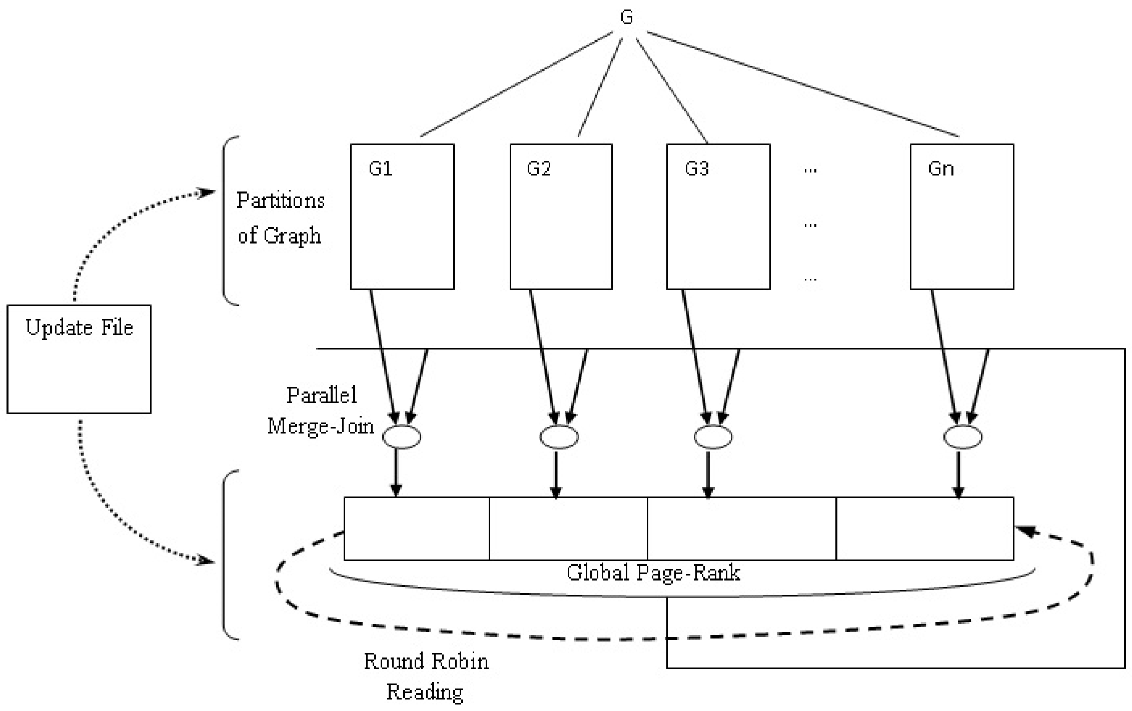

Figure 5.2. Incremental Page-Rank Computation using Parallel Merge Join.

Algorithm 8 The Streaming Function for Updating the Global Page Rank Table

```
1: partition($id$): the partition number of the vertex id
2: function INITIALIZE
3:     U.OpenUpdateFile()
4: end function

5: function MAPCOMPUTINGDEGREE(Vertex $from$, Vertex $to$)
6:     Emit($from$ , 1)
7: end function

8: function REDUCECOMPUTINGDEGREE(Vertex $from$, List $[c_1, \ldots, c_n]$)
9:     $degreeCount \leftarrow 0$
10:    for all $c \in [c_1, \ldots, c_n]$ do
11:        $degreeCount \leftarrow degreeCount + c$
12:    end for
13:    $partitionNo. \leftarrow$ partition($from$)
14:    GlobalPageRankPartition($partitionNo$).Write($from$, $degreeCount$)
15: end function

16: function CLOSE
17:    GlobalPageRankTable.ReduceByKey();
18: end function
```

CHAPTER 6

Experimentation

6.1 Graph Generators

Most of the time, real graphs are not available for experimentation. Industry and researchers use graph generators to create synthetic graphs which can then be used for experimentation and simulation. Synthetic graphs is considered to be similar to a real graph when the synthetic graph matches most of the patterns of the real graphs. These patterns are power law distribution of the degrees of the vertices of the graph, diameters of the graph and community structure inside the graph. Graph generators models are classified into five categories:

1. Random graph models: Random graphs are generated by randomly picking a pair of vertices and connecting them by edge with some random probability. The basic random graph model is the Erdos-Renyi Graph Model [Reference]. This was one of the first random graph model and the simplest model to generate a graph. Assuming a graph having N vertices, then an edge is added between a pair of node with a probability p. The distribution of the degrees of vertices of graph is poisson and hence it is called Poisson model. There are variants of Erdos-Renyi model that generates graph in power law distribution also.
2. Preferential attachment models: The rich gets richer in the network as it grows resulting in the power law effects. Most of the popular graph generators belong to this class. Random Graph Model try to model graph with different degree distributions but it doesn't take into account the processes which are actually generating the network. The search for a mechanism for network generation

was a motive for finding preferential attachment models. The basic preferential model starts with few vertices. The model grows the network by adding vertices over time. Each outgoing edge from the new vertex connects to an old vertex with a probability proportional to the in-degree of the old vertex. Initial degree of the vertices is 0 and hence a constant is added to the current degree of a node.

3. Geographical models: These models take the effects of position of vertices of graphs on topology of the graph. Such graphs are essential for the modeling router or power-grid networks. As the random graphs and preferential model do not take the constraints of the geography into account. It is easier to connect two routers which are physically close to each other. Similarly, in social network, people living close to each other has a high probability to connect to each other. The basic model starts with a ring network where each vertices has some k edges, $k/2$ edges on each side. For each node u, each of its edge (u, v) is rewired with probability p to form its some different edge (u, w) where vertex w is chosen randomly. The graph generated using this model has small diameter and high clustering coefficient which are characteristics of real graphs. These graphs do not follow power law distribution which is also characteristic of a real graph. In these graphs, degree distribution decays exponentially.
4. Internet-specific models: In computer science, the most important graph is the internet and hence specific models have been developed to model the special features of the internet. These are mostly hybrids using the other categories and matching them with the characteristics of the Internet.
5. Optimization-based model: Power laws are shown to evolve when risks are minimized using limited resources. These models work with the preferential

attachment model providing mechanisms that automatically results in power law effects.

