Parallel Data Processing with MapReduce: A Survey

Kyong-Ha Lee
Yoon-Joon Lee
Department of Computer Science
KAIST
bart7449@gmail.com
yoonjoon.lee@kaist.ac.kr

Hyunsik Choi
Yon Dohn Chung
Department of Computer Science and Engineering
Korea University
hyunsik.choi@korea.ac.kr
ydchung@korea.ac.kr

Bongki Moon
Department of Computer Science
University of Arizona
bkmoon@cs.arizona.edu

ABSTRACT

A prominent parallel data processing tool MapReduce is gaining significant momentum from both industry and academia as the volume of data to analyze grows rapidly. While MapReduce is used in many areas where massive data analysis is required, there are still debates on its performance, efficiency per node, and simple abstraction. This survey intends to assist the database and open source communities in understanding various technical aspects of the MapReduce framework. In this survey, we characterize the MapReduce framework and discuss its inherent pros and cons. We then introduce its optimization strategies reported in the recent literature. We also discuss the open issues and challenges raised on parallel data analysis with MapReduce.

1. INTRODUCTION

In this age of data explosion, parallel processing is essential to processing a massive volume of data in a timely manner. MapReduce, which has been popularized by Google, is a scalable and fault-tolerant data processing tool that enables to process a massive volume of data in parallel with many low-end computing nodes[44, 38]. By virtue of its simplicity, scalability, and fault-tolerance, MapReduce is becoming ubiquitous, gaining significant momentum from both industry and academia. However, MapReduce has inherent limitations on its performance and efficiency. Therefore, many studies have endeavored to overcome the limitations of the MapReduce framework[10, 15, 51, 32, 23].

The goal of this survey is to provide a timely remark on the status of MapReduce studies and related work focusing on the current research aimed at improving and enhancing the MapReduce framework. We give an overview of major approaches and classify them with respect to their strategies. The rest of the survey is organized as follows. Section 2 reviews the architecture and the key concepts of MapReduce. Section 3 discusses the inherent pros and cons of MapReduce. Section 4 presents the classification and details of recent approaches to improving the MapReduce framework. In Section 5 and 6, we overview major application domains where the MapReduce framework is adopted and discuss open issues and challenges. Finally, Section 7 concludes this survey.

2. ARCHITECTURE

MapReduce is a programming model as well as a framework that supports the model. The main idea of the MapReduce model is to hide details of parallel execution and allow users to focus only on data processing strategies. The MapReduce model consists of two primitive functions: *Map* and *Reduce*. The input for MapReduce is a list of ($key1$, $value1$) pairs and `Map()` is applied to each pair to compute intermediate key-value pairs, ($key2$, $value2$). The intermediate key-value pairs are then grouped together on the key-equality basis, *i.e.* ($key2$, $list(value2)$). For each $key2$, `Reduce()` works on the list of all values, then produces zero or more aggregated results. Users can define the `Map()` and `Reduce()` functions however they want the MapReduce framework works.

MapReduce utilizes the Google File System(GFS) as an underlying storage layer to read input and store output[59]. GFS is a chunk-based distributed file system that supports fault-tolerance by data partitioning and replication. Apache Hadoop is an open-source Java implementation of MapReduce[81]. We proceed our explanation with Hadoop since Google's MapReduce code is not available to the public for its proprietary use. Other implementations (such as DISCO written in Erlang[6]) are also available, but not as popular as Hadoop. Like MapReduce, Hadoop consists of two layers: a data storage layer called Hadoop DFS(HDFS) and a data processing layer called Hadoop MapReduce Framework. HDFS is a block-structured file system managed by a single master node like Google's GFS. Each processing job in Hadoop is broken down to as many Map tasks as input data blocks and one or more Reduce tasks. Figure 1 illustrates an overview of the Hadoop architecture.

A single MapReduce(MR) job is performed in two phases: Map and Reduce stages. The master picks idle

workers and assigns each one a map or a reduce task according to the stage. Before starting the Map task, an input file is loaded on the distributed file system. At loading, the file is partitioned into multiple data blocks which have the same size, typically 64MB, and each block is *triplicated* to guarantee fault-tolerance. Each block is then assigned to a mapper, a worker which is assigned a map task, and the mapper applies `Map()` to each record in the data block. The intermediate outputs produced by the mappers are then sorted locally for grouping key-value pairs sharing the same key. After local sort, `Combine()` is optionally applied to perform pre-aggregation on the grouped key-value pairs so that the communication cost taken to transfer all the intermediate outputs to reducers is minimized. Then the mapped outputs are stored in local disks of the mappers, partitioned into R, where R is the number of Reduce tasks in the MR job. This partitioning is basically done by a hash function *e.g.*, `hash(key) mod R`.

When all Map tasks are completed, the MapReduce scheduler assigns Reduce tasks to workers. The intermediate results are shuffled and assigned to reducers via HTTPS protocol. Since all mapped outputs are already partitioned and stored in local disks, each reducer performs the shuffling by simply pulling its partition of the mapped outputs from mappers. Basically, each record of the mapped outputs is assigned to only a single reducer by *one-to-one shuffling* strategy. Note that this data transfer is performed by reducers' pulling intermediate results. A reducer reads the intermediate results and merge them by the intermediate keys, *i.e. key*2, so that all values of the same key are grouped together. This grouping is done by *external merge-sort*. Then each reducer applies `Reduce()` to the intermediate values for each *key*2 it encounters. The output of reducers are stored and triplicated in HDFS.

