The Performance of MapReduce: An In-depth Study

Dawei Jiang Beng Chin Ooi Lei Shi Sai Wu

School of Computing
National University of Singapore
{jiangdw, ooibc, shilei, wusai}@comp.nus.edu.sg

ABSTRACT

MapReduce-based systems have been widely used for large-scale data analysis. Although these systems achieve storage-system independence, high scalability, and fine-grained fault tolerance, their performance are not satisfactory. According to a recent study [14], MapReduce-based systems are significantly slower than Parallel Database systems in performing a variety of analytic tasks. Some attribute the performance gap between MapReduce-based and Parallel Database systems to architectural design. This speculation yields an interesting question: Must a system sacrifice performance to achieve flexibility and scalability?

Inspired by the question above, we conducted an in-depth performance study of MapReduce in its open source implementation, Hadoop. We identify four factors that have significant performance effect on MapReduce, and investigate alternative strategies for each factor. Finally, we evaluate the performance of MapReduce on a representative yet tractable combinations of the four factors using the same benchmark in [14]. The results show that with proper implementation, the performance of MapReduce can be improved by a factor of 2.5 to 3.5 and approaches to Parallel Databases. Our results show that it is possible to build a MapReduce-based system that is not only flexible and scalable, but is also efficient.

1. INTRODUCTION

MapReduce-based systems are increasingly being used for large-scale data analysis. There are several reasons for this. First, the interface of MapReduce is simple yet expressive. Although MapReduce only involves two functions `map()` and `reduce()`, a number of data analytical tasks including traditional SQL query, data mining, machine learning, and graph processing can be expressed with a set of MapReduce jobs. Second, MapReduce is flexible. It is designed to be independent of storage systems and is able to analyze various kinds of data, structured and unstructured. Finally, MapReduce is scalable. Installation of MapReduce on a 4,000 nodes shared-nothing cluster has been reported [2]. MapReduce also provides fine-grain fault tolerance whereby only tasks on failed nodes need to be restarted.

Traditionally, the large-scale data analysis market is dominated by Parallel Database systems. The popularity of MapReduce gives rise to the question of whether there are fundamental differences between MapReduce-based and Parallel Database systems. Along this direction, [14] reported a comparative evaluation of the two systems in many dimensions, including schema support, data access methods, fault tolerance and so on. The authors also introduced a benchmark to evaluate the performance of both systems. The results showed that the observed performance of a Parallel Database system is much better than that of a MapReduce-based system. The authors of [14] speculated about possible architectural causes for the performance gap between the two systems. For instance, MapReduce-based systems need to repetitively parse records since it is designed to be independent of the storage system. Thus, parsing introduces performance overhead. These speculations prompted an interesting question: In achieving scalability and flexibility, must MapReduce-based systems trade off performance? In other words, is it not possible to have a flexible, scalable and efficient MapReduce-based systems?

Inspired by the above question, we conducted an in-depth performance study of MapReduce on an open source implementation, Hadoop. Based on our study, we identify several performance bottlenecks, some of these have been previously observed by other researchers. We then adopt/adapt well-known engineering and database techniques to manage these bottlenecks. Our performance evaluation using the benchmark developed in [14]. shows the effectiveness of these techniques. Our findings are as follows:

- Although MapReduce is independent of the underline storage system, it still requires the storage system to provide efficient I/O modes for scanning data. We identify two kinds of I/O modes: direct I/O and streaming I/O. Benchmark on HDFS shows that direct I/O outperforms streaming I/O by 10%∼15%.

- We find that MapReduce can utilize an index in a straightforward way. We show MapReduce can benefit from three kinds of indices: range-indexes, block-level indexes, and database indexed tables. Our benchmarking shows that the range-index improves the performance of MapReduce by a factor of 2 in the selection task and a factor of 10 in the join task when selectivity is high.

- Parsing records are not the source of poor performance. Our conclusion is different from that reported in [14], [15], and [6]. In our study, we identify two kinds of record decoders: mutable decoders and immutable decoders. We find that only immutable decoders introduce performance bottleneck. To handle database-like workloads, MapReduce users should strictly use mutable decoders. We show that a mutable decoder is faster than an immutable decoder by a factor of 10, and improves the performance of selection by a factor of 2. Using a mutable decoder, even parsing text record is efficient.