Most of the current graph generators focus on only one graph pattern - typically the degree distribution - and give low importance to all the others. Then, there are problems in how to fit model parameters to match a given graph. Most of the graph generators has trade-offs between number of model parameters, matching different graph patterns and efficiency in generation speed. In the next section we describe the R-MAT generator which has few model parameters, can generate graphs matching different graph patterns and efficient also.

6.1.1 Recursive MATrix Graph Generator (RMAT)

The RMAT model for graph generation was introduced by Chakrabarti, Faloutsos, and Zhan [17]. The generator has an elegant design that is also very easy to implement. Additionally, RMAT can be implemented easily in distributed fashion and it is capable of quickly generating very large graphs. It has been shown that RMAT can generate graph power law degree distribution. Authors have also found the parameters for generating graphs which has the similar degree distribution as that of the real graphs. RMAT is a widely used graph generator due to its speed and simplicity. Graphs generated by RMAT have been used in the variety of research disciplines including graph theoretic benchmarks, social network analysis and network monitoring. We are using RMAT graph for our detailed experimental analysis of our design pattern.

We first describe the methodology for the generating graph using RMAT. The graph is defined as $G(V, E)$ where V is a set of vertices and E is a set of directed edges. Supposing, the graph has n vertices, then the adjacency matrix A of the graph

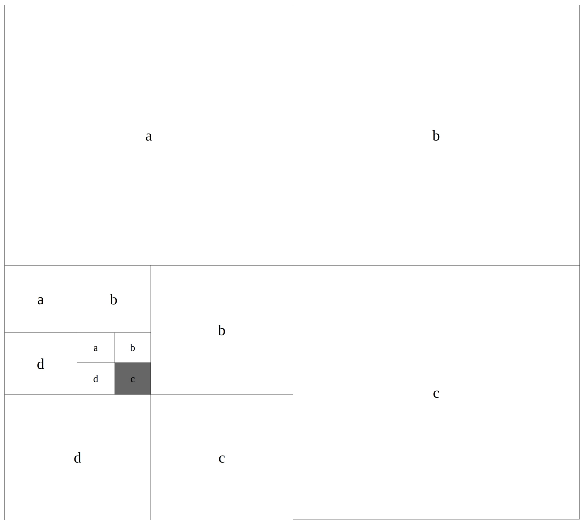

Figure 6.1. Edge Generation from RMAT algorithm.

is a $n \times n$ matrix with positive entry A_{ij} for an edge from vertex i to vertex j. $A_{ij} = k$ means the graph has k edges from i to j.

The RMAT model generates graph by recursively subdividing the adjacency matrix of a directed graph into four equally sized partitions and distributing M edges within these partitions with unequal probabilities. The distribution is determined by four non-negative parameters a, b, c and d such that $a + b + c + d = 1$. Starting with $a_{ij} = 0$ for all $0 \leq i, j < leq(n1)$, the algorithm places an edge in the matrix by choosing one of the four partitions with probability $a, b, cord$ respectively. The chosen quadrant is then subdivided into four smaller partitions, and the procedure repeated until we have selected a 1×1 partition, where we increment that entry of the adjacency matrix by one. For example, in Figure 6.1, we recursively partition the matrix five times before arriving at the shaded 1×1 partition.

RMAT generates the communities in the graphs. Typically, $a \geq b, a \geq c, a \geq d$.

1. The partitions a and d represents separate group of vertices which can be seens as separate communities.

2. The partitions b and c are the cross links between these two groups and edges between them can be seen as friends with separate interests.
3. The recursive nature of the partition can automatically generate sub-communities within the existing communities.

Parameters, $a, bandc$, of RMAT model can be computed from the out-degree and in-degree distribution. It turns out, for most of the real graphs, $a : b$ and $a : c$ ratios are approximately $75 : 25$. From this, the parameters can be computed.

RMAT can generate different kinds of graph with different parameter values $(a, b, candd)$. We generated a graph that resembles a real graph having communities and sub-communities and having power law distribution. Other graph we generated is the Erdos-Renyi Graph.

The parameter values for the modeling real graph are $a = 0.57, b = 0.19andc = 0.19$ and parameter values for modeling Erdos-Renyi Graph are $a = b = c = d = 0.25$. We generated graphs of 50 million, 75 million and 100 million edges. We have also generated graph with 600 thousand vertices and 7.5 million edges to compare it with the stanford web graph. Similarly, a graph of 875000 vertices and 5 million edges to compare it with google web graph. Both the real graphs, stanford web graph and google web graph are collected from SNAP datasets [18]

6.2 MRQL

MRQL [19] (pronounced miracle) is a query processing and optimization system for large-scale, distributed data analysis. MRQL (the MapReduce Query Language) is an SQL-like query language for large-scale data analysis on a cluster of computers. The MRQL query processing system can evaluate MRQL queries in four modes:

1. Map-Reduce mode using Apache Hadoop
2. BSP mode (Bulk Synchronous Parallel mode) using Apache Hama

3. Spark mode using Apache Spark
4. Flink mode using Apache Flink

The MRQL query language is powerful enough to express most common data analysis tasks over many forms of raw in-situ data, such as XML and JSON documents, binary files, and CSV documents. MRQL is more powerful than other current high-level MapReduce languages, such as Hive and PigLatin, since it can operate on more complex data and supports more powerful query constructs, thus eliminating the need for using explicit MapReduce code. With MRQL, users are able to express complex data analysis tasks, such as PageRank, k-means clustering, matrix factorization, etc, using SQL-like queries exclusively, while the MRQL query processing system is able to compile these queries to efficient Java code.