Note that the number of Map tasks does not depend on the number of nodes, but the number of input blocks. Each block is assigned to a single Map task. However, all Map tasks do not need to be executed simultaneously and neither are Reduce tasks. For example, if an input is broken down into 400 blocks and there are 40 mappers in a cluster, the number of map tasks are 400 and the map tasks are executed through 10 waves of task runs. This behavior pattern is also reported in [60].

The MapReduce framework executes its tasks based on *runtime scheduling scheme*. It means that MapReduce does not build any execution plan that specifies which tasks will run on which nodes before execution. While DBMS generates a query plan tree for execution, a plan for executions in MapReduce is determined entirely at runtime. With the runtime scheduling, MapReduce achieves fault tolerance by detecting failures and reassigning tasks of failed nodes to other healthy nodes in the cluster. Nodes which have completed their tasks

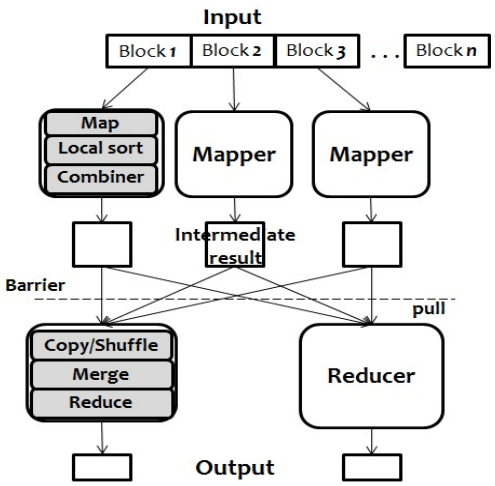

Figure 1: Hadoop Architecture

are assigned another input block. This scheme naturally achieves load balancing in that faster nodes will process more input chunks and slower nodes process less inputs in the next wave of execution. Furthermore, MapReduce scheduler utilizes a speculative and redundant execution. Tasks on straggling nodes are redundantly executed on other idle nodes that have finished their assigned tasks, although the tasks are not guaranteed to end earlier on the new assigned nodes than on the straggling nodes. Map and Reduce tasks are executed with no communication between other tasks. Thus, there is no contention arisen by synchronization and no communication cost between tasks during a MR job execution.

3. PROS AND CONS

3.1 Debates

As suggested by many researchers, commercial DBMSs have adopted "one size fits all" strategy and are not suited for solving extremely large scale data processing tasks. There has been a demand for special-purpose data processing tools that are tailored for such problems [79, 50, 72]. While MapReduce is referred to as a new way of processing big data in data-center computing [77], it is also criticized as a "major step backwards" in parallel data processing in comparison with DBMS [10, 15]. However, many MapReduce proponents in industry argue that MapReduce is not a DBMS and such an apple-to-orange comparison is unfair. As the technical debate continued, ACM recently invited both sides in January edition of CACM, 2010 [51, 39]. Panels in DOLAP'10 also discussed pros and cons of MapReduce and relational DB for data warehousing [23].

Pavlo *et al*'s comparison show that Hadoop is 2∼50 times slower than parallel DBMS except in the case of

data loading [15]. Anderson *et al* also criticize that the current Hadoop system is scalable, but achieves very low efficiency per node, less than 5MB/s processing rates, repeating a mistake that previous studies on high-performance systems often made by "focusing on scalability but missing efficiency" [32]. This poor efficiency involves many issues such as performance, total cost of ownership(TCO) and energy. Although Hadoop won the 1st position in GraySort benchmark test for 100 TB sorting(1 trillion 100-byte records) in 2009, its winning was achieved with over 3,800 nodes [76]. MapReduce or Hadoop would not be a cheap solution if the cost for constructing and maintaining a cluster of that size was considered. Other studies on the performance of Hadoop are also found in literature [28, 61]. Analysis of 10-months of MR logs from Yahoo's M45 Hadoop cluster and MapReduce usage statistics at Google are also available [60, 9].

The studies exhibit a clear tradeoff between efficiency and fault-tolerance. MapReduce increases the fault tolerance of long-time analysis by frequent checkpoints of completed tasks and data replication. However, the frequent I/Os required for fault-tolerance reduce efficiency. Parallel DBMS aims at efficiency rather than fault tolerance. DBMS actively exploits pipelining intermediate results between query operators. However, it causes a potential danger that a large amount of operations need be redone when a failure happens. With this fundamental difference, we categorize the pros and cons of the MapReduce framework below.

3.2 Advantages

MapReduce is simple and efficient for computing aggregate. Thus, it is often compared with "*filtering then group-by aggregation*" query processing in a DBMS. Here are major advantages of the MapReduce framework for data processing.

Simple and easy to use The MapReduce model is simple but expressive. With MapReduce, a programmer defines his job with only Map and Reduce functions, without having to specify physical distribution of his job across nodes.

Flexible MapReduce does not have any dependency on data model and schema. With MapReduce a programmer can deal with irregular or unstructured data more easily than they do with DBMS.

Independent of the storage MapReduce is basically independent from underlying storage layers. Thus, MapReduce can work with different storage layers such as BigTable[35] and others.

Fault tolerance MapReduce is highly fault-tolerant. For example, it is reported that MapReduce can continue to work in spite of an average of 1.2 failures per analysis job at Google[44, 38].

High scalability The best advantage of using MapReduce is high scalability. Yahoo! reported that their Hadoop gear could scale out more than 4,000 nodes in 2008[4].