- We find that map-side sorting exerts negative performance effect on large aggregation tasks which require nontrivial key comparisons and produce millions of groups. We show that fingerprinting-based sort can significantly improve the performance of MapReduce on such aggregation tasks. Our benchmarking shows that fingerprinting-based sort outperforms direct sort by a factor of 4 to 5, and improves overall performance of the job by 20%∼25%.

- We also find scheduling strategy affects the performance of MapReduce. The current scheduling strategy in Hadoop is sensitive to the processing speed of slave nodes, and slows down the execution time of the entire job by 25%∼35%.

In summary, with proper implementation, the performance of MapReduce-based systems can be improved by a factor of 2.5 to 3.5. This means that, contrary to popular belief, MapReduce-based systems are not inferior to Parallel Database systems in terms of performance; instead, they can offer a competitive edge as they are flexible, scalable and efficient. The rest of the paper is organized as follows: Section 2 presents factors that affect the performance of MapReduce. Section 3 presents the combination factors that we will evaluate. Section 4 presents implementation details. Section 5 presents our benchmark results. Section 6 discusses related work, and we conclude in Section 7.

2. FACTORS AFFECTING PERFORMANCE OF MAPREDUCE

This Section identifies factors that may have a significant effect on the observed performance of MapReduce. The execution of a single MapReduce job consists of seven steps: 1) invoke map functions to read data from a storage system into memory, 2) parse/decode data into records, 3) process the records, 4) sort the intermediate data emitted by map functions according to keys, 5) shuffle intermediate data from mappers to reducers, 6) merge data into groups by intermediate keys, 7) invoke reduce functions and write resulting records back to a storage system. In Hadoop, the execution of a MapReduce job is monitored and managed by a JobTracker node, i.e., a master node which manages a MapReduce cluster. Map functions and Reduce functions are executed on a number of slave nodes (called TaskTracker nodes).

We find that different implementation strategies in Step 1), Step 2), and Step 4) are factors that have major performance impact, namely: 1) the I/O mode that the map function reads data off the storage system, 2) the decoding method that transforms raw data into records, and 3) the key comparison strategy used during sorting for computing aggregations.

2.1 I/O mode

MapReduce is designed to be independent of the underlying storage system. A Map function takes input from a *reader* instead of the storage system. The reader repeatedly reads data from the storage system into a memory buffer (64KB or 128KB in typical) for the map function to process until all the data in the data chunk are exhausted. A reader is conceptually similar to a wrapper in terms of database terminology. To analyze data stored in a new storage system, the user implements a new reader that can operate on that storage system. Currently, Hadoop provides readers for MapReduce jobs to read data from two kinds of storage systems: HDFS and relational databases.

We now examine the overhead of reading data through readers. Readers have two ways to read data from storage systems: 1) direct I/O, namely read data from the disk directly, 2) streaming I/O, namely streaming data from the storage system by an inter-process communication scheme, such as TCP/IP or JDBC. Streaming I/O is more general in that it can be used for reading data from both local node and remote node. We call a map function a data-local map if the map function reads data from the local node.

Streaming I/O is the only choice if a map function is not a data-local map. However, for data-local maps, we would expect direct I/O is more efficient. In Hadoop, all readers use streaming I/O to extract data from storage systems, i.e. HDFS or databases, no matter whether the data is stored in the local node or a remote node. In this paper, we implement a direct I/O mode on HDFS and evaluate the performance of different I/O modes.

2.2 Indexing

The factor that we next consider is how MapReduce utilizes indexing to speedup data process. We identify three cases.

First, if the input of a MapReduce job is a set of files stored in a distributed file system, i.e. HDFS, and each file is already sorted on keys, MapReduce can use a range-index to prune unnecessary data chunks and direct map functions to only scan data chunks that store records of interest. In this paper, we evaluate a simple range indexing scheme. This indexing creates an index entry for each fixed sized data chunk [1], sorts those index entries and store them in an index file. Each index entry is of the form (k_b, k_e, D_s) where k_b and k_e are the start key and end key of the indexed data chunk, D_s is the offset of the data chunk in the HDFS file. Index entries are sorted based on k_b.