We used MRQL for generating synthetic huge graphs for our experiments. These graphs are generated by R-MAT algorithm [17] using the different parameters for different different types of graphs. We generated two different types of graphs, Erdos-Renyi Graphs and Kronecker graphs. To generate kronecker graphs, we used parameters $a = 0.59$, $b = 0.19$ and $d = 0.05$. To generate Erdos-Renyi graph, we used parameters $a = b = c = d = 0.25$. The graphs with following configurations are generated for big data evaluation:

1. 500,000 vertices and 50 million edges (approximately 650 MB in size).
2. 750,000 vertices and 75 million edges (approximately 1.1 GB in size)
3. 1,000,000 vertices and 100 million edges (approximately 1.5 GB in size)

The file generated is represented as a flat list of edges stored in the text format.

6.3 Real Graphs

We have also used real graphs for evaluation of our design pattern. We used the following graphs:

1. Stanford Web Graph (685230 vertices, 7600595 edges)
2. Google Web Graph (875700 vertices, 5105039 edges)

Both of the real graphs were downloaded from the snap datasets [18]. We have also generated kronecker and erdos-renyi graphs of equal sizes for comparison of our design pattern between synthetic and real graphs. So following synthetic graphs were also generated

1. 600,000 vertices, 7,500,000 edges kronecker and erdos-renyi to compare the performance of design pattern with that of stanford web graph
2. 875,700 vertices, 5,105,039 edges Kronecker and erdos-renyi to compare the performance of design pattern with that of google web graph

6.4 Evaluation on Real Graphs

Figure 6.2 shows the comparison between the time taken to pre-process stanford graph, erdos-renyi graph and kronecker graph of same size using range partitioning method. It can be observed from the figure that time to pre-process the stanford web graph is high as compared to pre-process the erdos-renyi or kronecker graph because partitioning of stanford graph is skewed and hence, some all the edges are in the first few paritions only making them bigger in size than the rest of the partitions. So the most time was taken to write the bigger partitions. Erdos-Renyi graph and Kronecker graph are partitioned properly making their partitions approximately equal in size leading to proper distribution of data and hence time taken to complete the pre-processing of erdos-renyi and kronecker graphs is less.

Figure 6.3 shows the comparison between the time taken to pre-process stanford graph, erdos-renyi graph and kronecker graph of same size using range partitioning method. It shows the same behavior as that in 6.2 because the partitioning of google

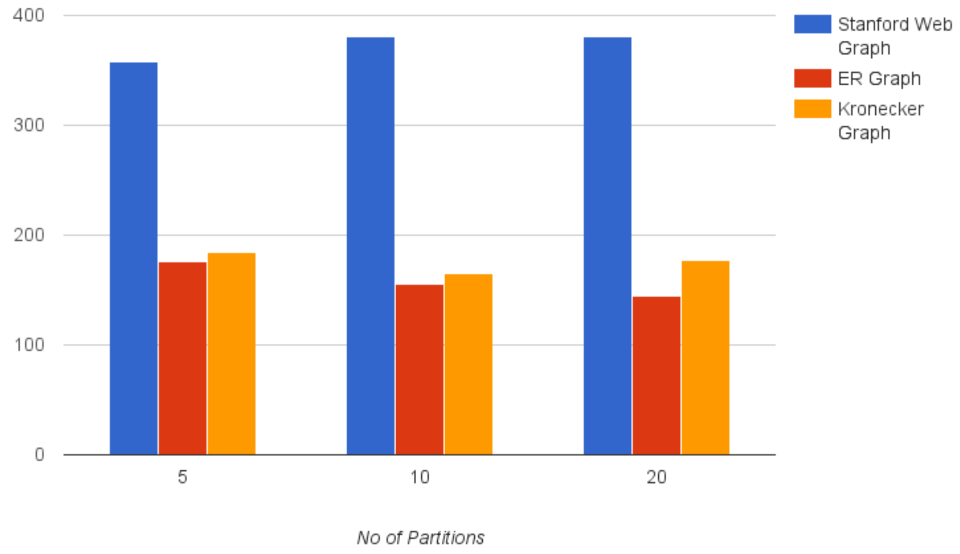

Figure 6.2. Evaluation of pre-processing step using range partitioning method on Stanford Graph, Erdos-Renyi Graph and Kronecker Graph.

graph is skewed and the partitioning of erdos-renyi and kronecker graphs produced approximately equal partitions.

Figure 6.4 shows the comparison between the time taken to pre-process stanford graph, erdos-renyi graph and kronecker graph of same size using hash partitioning method. This partitioned the graph with approximately equal sizes. As the sizes of the parition was not big enough when increasing the number of partitions, they were written to distributed file system in less time.