3.3 Pitfalls

Despite many advantages, MapReduce lacks some of the features that have proven paramount to data analysis in DBMS. In this respect, MapReduce is often characterized as an *Extract-Transform-Load*(ETL) tool[51]. We itemize the pitfalls of the MapReduce framework below, compared with DBMS.

No high-level language MapReduce itself does not support any high-level language like SQL in DBMS and any query optimization technique. Users should code their operations in Map and Reduce functions.

No schema and no index MapReduce is schema-free and index-free. An MR job can work right after its input is loaded into its storage. However, this impromptu processing throws away the benefits of data modeling. MapReduce requires to parse each item at reading input and transform it into data objects for data processing, causing performance degradation [15, 11].

A Single fixed dataflow MapReduce provides the ease of use with a simple abstraction, but in a fixed dataflow. Therefore, many complex algorithms are hard to implement with Map and Reduce only in an MR job. In addition, some algorithms that require multiple inputs are not well supported since the dataflow of MapReduce is originally designed to read a single input and generate a single output.

Low efficiency With fault-tolerance and scalability as its primary goals, MapReduce operations are not always optimized for I/O efficiency. (Consider for example sort-merge based grouping, materialization of intermediate results and data triplication on the distributed file system.) In addition, Map and Reduce are blocking operations. A transition to the next stage cannot be made until all the tasks of the current stage are finished. Consequently, pipeline parallelism may not be exploited. Moreover, block-level restarts, a one-to-one shuffling strategy, and a simple runtime scheduling can also lower the efficiency per node. MapReduce does not have specific execution plans and does not optimize plans like DBMS does to minimize data transfer across nodes. Therefore, MapReduce often shows poorer performance than DBMS[15]. In addition, the MapReduce framework has a latency problem that comes from its inherent batch processing nature. All of inputs for an MR job should be prepared in advance for processing.

Very young MapReduce has been popularized by Google since 2004. Compared to over 40 years of DBMS, codes are not mature yet and third-party tools available are still relatively few.

4. VARIANTS AND IMPROVEMENTS

We present details of approaches to improving the pitfalls of the MapReduce framework in this section.

4.1 High-level Languages

Microsoft SCOPE[53], Apache Pig[22, 18], and Apache Hive[16, 17] all aim at supporting declarative query languages for the MapReduce framework. The declarative query languages allow query independence from program logics, reuse of the queries and automatic query optimization features like SQL does for DBMS. SCOPE works on top of the Cosmos system, a Microsoft's clone of MapReduce, and provides functionality similar to SQL views. It is similar to SQL but comes with C# expressions. Operators in SCOPE are the same as Map, Reduce and Merge supported in [37].

Pig is an open source project that is intended to support ad-hoc analysis of very large data, motivated by Sawzall[55], a scripting language for Google's MapReduce. Pig consists of a high-level dataflow language called *Pig Latin* and its execution framework. Pig Latin supports a nested data model and a set of pre-defined UDFs that can be customized [22]. The Pig execution framework first generates a logical query plan from a Pig Latin program. Then it compiles the logical plan down into a series of MR jobs. Some optimization techniques are adopted to the compilation, but not described in detail[18]. Pig is built on top of Hadoop framework, and its usage requires no modification to Hadoop.

Hive is an open-source project that aims at providing data warehouse solutions on top of Hadoop, supporting ad-hoc queries with an SQL-like query language called HiveQL. Hive compiles a HiveQL query into a directed acyclic graph(DAG) of MR jobs. The HiveQL includes its own type system and data definition language(DDL) to manage data integrity. It also contains a system catalog, containing schema information and statistics, much like DBMS engines. Hive currently provides only a simple, naive rule-based optimizer.

Similarly, DryadLINQ[71, 49] is developed to translate LINQ expressions of a program into a distributed execution plan for Dryad, Microsoft's parallel data processing tool [48].

4.2 Schema Support

As described in Section 3.3, MapReduce does not provide any schema support. Thus, the MapReduce framework parses each data record at reading input, causing performance degradation [15, 51, 11]. Meanwhile, Jiang *et al* report that only immutable decoding that transforms records into immutable data objects severely causes performance degradation, rather than record parsing [28].

While MapReduce itself does not provide any schema support, data formats such as Google's Protocol Buffers, XML, JSON, Apache's Thrift, or other formats can be used for checking data integrity [39]. One notable thing about the formats is that they are self-describing formats that support a nested and irregular data model, rather than the relational model. A drawback of the use of the formats is that data size may grow as data contains schema information in itself. Data compression is considered to address the data size problem [47].

4.3 Flexible Data Flow

There are many algorithms which are hard to directly map into Map and Reduce functions. For example, some algorithms require global state information during their processing. Loop is a typical example that requires the state information for execution and termination. However, MapReduce does not treat state information during execution. Thus, MapReduce reads the same data iteratively and materializes intermediate results in local disks in each iteration, requiring lots of I/Os and unnecessary computations. HaLoop[66], Twister[42], and Pregel[36] are examples of systems that support loop programs in MapReduce.

HaLoop and Twister avoid reading unnecessary data repeatedly by identifying and keeping invariant data during iterations. Similarly, Lin *et al* propose an in-mapper combining technique that preserves mapped outputs in a memory buffer across multiple map calls, and emits aggregated outputs at the last iteration [75]. In addition, Twister avoids instantiating workers repeatedly during iterations. Previously instantiated workers are reused for the next iteration with different inputs in Twister. HaLoop is similar to Twister, and it also allows to cache both each stage's input and output to save more I/Os during iterations. Vanilla Hadoop also supports task JVM reuse to avoid the overhead of starting a new JVM for each task [81]. Pregel mainly targets to process graph data. Graph data processing are usually known to require lots of iterations. Pregel implements a programming model motivated by the Bulk Synchronous Parallel(BSP) model. In this model, each node has each own input and transfers only some messages which are required for next iteration to other nodes.