Second, if the input HDFS files of MapReduce are not sorted but each data chunk in the files are indexed by keys, MapReduce can also utilize this chunk-index by scheduling a map task on each data chunk and configure the reader to apply the index for searching desired records. This strategy can be used when HDFS is used for backup or archiving a set of database files where each database file is an indexed table.

Finally, if the input of MapReduce are tables stored in database servers, MapReduce can transparently utilize database

[1]The size is defined by the user

indexed tables by pushing SQL queries to database and process the query results. This strategy is already used in [6].

We initially plan to evaluate all three possibilities. For chunk-indexing, we implement a scheme which stores Berkeley DB's database files in HDFS and enable MapReduce to analyze records stored in those databases by using of index. However, Berkeley DB application automatically performs database recovery each time the application is launched. This recovery process takes a long time and thus slows down all benchmarks. Therefore, we remove the results of Berkeley DB.

2.3 Parsing

When a reader loads data into the memory buffer, before the data can be processed by the map function, the raw data must be converted into a set of records, i.e., key-value pairs, and fields in each value part must also be decoded to appropriate types. Previous studies show that parsing has considerable performance overhead [14][15][6]. This Section discusses this problem. Without loss of generality, we focus our study on decoding structured records where value part in each record consists of many attributes. We both consider decoding text record and binary record.

In this paper, we identify two kinds of decoding scheme: immutable decoding and mutable decoding. An immutable decoder decodes raw data as immutable records. Immutable records are read-only records whose fields in the value part can only be set once and cannot be changed after the record is created. Using immutable decoder, a new immutable record is created each time the decoder is called by the map function for parsing the next input record. As a result, parsing four million records produces four million immutable records.

One can also use mutable decoding scheme. Using this approach, a mutable record whose fields can be reset is created first and is reused for decoding all records. Each time a mutable decoder is called by the map function, it decodes the raw data and fills the fields of the mutable record with next record. Thus, no matter how many records will be decoded, only one mutable record is created.

We found that the poor performance of record parsing observed in [14], [15], and [6] is mainly due to fact that all these studies adopt an immutable decoder rather than a mutable decoder. Immutable decoder is significantly slower than mutable decoder as its produces a huge number of immutable records in the decoding process. Creation of those immutable records consumes a huge amount of memory and incurs a huge overhead on CPU. In this paper, we will evaluate the performance of both decoders. In general, MapReduce users should strictly use mutable decoders whenever possible.

2.4 Sorting

The fourth factor we consider is sorting. MapReduce employs a sort-merge strategy for performing all kinds of data analysis tasks. This sort-merge strategy affects the performance of aggregation tasks, especially when the aggregation will produce a large number of groups and the cost of key comparison is not trivial. Consider we are required to calculate the total revenue of each sourceIP in the UserVisits table, namely the aggregation benchmark used in [14]. To perform this task, the MapReduce framework needs to sort the intermediate records emitted by the map function according to sourceIP. Each sourceIP is a variable-length string with 16 bytes at most. Thus the comparison of two sourceIPs need 16 byte-to-byte comparisons in the worst case. Performing such string comparisons millions of times introduces performance penalty.

This paper evaluates an alternative strategy. When an intermediate record is emitted, we store a fingerprint, a 32bits integer, of the key along with that record. When MapReduce sorts the intermediate records, we first compare their fingerprints of keys. If two keys have the same fingerprint, we compare the original keys. This fingerprints comparison strategy reduces the cost of comparing two keys which are fingerprinted to different values. In such cases, only one integer comparison, typically finished in one CPU instruction, is needed. Merging intermediate records at reduce side also needs to be adjusted accordingly. This is achieved by first grouping records with fingerprints and for records with the same fingerprint, we group them with original keys.

3. PRUNING SEARCH SPACE

Combinations of the four factors discussed so far result in a huge search space. In addition, as all benchmarks are conducted on Amazon EC2, the budget on EC2 usage also imposes a constraint on us. Therefore, we narrow down the search space into a representative but tractable set. The pruning approach that we used is as follows.

We evaluate the performance of MapReduce on two kinds of storage systems: a distributed file system, i.e., HDFS and a database system, i.e., PostgreSQL. These two storage systems, we believe, represent the typical usage of MapReduce based data analysis. For HDFS, we report the performance of all benchmark data analysis tasks. For PostgreSQL, we only perform a subset of benchmark analysis tasks. This is because loading data to PostgreSQL and building cluster index consumes a lot of time. Thus we only run analysis tasks on those datasets whose loading time is within 3 hours.