Figure 6.5 shows the comparison between the time taken to pre-process google graph, erdos-renyi graph and kronecker graph of same size using hash partitioning method. Skewed partitioning was observed when using hash partitioning using small number of partitions resulting in few big partitions which took most of the time to get

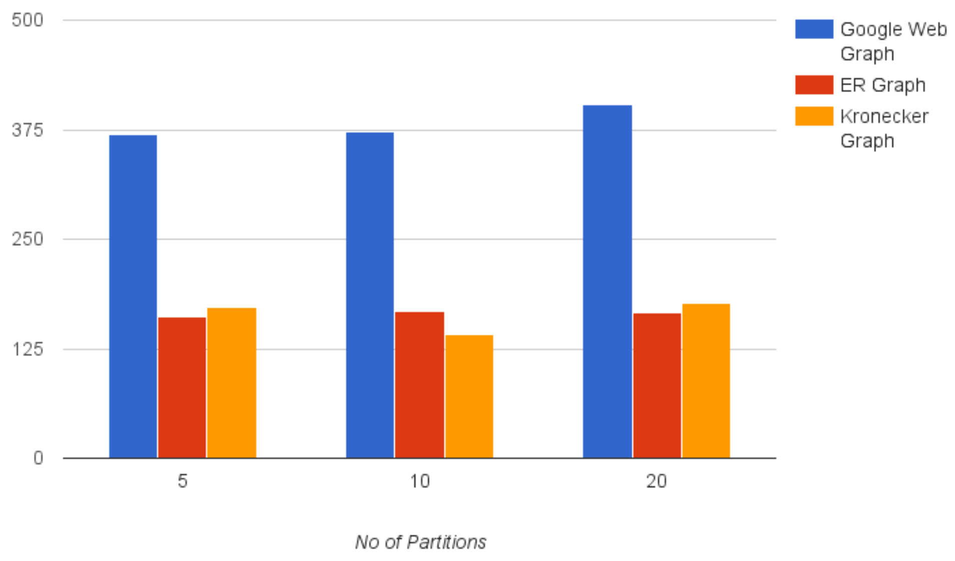

Figure 6.3. Evaluation of pre-processing step using range partitioning method on Google Graph, Erdos-Renyi Graph and Kronecker Graph.

written back to distributed file system. Hash partitioning method worked best when number of partition = 10. When number of partition = 20, it took more time because the cluster was waiting for the computer nodes to be free to write the partitions. As we were performing evaluation on 12 node cluster, at the time only 12 partitions were being written to distributed file system.

Figure 6.6 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over stanford graph, erdos-renyi graph and kronecker graph partitioned using range partitioning method. Merge-Join was faster stanford graph because all the edges and nodes are their in first few partitions only. Most of the merge-joined was finished in first few partitions only and there was no need to read all the partitions of global table which contains the page-rank and

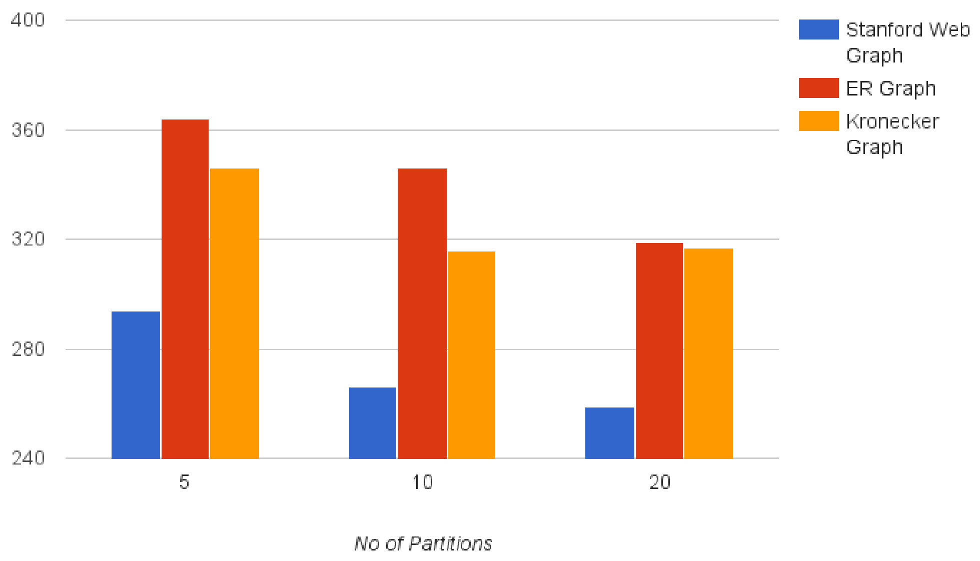

Figure 6.4. Evaluation of pre-processing step using hash partitioning method on Stanford Graph, Erdos-Renyi Graph and Kronecker Graph.

characteristics of the graph vertices. When number of partitions = 20, the time increased because paritions were waiting for the computer nodes to be free to start processing them.

Figure 6.7 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over google graph, erdos-renyi graph and kronecker graph partitioned using range partitioning method. It shows the decreasing trend for kronecker and erdos renyi graph as they were partitioned in partitions of approximately equal sizes but in case of google graph, the partitioning was skewed has resulting in high time. Erdos-renyi graph took more time than kronecker graph because the first few partitions of erdos-renyi are comparatively bigger than the rest of them when compared to the partitioning of kronecker graphs.