MapReduce reads a single input. However, many important relational operators are binary operators that require two inputs. Map-Reduce-Merge addresses the support of the relational operators by simply adding a third *merge* stage after reduce stage [37]. The merge stage combines two reduced outputs from two different MR jobs into one.

Clustera, Dryad and Nephele/PACT allow more flexible dataflow than MapReduce does [31, 48, 30, 26]. Clustera is a cluster management system that is designed to handle a variety of job types including MR-style jobs [31]. Job scheduler in Clustera handles MapReduce, workflow and SQL-type jobs, and each job can be connected to form a DAG or a pipeline for complex computations.

Dryad is a notable example of distributed data-parallel tool that allows to design and execute a dataflow graph as users' wish [48]. The dataflow in Dryad has a form of DAG that consists of vertices and channels. Each vertex represents a program and a channel connects the vertices. For execution, a logical dataflow graph is mapped onto physical resources by a job scheduler at runtime. A vertex runs when all its inputs are ready and outputs its results to the neighbor vertices via channels as defined in the dataflow graph. The channels can be either of files, TCP pipes, or shared-memory. Job executions are controlled by a central job scheduler. Redundant executions are also allowed to handle apparently very slow vertices, like MapReduce. Dryad also allows to define how to shuffle intermediate data specifically.

Nephele/PACT is another parallel execution engine and its programming model[30, 26]. The PACT model extends MapReduce to support more flexible dataflows. In the model, each mapper can have a separate input and a user can specify its dataflow with more various stages including Map and Reduce. Nephele transforms a PACT program into a physical DAG then executes the DAG across nodes. Executions in Nephele are scheduled at runtime, like MapReduce.

4.4 Blocking Operators

Map and Reduce functions are blocking operations in that all tasks should be completed to move forward to the next stage or job. The reason is that MapReduce relies on external merge sort for grouping intermediate results. This property causes performance degradation and makes it difficult to support online processing.

Logothetis *et al* address this problem for the first time when they build MapReduce abstraction onto their distributed stream engine for ad-hoc data processing[29]. Their *incremental* MapReduce framework processes data like streaming engines. Each task runs continuously with a sliding window. Their system generates MR outputs by reading the items within the window. This stream-based MapReduce processes arriving increments of update tuples, avoiding recomputation of all the tuples from the beginning.

MapReduce Online is devised to support online aggregation and continuous queries in MapReduce[63]. It raises an issue that pull-based communication and checkpoints of mapped outputs limit pipelined processing. To promote pipelining between tasks, they modify MapReduce architecture by making Mappers push their data temporarily stored in local storage to Reducers periodically in the same MR job. Map-side pre-aggregation is also used to reduce communication volumes further.

Li *et al* and Jiang *et al* have found that the merge sort in MapReduce is I/O intensive and dominantly affects the performance of MapReduce [21, 28]. This leads to the use of hash tables for better performance and also incremental processing [21]. In the study, as soon as each map task outputs its intermediate results, the results are hashed and pushed to hash tables held by reducers. Then, reducers perform aggregation on the values in each bucket. Since each bucket in the hash table holds all values which correspond to a distinct key, no grouping is required. In addition, reducers can perform aggregation on the fly even when all mappers are not completed yet.

4.5 I/O Optimization

There are also approaches to reducing I/O cost in MapReduce by using index structures, column-oriented storage, or data compression.

Hadoop++ provides an index-structured file format to improve the I/O cost of Hadoop [40]. However, as it needs to build an index for each file partition at data loading stage, loading time is significantly increased. If the input data are processed just once, the additional cost given by building index may not be justified. HadoopDB also benefits from DB indexes by leveraging DBMS as a storage in each node [11].

There are many studies that describe how column-oriented techniques can be leveraged to improve MapReduce's performance dramatically [35, 62, 68, 12, 69]. Google's BigTable proposes the concept of column family that groups one or more columns as a basic working unit[35]. Google's Dremel is a nested column-oriented storage that is designed to complement MapReduce[62]. The read-only nested data in Dremel are modeled with Protocol Buffers [47]. The data in Dremel are split into multiple columns and records are assembled via finite state machines for record-oriented requests. Dremel is also known to support ad-hoc queries like Hive [16].

Record Columnar File(RCFile), developed by Facebook and adopted by Hive and Pig, is a column-oriented file format on HDFS [68]. Data placement in HDFS is determined by the master node at runtime. Thus, it is argued that if each column in a relation is independently stored in a separate file on HDFS, all related fields in the same record cannot guarantee to be stored in the same node. To get around this, a file format that represents all values of a relation column-wise in a single file is devised. A RCFile consists of a set of row groups, which are acquired by partitioning a relation horizontally. Then in each row group, values are enumerated in column-wise, similar to PAX storage scheme [3].

Llama shows how column-wise data placement helps join processing [69]. A column-oriented file in Llama stores a particular column data with optional index information. It also witnesses that *late materialization* which delays record reconstruction until the column is necessary during query processing is no better than early materialization in many cases.