We first conduct a micro benchmark on HDFS to study the performance of streaming I/O and direct I/O when map reads local data. The fastest I/O mode will be used in subsequent benchmark. For PostgreSQL, streaming I/O with JDBC is always used. There is a trend in database industries called "In-Database MapReduce" which is a technology of running map and reduce functions inside database query engine. This "In-Database MapReduce" strategy could be characterized as direct I/O support for MapReduce from databases. However, such a system is unavailable to us.

Second, we evaluate the performance of record parsing with a special focus on the difference of performance between immutable decoders and mutable decoders. Text record parsing and binary record parsing are both considered. For binary record parsing, we evaluate three schemes: 1) Hadoop's `Writable`, 2) Google's protocol buffer [1], and 3) Berkeley DB's tuple binding API (BDB API) [4]. We implement a mutable decoder and an immutable decoder for both kinds of records, text and binary, with exception for BDB API as BDB only supports mutable decoding. The decoder with the best performance is then chosen for the benchmark. We have designed a `KeyValueSequenceFile` storage format for storing binary records in HDFS files. We do not use Hadoop's `SequenceFile` since it only supports `Writable` decoding. We initially plan to evaluate the effect of compression on the performance of MapReduce. However, compression is only available in `SequenceFile`, port-

ing compress/uncompress codes from `SequenceFile` to our `KeyValueSequenceFile` takes too many time. After a few trials and errors, we finally discard this option and only report results without compression. Enabling compression should not change the conclusions of this paper dramatically, as in previous study [15], compression only contributes a little bit performance improvement for MapReduce jobs.

Finally, we evaluate the performance of MapReduce on a real benchmark with HDFS with fastest I/O mode, decoders with best performance, along with different sort schemes (direct key comparison and fingerprint comparison). The benchmark we used is identical to the one in [14] and [15]. Descriptions of datasets and queries are described in Appendix. We use three datasets (Grep, Rankings, and UserVisits) and four queries (grep, selection, aggregation, and join) in this study. This benchmark consists of four datasets and five queries. The dataset and query that we do not use is designed for benchmarking user defined functions which is not the focus of this paper. We choose this benchmark since it is widely used in recent studies for benchmarking the performance of MapReduce and Parallel Databases.

4. IMPLEMENTATION DETAILS

We use Hadoop v0.19.2 as the code base. All MapReduce programs are written in Java, the default programming language for MapReduce in Hadoop. All the source codes will be available in our project's website.

We implement direct I/O for HDFS to enable map functions to read data directly from the disk on local data node instead of streaming data through TCP. This is achieved by modifications of data node implementation. HDFS stores a large data file as a set of data chunks among available data nodes. Each data chunk is stored as a local file in a certain data node. When a data node receives a data request, it first checks whether the data request is from a local map's reader [2]. If that is the case, the data node passes the local data chunk file's name to the reader. The reader, then, can read data from local disk to the memory buffer of the map function.

Hadoop is already shipped with API for analyzing data stored in databases using MapReduce. However, we found using HadoopDB [6] is much easier. Since authors of HadoopDB has already tuned the MapReduce code for the benchmark that we used. Therefore, we use HadoopDB's MapReduce program for performing analysis tasks on PostgreSQL.

For text record decoder, we use the same immutable decoder presented in [14][15][6]. In Hadoop, a text record is stored as a line in the file with key being the offset and value consists of all fields separated by a field delimiter, such as comma or vertical bar. The immutable decoder decodes the value of a record by first splitting the value into a set of immutable string fields and then converts each immutable string field into a type.

We implement a mutable text decoder ourselves. The mutable text record decoder treats the input string (value of a record) as a byte array. It iterates each byte in the array to find field delimiters and converts bytes between two successive field delimiters into the desired type.

For binary decoder, we use the same immutable `Writable` decoder released by the authors of [14] and implement our own mutable decoder by using of Hadoop's API. We use the compiler shipped with Google's protocol buffer for generating its immutable decoder. The compiler could not generate an immutable decoder for protocol buffer, so we implement it by using Google's API directly. The mutable decoder of BDB binary record format is implemented by using of BDB tuple binding API.