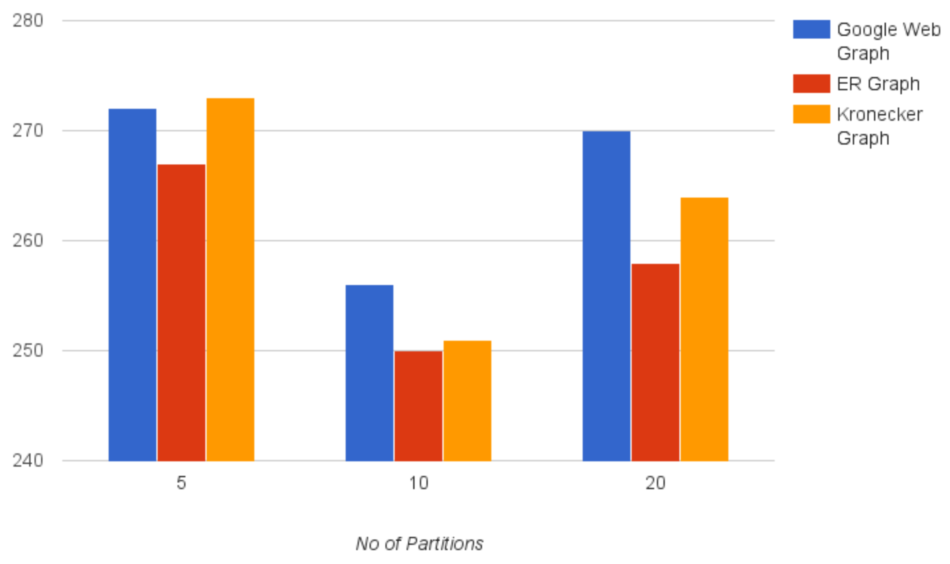

Figure 6.5. Evaluation of pre-processing step using hash partitioning method on Google Graph, Erdos-Renyi Graph and Kronecker Graph.

Figure 6.8 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over stanford graph, erdos-renyi graph and kronecker graph partitioned using hash partitioning method. As hash partitioning produced better partitions for stanford graph as compared to the erdos-renyi and kronecker graphs. It took less time. While incresing the number of partitions, the time increased because time to complete the merge-join between a partition of graph and partition of graph table was more than the time to communicate a partition of graph table among the computer nodes. In other words, for small graph, the communication time is more than the actual computation time resulting in higher execution time when increasing the number of partitions.

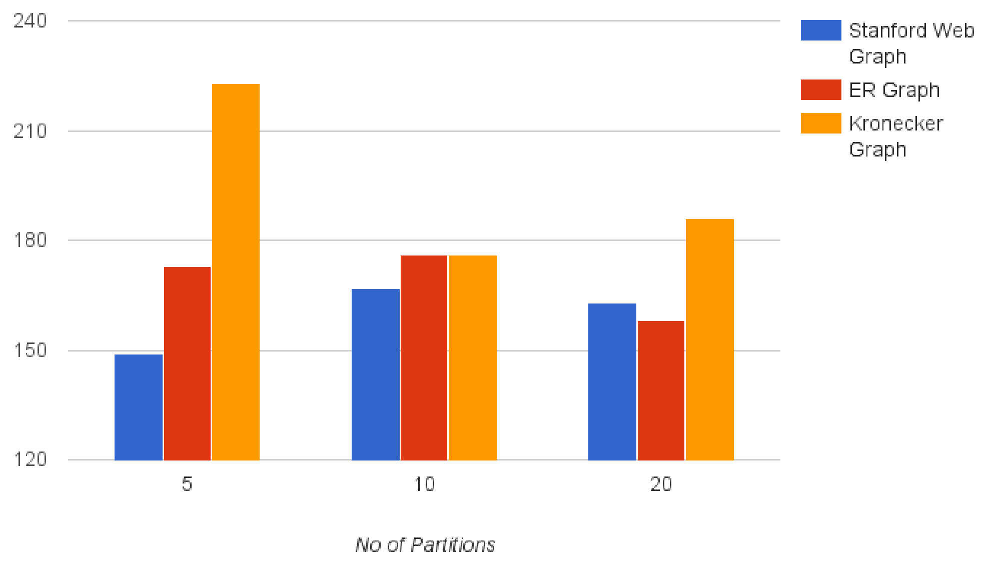

Figure 6.6. Evaluation of 5 iterations of page-rank algorithm using range partitioning method on Stanford Graph, Erdos-Renyi Graph and Kronecker Graph.

Figure 6.8 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over google graph, erdos-renyi graph and kronecker graph partitioned using hash partitioning method. It follows the same trend as shown in previous chart 6.8.

6.4.1 Evaluation on Big Synthetic Graphs

We evaluated the efficiency of the map-reduce optimizations by computing 5 iterations of the page-rank algorithm. We first compared the time taken to perform the page-rank iterations using the basic, schimmy and our map-based implementations. We used the hash-partitioning method to partition the graph for each of the methods. The computation time for 5 iterations of the page-rank algorithm for different graphs.

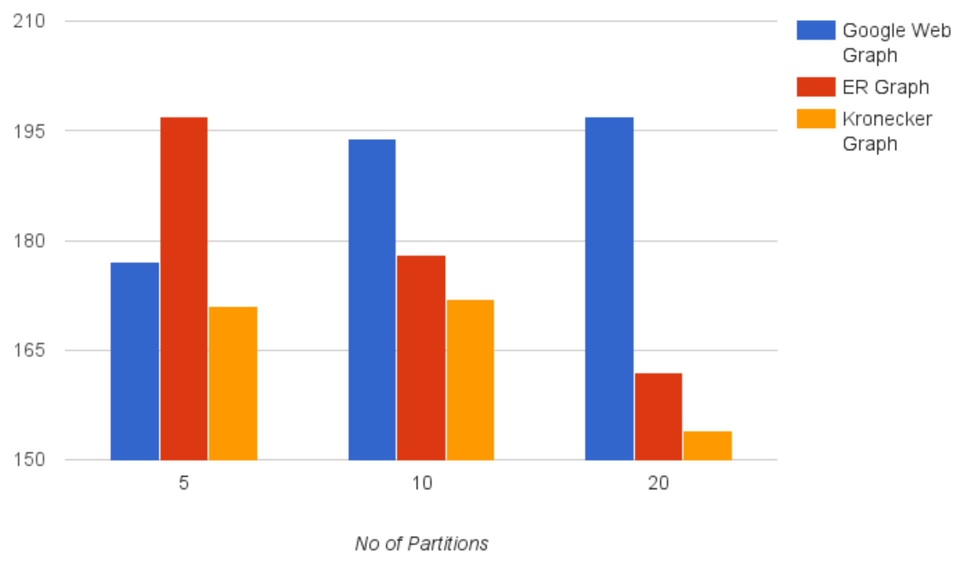

Figure 6.7. Evaluation of 5 iterations of page-rank algorithm using range partitioning method on Google Graph, Erdos-Renyi Graph and Kronecker Graph.