Floratou *et al* propose a binary column-oriented storage that boosts the performance of Hadoop by an order of magnitude[12]. Their storage format stores each column in a separate file but co-locate associated column files in the same node by changing data placement policy of Hadoop. They also suggest that late materialization with skiplist shows better performance than early materialization, contrary to the result of RCFile. Both Floratou's work and RCFile also use a column-wise data compression in each row group, and adopt a lazy decompression technique to avoid unnecessary decompression during query execution. Hadoop also supports the compression of mapped outputs to save I/Os during the checkpoints[81].

4.6 Scheduling

MapReduce uses a block-level runtime scheduling with a speculative execution. A separate Map task is created to process a single data block. A node which finishes its task early gets more tasks. Tasks on a straggler node are redundantly executed on other idle nodes.

Hadoop scheduler implements the speculative task scheduling with a simple heuristic method which compares the progress of each task to the average progress. Tasks with the lowest progress compared to the average are selected for re-execution. However, this heuristic method is not well suited in a heterogeneous environment where each node has different computing power. In this environment, even a node whose task progresses further than others may be the last if the node's computing power is inferior to others. Longest Approximate Time to End(LATE) scheduling is devised to improve the response time of Hadoop in heterogeneous environments [52]. This scheduling scheme estimates the task progress with the *progress rate*, rather than simple progress score.

Parallax is devised to estimate job progress more precisely for a series of jobs compiled from a Pig program [45]. it pre-runs with sampled data for estimating the processing speeds of each stage. ParaTimer is an extended version of Parallax for DAG-style jobs written in Pig [46]. ParaTimer identifies a critical path that takes longer than others in a parallel query plan. It makes the indicator ignore other shorter paths when estimating progress since the longest path would contribute the overall execution time. Besides, it is reported that the more data blocks to be scheduled, the more cost the scheduler will pay [65]. Thus, a rule of thumb in industry – making the size of data block bigger makes Hadoop work faster – is credible.

We now look into multi-user environment whereby users simultaneously execute their jobs in a cluster. Hadoop implements two scheduling schemes: *fair scheduling* and *capacity scheduling*. The default fair scheduling works with a single queue of jobs. It assigns physical resources to jobs such that all jobs get an equal share of resources over time on average. In this scheduling scheme, if there is only a single MR job running in a cluster, The job solely uses entire resources in the cluster. Capacity sharing supports designing more sophisticated scheduling. It provides multiple queues each of which is guaranteed to possess a certain capacity of the cluster.

MRShare is a remarkable work for sharing multiple query executions in MapReduce [64]. MRShare, inspired by multi query optimization techniques in database, finds an optimal way of grouping a set of queries using dynamic programming. They suggest three sharing opportunities across multiple MR jobs in MapReduce, like found in Pig [18]: scan sharing, mapped outputs sharing, and Map function sharing. They also introduce a cost model for MR jobs and validate this with experiments. Their experiments show that intermediate result sharing improves the execution time significantly. In addition, they have found that sharing all scans yield poorer performance as the size of intermediate results increases, because of the complexity of the merge-sort operation in MapReduce. Suppose that $|D|$ is the size of input data that n MR jobs share. When sharing all scans, the cost of scanning inputs is reduced by $|D|$, compared to $n \cdot |D|$ for no sharing scans. However, as a result, the complexity of sorting the *combined* mapped output of all jobs will be $O(n \cdot |D| log(n \cdot |D|))$ since each job can generate its own mapped output with size $O(|D|)$. This cost can be bigger than the total cost of sorting n different jobs, $O(n \cdot |D| log|D|)$ in some cases.

4.7 Joins

Join is a popular operator that is not so well dealt with by Map and Reduce functions. Since MapReduce is designed for processing a single input, the support of joins that require more than two inputs with MapReduce has been an open issue. We roughly classify join methods within MapReduce into two groups: *Map-side join* and *Reduce-side join*. We also borrow some of terms from Blanas *et al*'s study, which compares many join techniques for analysis of clickstream logs at Facebook [57], for explaining join techniques.

Map-side Join

Map-Merge join is a common map-side join that works similarly to sort-merge join in DBMS. Map-Merge join performs in two steps. First, two input relations are partitioned and sorted on the join keys. Second, map-

pers read the inputs and merge them [81]. Broadcast join is another map-side join method, which is applicable when the size of one relation is small [57, 7]. The smaller relation is broadcast to each mapper and kept in memory. This avoids I/Os for moving and sorting on both relations. Broadcast join uses in-memory hash tables to store the smaller relation and to find matches via table lookup with values from the other input relation.

Reduce-side Join

Repartition join is the most general reduce-side join [81, 57]. Each mapper tags each row of two relations to identify which relation the row come from. After that, rows of which keys have the same key value are copied to the same reducer during shuffling. Finally, each reducer joins the rows on the key-equality basis. This way is akin to hash-join in DBMS. An improved version of the repartition join is also proposed to fix the buffering problem that all records for a given key need to be buffered in memory during the joining process[57].

Lin *et al* propose a scheme called "schimmy" to save I/O cost during reduce-side join[75]. The basic concept of the scheme is to separate messages from graph structure data, and shuffle only the message to avoid shuffling data, similar to Pregel [36]. In this scheme, mappers emit only messages. Reducers read graph structure data directly from HDFS and do reduce-side merge join between the data and the messages.

MapReduce Variants

Map-Reduce-Merge is the first that attempts to address join problem in the MapReduce framework [37]. To support binary operations including join, Map-Reduce-Merge extends MapReduce model by adding Merge stage after Reduce stage.