We store binary tuples in our own `KeyValueSequenceFile` storage format. The `KeyValueSequenceFile` follows read-optimized row storage format [12][11]. `KeyValueSequenceFile` stores records together, one after another, in a set of pages. The page size is 128KB with a five bytes header storing the offset of the first record and the offset of the last record in the page respectively. We found that our `KeyValueSequenceFile` is much easier than Hadoop's `SequenceFile` for file splitting. Hadoop is able to automatically split the file into a set of file chunks and launch corresponding maps to process, one map for each file chunk. The split point could be happened at any place in the file. Thus, the map function should be able to deal with this and correctly locate the record boundary. `KeyValueSequenceFile` is much easier than `SequenceFile` to perform this task since it stores records in fixed size pages. Thus, the boundary only occurs at multiples of page size.

We implement a range-index scheme described in Section 2.2. The index is built by a MapReduce job called Indexer. The Indexer takes a set of files stored in HDFS as input and outputs a set of files back to HDFS in terms of a sort key and a partition key. The map functions of Indexer iterate each record in the input files and shuffle the records to reducers according to the user specified partition key. Each reduce function of the Indexer collects all records emitted by map functions, sorts them according to the user specified sort key, and writes two output files to HDFS with one sorted data file and one index file including all index entries. The indexer can output both text and binary records.

We modify Hadoop's `MapTask` implementation to support fingerprinting based sort. When an intermediate record is emitted by the map function and is placed in the output buffer, we store a fingerprint of the key in an array. When `MapTask` sorts records in the output buffer, we instruct `MapTask` to compare fingerprints first. If two keys agree on the same fingerprint, `MapTask` then performs an additional comparison for comparing original keys. The fingerprinting function we used is djb2 described in [3]

5. BENCHMARK

This Section reports our performance evaluation of Hadoop, an open source MapReduce implementation, under various settings.

5.1 Benchmark Environment

We run all experiments on Amazon EC2, an elastic computing Cloud, with large instances. Each EC2 large instance equips with 7.5 GB memory, 2 virtual cores, and two disks with 850 GB instance storage (2×420 GB plus 10 GB root partition), and runs 64-bit Fedora 8 Linux Operating System. EC2 automatically mount one 420 GB disk to `/mnt` when the instance is booted. The other 420 GB disk needs to be manually mounted.

We found the disk I/O performance of EC2 is not stable. According to `hdparm`, the buffered reads range from 100∼140 MB/sec (when the instance is launched in off-peak hours) to 40∼70 MB/sec (when the instance is launched in peak hours). In order to make the benchmark results consis-

[2]This is performed by examining the IP address of the reader

tent, we try our best to launch instances in off-peak hours. In average, the disk performance (buffered reads) that we measured is approximately 90∼95 MB/sec in conducting analytical tasks. We also measured that the network speed of EC2 is approximately 100MB/s. We use Hadoop v0.19.2 for all experiments. The Java system we used is 1.6.0_16. For Berkeley DB, we use Berkeley DB Java Edition v4.0.9. For protocol buffer, we use its latest version 2.3.0.

5.2 Hadoop settings

We have tried a lot of configurations for Hadoop and choose the configuration with best observed performance. Finally, we set 1) the block size of HDFS to be 512 MB, 2) the heap size of JVM running the Map/Reduce task to be 1024 MB and enforce the JVM running in server mode with (`-server`), 3) the I/O file buffer size to be 128 KB, 4) the memory used for map-side sort to be 512 MB and the space ratio for intermediate record metadata to be 0.25, 5) merge factor to be 300.

We also set each TaskTracker node to run two map tasks and one reduce task. We configure the HDFS to strip data to two disks. To speedup the reducer-side merge, we set input buffer percent to be 1.0 so that all heap memory of a reducer can be used for holding merge results. We finally store data in HDFS with no replication.

For HadoopDB settings, we carefully follow the instructions presented in [6]. Thus, we use PostgreSQL version 8.2.5, set shared buffers to 512MB. The working memory size is 1GB. We store data in PostgreSQL without compression. We also set the number of concurrent map tasks to one according to the suggestions in [6].