To show the difference in timings, we generated the synthetic large graph of size 7GB, 14GB and 28 GB. They contain 500 million, 1 billion and 2 billion edges respectively and all of them have 15 million vertices.

For the basic and schimmy implementations, there is only one preprocessing step, but for our map-based implementation there are two preprocessing steps: one to compute the metadata information and the other to initialize the first global table with initial page-rank values for every vertex of the graph.

Our Map-Based approach improves the performance of graph-analysis over the basic approach as well as the Schimmy approach. As our appraoch separates the immutable graph topology from the graph analysis results, there is no shuffling and sorting phase and hence our approach improves the performance of the graph analysis.

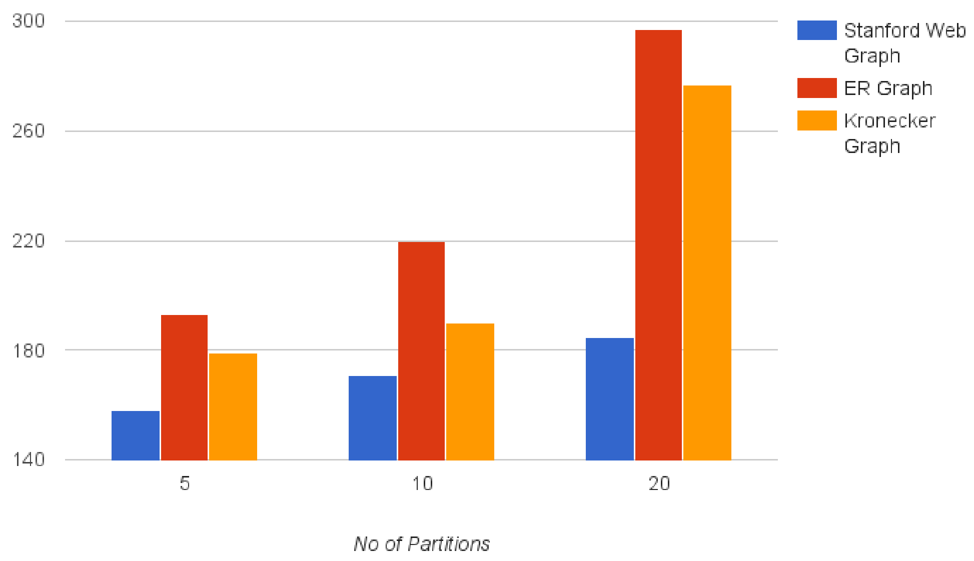

Figure 6.8. Evaluation of 5 iterations of page-rank algorithm using hash partitioning method on Stanford Graph, Erdos-Renyi Graph and Kronecker Graph.

There has been an improvement of around 10% over the schimmy based approach. It should be noted that our approach is applicable to a general class of graph algorithms, as discussed in 3.1. It should also be noted that the page-rank is computed for each vertex of the graph and hence, it is an exhaustive graph analysis. Graph analysis other than page-rank may be less exhaustive and hence can benefit more from our approach.

Now we will compare our new approach with the our earlier approach. Figure 6.11 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over kronecker graphs of different sizes. As stated in 6.2, we generated three big graphs for big data evaluation. It can be observed that our hash partitioning method worked better than as compared to range partitioning and pre-

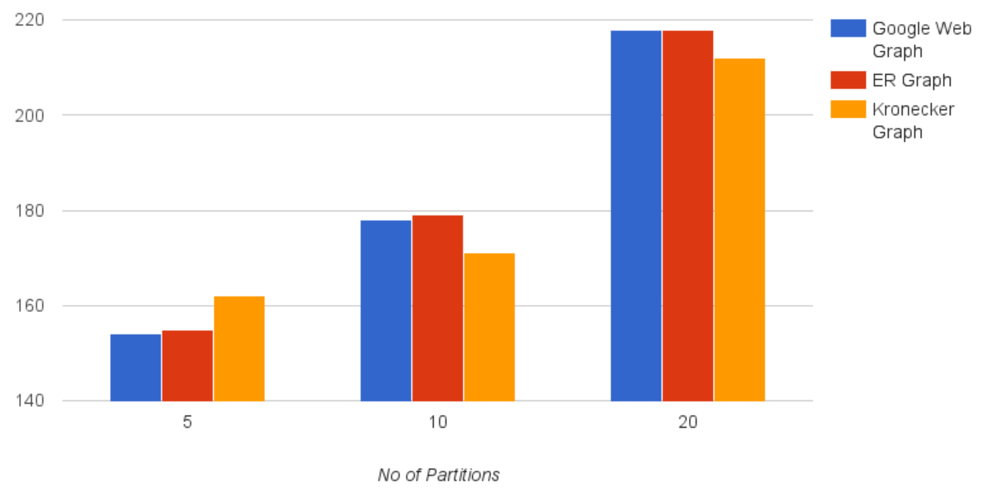

Figure 6.9. Evaluation of 5 iterations of page-rank algorithm using hash partitioning method on Stanford Graph, Erdos-Renyi Graph and Kronecker Graph.