Map-Join-Reduce is another variant of MapReduce framework for one-phase joining [27]. The authors propose a filtering-join-aggregation model that adds *Join* stage before Reduce stage to perform joining within a single MR job. Each mapper reads tuples from a separate relation which take part in a join process. After that, the mapped outputs are shuffled and moved to joiners for actual joining, then the `Reduce()` function is applied. Joiners and reducers are actually run inside the same reduce task. An alternative that runs Map-Join-Reduce with two consecutive MR jobs is also proposed to avoid modifying MapReduce framework. For multi-way join, join chains are represented as a left-deep tree. Then previous joiners transfer joined tuples to the next joiner that is the parent operation of the previous joiners in the tree. For this, Map-Join-Reduce adopts one-to-many shuffling scheme that shuffles and assigns each mapped outputs to multiple joiners at a time.

Other Join Types

Joins may have more than two relations. If relations are simply hash-partitioned and fed to reducers, each reducer takes a different portion of the relations. However, the same relation must be copied to all reducers to avoid generating incomplete join results in the cases. For example, given a multi-way join that reads 4 relations and with 4 reducers, we can split only 2 relations making 4 partitions in total. The other relations need to be copied to all reducers. If more relations are involved into less reducers, we spend more communication costs. Afrati *et al* focus on how to minimize the sum of the communication cost of data that are transferred to Reducers for multi-way join [2]. They suggest a method based on Lagrangean multipliers to properly select which columns and how many of the columns should be partitioned for minimizing the sum of the communication costs. Lin *et al* propose the concurrent join that performs a multi-way join in parallel with MapReduce [69].

In addition to binary equal-join, other join types have been widely studied. Okcan *et al* propose how to efficiently perform θ-join with a single MR job only [14]. Their algorithm uses a Reducer-centered cost model that calculates the total cost of Cartesian product of mapped output. With the cost model, they assigns mapped output to reducers that minimizes job completion time. The support of Semi-join, *e.g.* $R \ltimes S$, is proposed in [57]. Vernica *et al* propose how to efficiently parallelize set-similarity joins with Mapreduce [56]. They utilize prefix filtering to filter out non-candidates before actual comparison. It requires to extract common prefixes sorted in a global order of frequency from tuples, each of which consists of a set of items.

4.8 Performance Tuning

Most of MapReduce programs are written for data analysis and they usually take much time to be finished. Thus, it is straightforward to provide the feature of automatic optimization for MapReduce programs. Babu *et al* suggest an automatic tuning approach to finding optimal system parameters for given input data [5]. It is based on speculative pre-runs with sampled data. Jahani *et al* suggest a static analysis approach called MANIMAL for automatic optimization of a single MapReduce job [34]. In their approach, an analyzer examines program codes before execution without any runtime information. Based on the rules found during the analysis, it creates a pre-computed B^+-tree index and slices input data column-wise for later use. In addition, some semantic-aware compression techniques are used for reducing I/O. Its limitation is that the optimization considers only selection and projection which are primarily implemented in Map function.

4.9 Energy Issues

Energy issue is important especially in this data-center computing era. Since the energy cost of data centers hits 23% of the total amortized monthly operating ex-

penses, it is prudent to devise an energy-efficient way to control nodes in a data center when the nodes are idle[74]. In this respect, two extreme strategies for energy management in MapReduce clusters are examined [74, 43]. Covering-Set approach designates in advance some nodes that should keep at least a replica of each data block, and the other nodes are powered down during low-utilization periods [43]. Since the dedicated nodes always have more than one replica of data, all data across nodes are accessible in any cases except for multiple node failures. On the contrary, All-In strategy saves energy in an all-or-nothing fashion [74]. In the strategy, all MR jobs are queued until it reaches a threshold predetermined. If it exceeds, all nodes in the cluster run to finish all MR jobs and then all the nodes are powered down until new jobs are queued enough. Lang *et al* concluded that All-In strategy is superior to Covering-Set in that it does not require changing data placement policy and response time degradation. However, All-In strategy may not support an instant execution because of its batch nature. Similarly, Chen *et al* discuss the computation versus I/O tradeoffs when using data compressions in a MapReduce cluster in terms of energy efficiency [67].

4.10 Hybrid Systems

HadoopDB is a hybrid system that connects multiple single-node DBMS with MapReduce for combining MapReduce-style scalability and the performance of DBMS [11]. HadoopDB utilizes MapReduce as a distributing system which controls multiple nodes which run single-node DBMS engines. Queries are written in SQL, and distributed via MapReduce across nodes. Data processing is boosted by the features of single-node DBMS engines as workload is assigned to the DBMS engines as much as possible.

SQL/MapReduce is another hybrid framework that enables to execute UDF functions in SQL queries across multiple nodes in MapReduce-style [33]. UDFs extend a DBMS with customizing DB functionality. SQL/MapReduce presents an approach to implementing UDF that can be executed across multiple nodes in parallel by virtue of MapReduce. Greenplum also provides the ability to write MR functions in their parallel DBMS. Teradata makes its effort to combine Hadoop with their parallel DBMS [70]. The authors describe their three efforts toward tight and efficient integration of Hadoop and Teradata EDW: parallel loading of Hadoop data to EDW, retrieving EDW data from MR programs, and accessing Hadoop data from SQL via UDFs.