We run all experiments three times and report the average execution time. We run all benchmarks on a 100 nodes cluster with one additional node acting as a JobTracker and a NameNode. For micro-benchmark, namely I/O modes and record parsing benchmark, we only run one slave node. For analytical tasks, we benchmark performance on cluster sizes of 10, 50, 100 nodes, except for the Grep task. We found generating 1TB data in 10 and 50 nodes takes too much time. So, we only report the results for Grep task on 100 nodes. We found the instances launched on EC2 is not quite stable. Thus, we only report results for trials where all nodes are available, operating correctly. We have no Parallel Databases at hand. However, since we run the same benchmark with [14] and [15], we reproduce the performance numbers presented in [15] on our figures as a performance estimation for two Parallel Databases: DBMS-X and Vertica [5]. The authors of [15] claim that these performance numbers represent the best performance that is observed (as of August 2009). Note that these numbers are only available on 100 nodes.

5.3 Datasets

The benchmark dataset we used for the performance evaluation of Hadoop is identical to [14]. Particularly, we choose three datasets: Grep, Rankings, and UserVisits. The schemas of these three datasets are as follows.

```
CREATE TABLE Data(
   key VARCHAR(10) PRIMARY KEY,
   field VARCHAR(90) );

CREATE TABLE Rankings(
   pageURL VARCHAR(100) PRIMARY KEY,
   pageRank INT,
   avgDuration INT );

CREATE TABLE UserVisits(
   sourceIP VARCHAR(16),
   destURL VARCHAR(100),
   visitDate DATE,
   adRevenue FLOAT,
   userAgent VARCHAR(64),
   countryCode VARCHAR(3),
   languageCode VARCHAR(6),
   searchWord VARCHAR(32),
   duration INT );
```

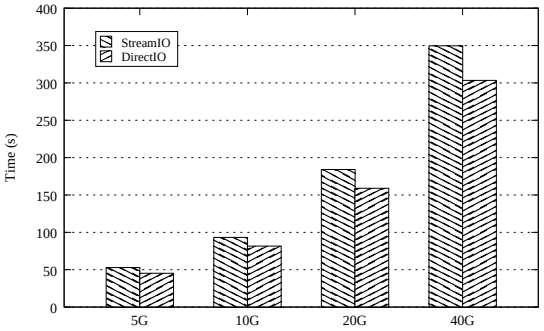

Figure 1: Performance comparison of direct I/O with streaming I/O

We also use the same data generators presented in [14] to generate data for all data analytical tasks. For Grep task, we generate two datasets following the settings in [14] for a 100 nodes cluster: 1TB per cluster and 535MB per node. For other data analytical tasks, we generate 155 million records (20GB) and 18 million Rankings records (1GB) as plain texts in each node. These settings are also identical to [14].

5.4 Results for different I/O modes

We first evaluate the performance of different I/O modes using HDFS. We conduct this micro-benchmark on one node cluster. In this setting, all data are stored on the unique node. Thus, map functions always perform local reads. We use a DFS I/O tool offered by Hadoop to generate dataset. This I/O tool writes random bytes to a HDFS file. Note we do not concern the concrete content of the file and only evaluate the performance of data reading. We generate four datasets: 5GB, 10GB, 20GB, and 40GB. We launch a No-Op MapReduce job on each dataset and measure the execution time of the launched job. The No-Op job only contains a map function and has no reduce function. The map function repeatedly reads a 128KB data block from HDFS into memory without further processing. Thus, in addition to the cost introduced by MapReduce framework, data reading is the only cost. Figure 1 shows the result of this benchmark. From Figure 1, we can clear see that direct I/O outperforms streaming I/O by approximately 15%. This performance gap holds for all four datasets.