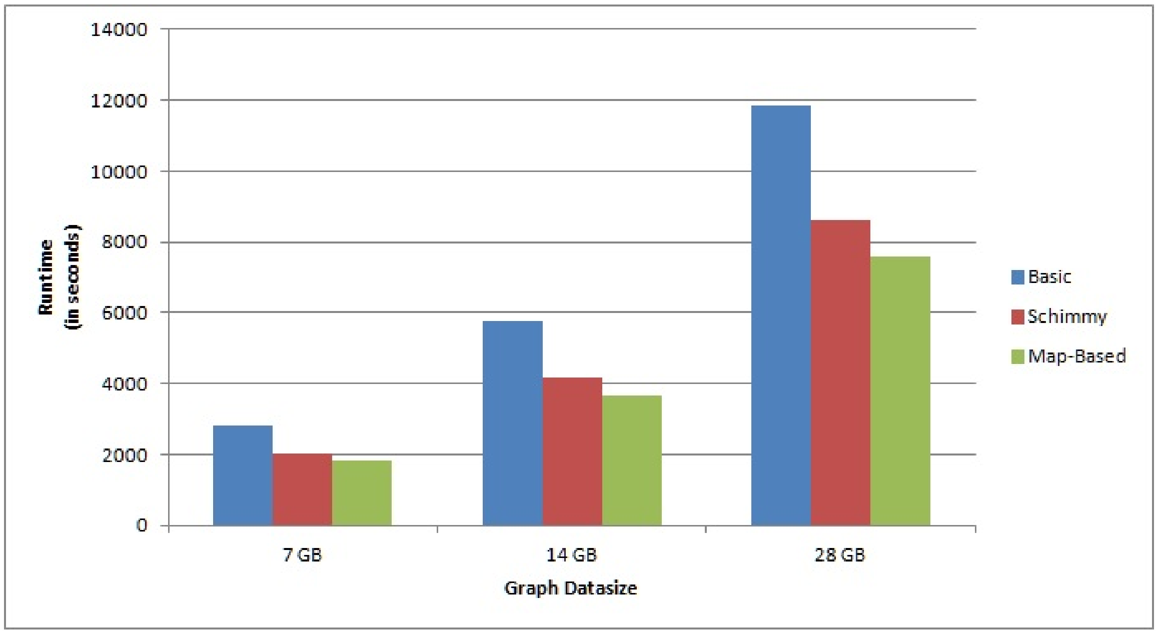

Figure 6.10. Evaluation of Earlier Design Patterns on a Synthetic Graph.

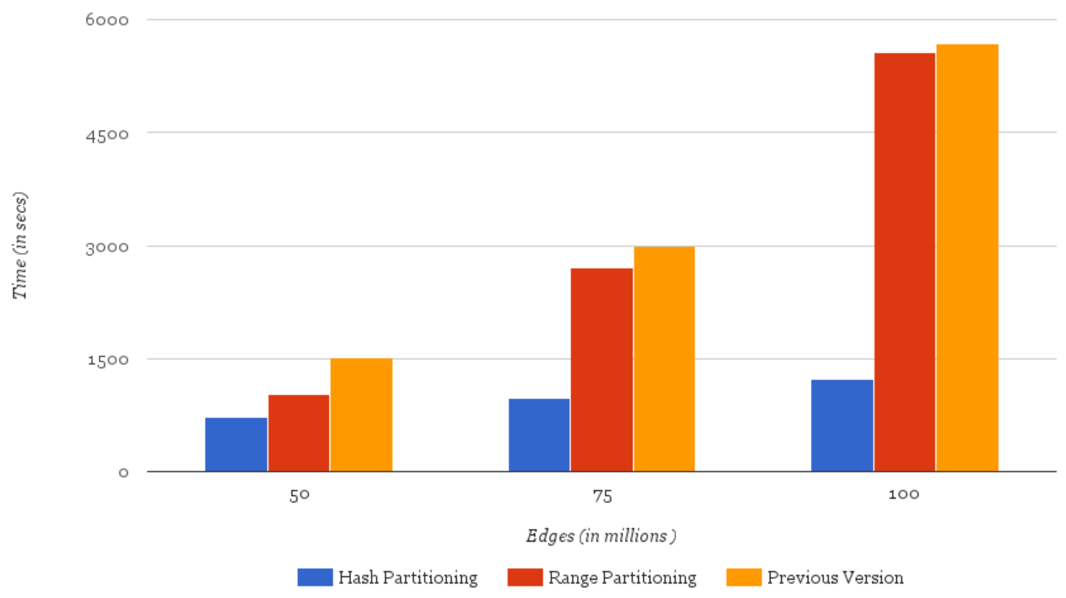

Figure 6.11. Evaluation of various Design Patterns on Kronecker Graphs.

vious design pattern using hash partitioning method. Our hash partitioning method worked better than range partitioning method because hash partitioning produced partitions of approximately equal number of sizes whereas range partitioning method produced skewed partitions. Previous hash-partitioning method took more time because every computer node was waiting for the first computer node finish reading the first partition of graph table.

Figure 6.12 shows the comparison between the time taken to compute the 5 iterations of the page-rank algorithm over erdos-renyi graphs of different sizes. It showed the same trend as in 6.11 except the range partitioning worked better than the hash partitioning as range partitioning method produced better partitions as compared to case in 6.11 and previous hash-partitioning method took more time