5. APPLICATIONS AND ADAPTATIONS

5.1 Applications

Mahout is an Apache project that aims at building scalable machine learning libraries which are executed in parallel by virtue of Hadoop [1]. RHIPE and Ricardo project are tools that integrate R statistical tool and Hadoop to support parallel data analysis [73, 58]. Cheetah is a data warehousing tool built on MapReduce with virtual view on top of the star or snowflake schemas and with some optimization techniques like data compression and columnar store [7]. Osprey is a shared-nothing database system that supports MapReduce-style fault tolerance [25]. Osprey does not directly use MapReduce or GFS. However, the fact table in star schema is partitioned and replicated like GFS, and tasks are scheduled by a central runtime scheduler like MapReduce. A difference is that Osprey does not checkpoint intermediate outputs. Instead, it uses a technique called chained declustering which limits data unavailability when node failures happen.

The use of MapReduce for data intensive scientific analysis and bioinformatics are well studied in [41, 80]. CloudBLAST parallelizes NCBI BLAST2 algorithm using Hadoop [13]. They break their input sequences down into multiple blocks and run an instance of the vanilla version of NCBI BLAST2 for each block, using the Hadoop Streaming utility [81]. CloudBurst is a parallel read-mapping tool that maps NGS read sequencing data to a reference genome for genotyping in parallel [78].

5.2 Adaptations to Intra-node Parallelism

Some studies use the MapReduce model for simplifying complex multi-thread programming on many-core systems such as multi-core[24, 65], GPU[20, 19], and Cell processors[8]. In the studies, mapped outputs are transferred to reducers via shared-memory rather than disks. In addition, a task execution is performed by a single core rather than a node. In this intra-node parallelism, fault-tolerance can be ignored since all cores are located in a single system. A combination of intra-node and inter-node parallelism by the MapReduce model is also suggested [54].

6. DISCUSSION AND CHALLENGES

MapReduce is becoming ubiquitous, even though its efficiency and performance are controversial. There is nothing new about principles used in MapReduce [10, 51]. However, MapReduce shows that many problems can be solved in the model at scale unprecedented before. Due to frequent checkpoints and runtime scheduling with speculative execution, MapReduce reveals low efficiency. However, such methods would be necessary to achieve high scalability and fault tolerance in massive data processing. Thus, how to increase efficiency guaranteeing the same level of scalability and fault tolerance is a major challenge. The efficiency problem is expected to be overcome in two ways: improving MapReduce it-

self and leveraging new hardware. How to utilize the features of modern hardware has not been answered in many areas. However, modern computing devices such as chip-level multiprocessors and Solid State Disk(SSD) can help reduce computations and I/Os in MapReduce significantly. The use of SSD in Hadoop with simple installation is briefly examined, but not in detail [21]. Self-tuning and job scheduling in multi-user environments are another issues that have not been well address yet. The size of MR clusters is continuously increasing. A 4,000-node cluster is not surprising any more. How to efficiently manage resources in the clusters of that size in multi-user environment is also challenging. Yahoo's M45 cluster reportedly shows only 5∼10% resource utilization [60]. Energy efficiency in the clusters and achieving high utilizations of MR clusters are also important problems that we consider for achieving better TCO and return on investments in practice.

7. CONCLUSION

We discussed pros and cons of MapReduce and classified its improvements. MapReduce is simple but provides good scalability and fault-tolerance for massive data processing. However, MapReduce is unlikely to substitute DBMS even for data warehousing. Instead, we expect that MapReduce complements DBMS with scalable and flexible parallel processing for various data analysis such as scientific data processing. Nonetheless, efficiency, especially I/O costs of MapReduce still need to be addressed for successful implications.

Acknowledgement

This work is supported by the National Research Fund (NRF) grant funded by Korea government(MEST)(No. 2011-0016282).

8. REFERENCES

[1] Mahout: Scalable machine-learning and data-mining library. *http://mapout.apache.org*, 2010.
[2] F.N. Afrati and J.D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th EDBT*, pages 99–110, 2010.
[3] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving relations for cache performance. *The VLDB Journal*, pages 169–180, 2001.
[4] A. Anand. Scaling Hadoop to 4000 nodes at Yahoo! *http://goo.gl/8dRMq*, 2008.
[5] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142, 2010.
[6] Nokia Research Center. Disco: Massive data- minimal code. *http://discoproject.org*, 2010.
[7] S. Chen. Cheetah: a high performance, custom data warehouse on top of MapReduce. *Proceedings of the VLDB*, 3(1-2):1459–1468, 2010.
[8] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, 2009.
[9] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.
[10] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. *The Database Column*, 1, 2008.
[11] A. Abouzeid *et al*. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
[12] A. Floratou *et al*. Column-Oriented Storage Techniques for MapReduce. *Proceedings of the VLDB*, 4(7), 2011.
[13] A. Matsunaga *et al*. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *Fourth IEEE International Conference on eScience*, pages 222–229, 2008.
[14] A. Okcan *et al*. Processing Theta-Joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD*, 2011.
[15] A. Pavlo *et al*. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD*, pages 165–178, 2009.
[16] A. Thusoo *et al*. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.
[17] A. Thusoo *et al*. Hive-a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th IEEE ICDE*, pages 996–1005, 2010.
[18] A.F. Gates *et al*. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.
[19] B. Catanzaro *et al*. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.
[20] B. He *et al*. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th PACT*, pages 260–269, 2008.
[21] B. Li *et al*. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proceedings of the 2011 ACM SIGMOD*, 2011.
[22] C. Olston *et al*. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD*, pages 1099–1110, 2008.
[23] C. Ordonez *et al*. Relational versus Non-Relational Database Systems for Data Warehousing. In *Proceedings of the ACM DOLAP*, pages 67–68, 2010.
[24] C. Ranger *et al*. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE HPCA*, pages 13–24, 2007.
[25] C. Yang *et al*. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proceedings of the 26th IEEE ICDE*, 2010.
[26] D. Battré *et al*. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, 2010.
[27] D. Jiang *et al*. Map-join-reduce: Towards scalable and efficient data analysis on large clusters. *IEEE Transactions on Knowledge and Data Engineering*, 2010.
[28] D. Jiang *et al*. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.
[29] D. Logothetis *et al*. Ad-hoc data processing in the cloud. *Proceedings of the VLDB Endowment*, 1(2):1472–1475, 2008.
[30] D. Warneke *et al*. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd MTAGS*, pages 1–10, 2009.
[31] D.J. DeWitt *et al*. Clustera: an integrated computation and data management system. *Proceedings of the VLDB Endowment*, 1(1):28–41, 2008.
[32] E. Anderson *et al*. Efficiency matters! *ACM SIGOPS Operating Systems Review*, 44(1):40–45, 2010.
[33] E. Friedman *et al*. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB*