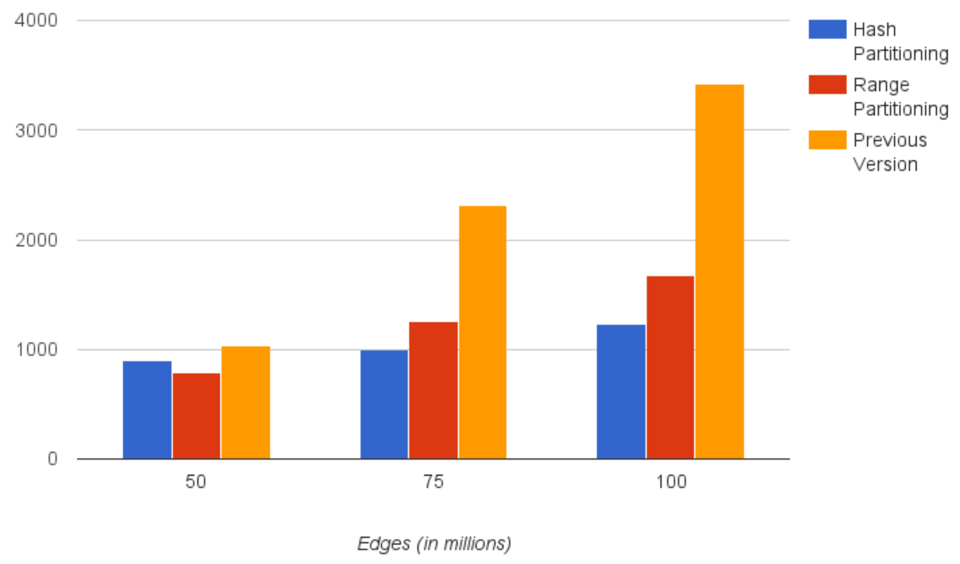

Figure 6.12. Evaluation of various Design Patterns on Erdos-Renyi Graphs.

because every computer node was waiting for the first computer node finish reading the first partition of graph table.

6.4.2 Incremental Graph Analysis

Figure 6.13 shows the comparison between the time taken to converge the page-rank algorithm over erdos-renyi graph and kronecker graph. The 50 million edge graph was divided into 40 million edges (lets call it base graph) and 10 million edges (lets call it update graph). We computed the page-rank of 40 million edges graph. We also added more 10 million edges randomly picked from the base graph to update graph flagged as for deletion. Now we updated base graph with 2 million, 4 million, 6 million, 8 million and 10 million edges one at a time, compute the time taken to converge the page rank algorithm. It can be observed from the graph that it takes

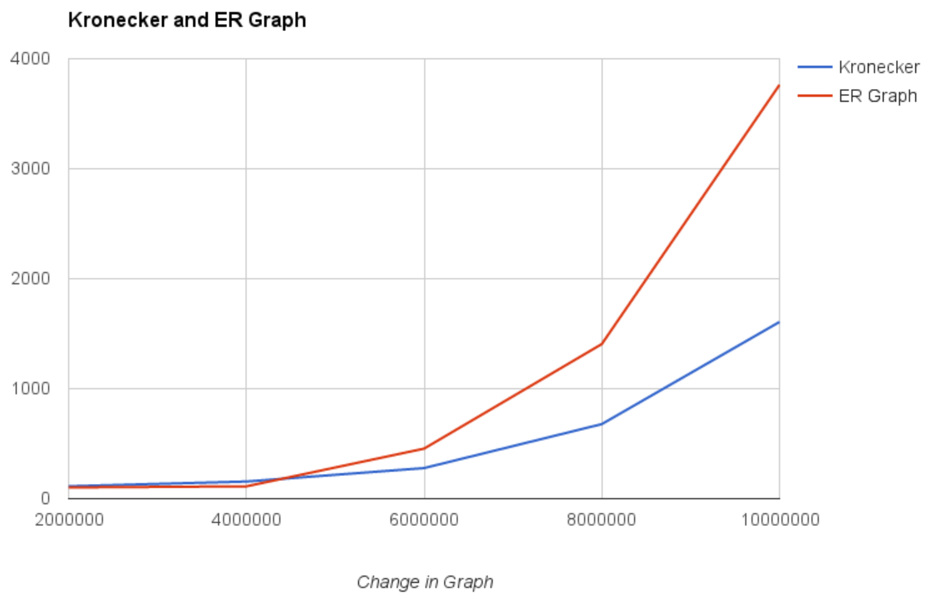

Figure 6.13. Distributed Incremental Page-Rank Evaluation.

less time for the page-algorithm to converge when doing small updates as compared to large updates. It is because more the number of updates, more the changes will be propagated through the graph more it will take time to converge

CHAPTER 7

Conclusion

Graph analysis in a distributed frameworks, such as Map-Reduce, is a challenge. There has been approaches for the analysis of graph algorithms, but most of these approaches has a high communication cost because of the shuffling and sorting phases of the map-reduce. Our approach detaches the immutable graph topology from the analysis and as a result we get an improved performance as there is less communication cost.

Also, batch graph analysis becomes very expensive for a graph that changes frequently because it repeats computations on the unchanged graph data. To avoid repeating computation, we need new graph processing methods that can analyze a graph incrementally and can update the graph results based on the changes in the graph. We have introduced a novel design pattern for incremental graph analysis that converges faster to a solution using special partitioning and sorting techniques to reduce data shuffling, and using a merge-join to combine the current graph results with the new. In the future, we are planning to use a key-value map to store the page-rank table, so that merging the new page-ranks with the existing ones can be done by updating the key-value map.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," vol. 51, no. 1, Jan. 2008, pp. 107–113.

[2] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, ser. MLG '10, 2010, pp. 78–85.

[3] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, 2010, pp. 810–818.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," vol. 3, no. 1-2, Sep. 2010.

[5] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[6] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10, 2010, pp. 135–146.

[7] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10, 2010, pp. 721–726.

[8] "Giraph,"
http://incubator.apache.org/giraph/,.

[9] U. Gupta and L. Fegaras, "Map-based graph analysis on mapreduce," in *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, 2013, pp. 24–30.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10, 2010, pp. 10–10.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 423–438.

[12] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI*, 2010, pp. 251–264.

[13] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online." in *Nsdi*, vol. 10, no. 4, 2010, p. 20.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1999.

[16] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[17] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *In SDM*, 2004.

[18] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[19] "Mrql,"
http://lambda.uta.edu/mrql/,.

BIOGRAPHICAL STATEMENT