Endowment, 2(2):1402–1413, 2009.
[34] E. Jahani *et al*. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB*, 4(6):385–396, 2011.
[35] F. Chang *et al*. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
[36] G. Malewicz *et al*. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD*, pages 135–146, 2010.
[37] H. Yang *et al*. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD*, pages 1029–1040, 2007.
[38] J. Dean *et al*. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[39] J. Dean *et al*. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.
[40] J. Dittrich *et al*. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.
[41] J. Ekanayake *et al*. Mapreduce for data intensive scientific analyses. In *the 4th IEEE International Conference on eScience*, pages 277–284, 2008.
[42] J. Ekanayake *et al*. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM HPDC*, pages 810–818, 2010.
[43] J. Leverich *et al*. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.
[44] Jeffrey Dean *et al*. Mapreduce: Simplified data processing on large clusters. In *In Proceedings of the 6th USENIX OSDI*, pages 137–150, 2004.
[45] K. Morton *et al*. Estimating the Progress of MapReduce Pipelines. In *Proceedings of the the 26th IEEE ICDE*, pages 681–684, 2010.
[46] K. Morton *et al*. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD*, pages 507–518, 2010.
[47] Kenton *et al*. Protocol Buffer- Google's data interchange format. *http://code.google.com/p/protobuf/*.
[48] M. Isard *et al*. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.
[49] M. Isard *et al*. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the ACM SIGMOD*, pages 987–994, 2009.
[50] M. Stonebraker *et al*. One size fits all? Part 2: Benchmarking results. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.
[51] M. Stonebraker *et al*. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.
[52] M. Zaharia *et al*. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX OSDI*, pages 29–42, 2008.
[53] R. Chaiken *et al*. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.
[54] R. Farivar *et al*. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.
[55] R. Pike *et al*. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
[56] R. Vernica *et al*. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD*, pages 495–506. ACM, 2010.
[57] S. Blanas *et al*. A comparison of join algorithms for log processing in MaPreduce. In *Proceedings of the 2010 ACM SIGMOD*, pages 975–986, 2010.
[58] S. Das *et al*. Ricardo: integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD*, pages 987–998, 2010.
[59] S. Ghemawat *et al*. The google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.
[60] S. Kavulya *et al*. An analysis of traces from a production mapreduce cluster. In *10th IEEE/ACM CCGrid*, pages 94–103, 2010.
[61] S. Loebman *et al*. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.
[62] S. Melnik *et al*. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.
[63] T. Condie *et al*. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, 2010.
[64] T. Nykiel *et al*. MRshare: Sharing across multiple queries in mapreduce. *Proceedings of the VLDB*, 3(1-2):494–505, 2010.
[65] W. Jiang *et al*. A Map-Reduce System with an Alternate API for Multi-core Environments. In *Proceedings of the 10th IEEE/ACM CCGrid*, pages 84–93, 2010.
[66] Y. Bu *et al*. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.
[67] Y. Chen *et al*. To compress or not to compress - compute vs. IO tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pages 23–28. ACM, 2010.
[68] Y. He *et al*. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 2011 IEEE ICDE*, 2011.
[69] Y. Lin *et al*. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *Proceedings of the 2011 ACM SIGMOD*, 2011.
[70] Y. Xu *et al*. Integrating Hadoop and parallel DBMS. In *Proceedings of the ACM SIGMOD*, pages 969–974, 2010.
[71] Y. Yu *et al*. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX OSDI*, pages 1–14, 2008.
[72] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *ACM SIGMOD Record*, 38(1):43–48, 2009.
[73] Saptarshi Guha. RHIPE- R and Hadoop Integrated Processing Environment. *http://www.stat.purdue.edu/ sguha/rhipe/*, 2010.
[74] W. Lang and J.M. Patel. Energy management for MapReduce clusters. *Proceedings of the VLDB*, 3(1-2):129–139, 2010.
[75] J. Lin and C. Dyer. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.
[76] O. O'Malley and A.C. Murthy. Winning a 60 second dash with a yellow elephant. *Proceedings of sort benchmark*, 2009.
[77] D.A. Patterson. Technical perspective: the data center is the computer. *Communications of the ACM*, 51(1):105–105, 2008.
[78] M.C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363, 2009.
[79] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *Proceedings of the 21st IEEE ICDE*, pages 2–11. IEEE, 2005.
[80] R. Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.
[81] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.