5.5 Results for record parsing

We evaluate the performance of different decoding schemes. The structure of Grep record is simple. According to the schema, each Grep record is of 100 bytes, consisting of a unique key in the first 10 bytes, followed by a 90-byte random value. Thus, decoding a Grep record is straightfor-

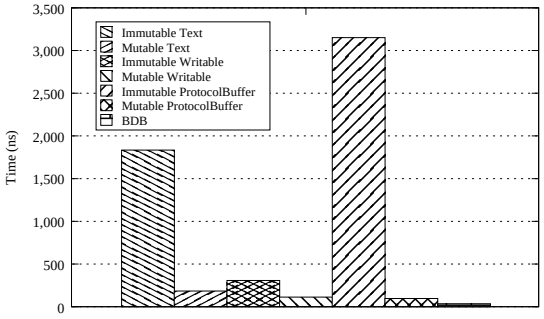

Figure 2: Performance of Decoding Rankings Records

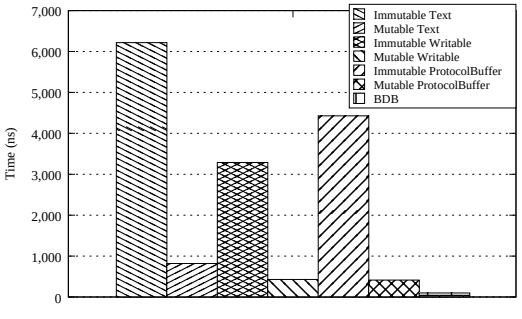

Figure 3: Performance of Decoding UserVisits Records

ward. We only report the results for decoding Rankings and UserVisits records. The text record of Rankings and UserVisits has of the same format. Each record occupies a line in the file and fields are separated by a vertical bar. We evaluate the immutable text decoder used in [14] and our own mutable text decoder. We also evaluate immutable and mutable decoders for `Writable`, Protocol Buffer, and Berkeley DB's tuple binding API. Totally, we evaluate seven decoders

In order to obtain the accurate performance of decoders, we adopt a different setting for this evaluation. We run all benchmark program in stand alone Java processes instead of MapReduce jobs. Thus, we have no additional overhead introduced by MapReduce framework. For text decoders, we first load all data that will be decoded into memory and then call decoders to decode records loaded into the memory buffer. As a result, we have no disk I/O overhead. For binary decoders, we first read records from plain text files and then store the encoded records in the memory buffer. After this loading process, the benchmark program call the corresponding decoders to decode records in the memory. Finally, we evaluate all seven decoders on decoding 8 millions Rankings records and 4 millions UserVisits records. The raw data of each dataset (Rankings and UserVisits) is roughly 512MB. This workload is representative since in real MapReduce jobs, map functions typically process 256MB or 512MB data chunk.

The performance results for the seven decoders for this task is shown in Figure 2 and Figure 3. We can clearly see that in spite of decoding scheme, the mutable decoder is almost faster than its immutable counterpart by a factor of 10. The mutable text decoder also outperforms immutable binary decoders by a large gap. Profiling of the benchmark program shows that immutable decoders spend 80∼90% CPU time on object creations. The huge CPU overheads introduced by object creations make them significantly slower than immutable decoders.

5.6 Grep Task

The Grep task is included as one benchmark proposed by [14] and adopted in [6]. The data set for this task consists of records with fixed length of 100 bytes. The keys are unique and stored as the first 10 bytes, and the rest 90 bytes are treated as values. The task will parse all the records and select the ones with a three-character pattern provided by the user. For this specific benchmark, the search pattern will only appear once in every 10,000 records.

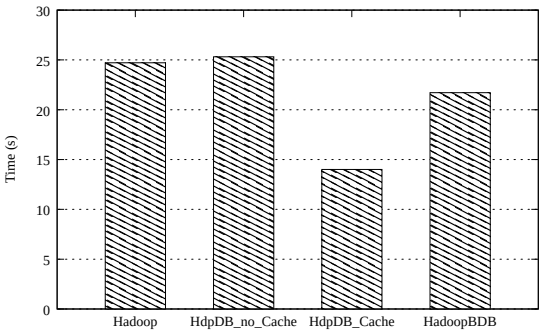

Figure 4: Grep Task Results – 535MB/node

Comparing to other benchmarks proposed in [14], Grep task is relatively straight forward. To save both time and expenses, we only conduct this benchmark on 100 nodes of EC2. Same as [14], we have two categories of Grep tasks: 535MB/node, and 1TB/cluster.

For the first category, we have compared the performance of these applications: Hadoop, HadoopDB (to examine the effect of caches in DBMS, we have reported the results of both with and without caches), and Hadoop with Berkeley DB decoder. Both [14] and [6] provide their source codes online, and we adopts their source codes for the first two experiments. For Hadoop with Berkeley DB decoder, we implemented by our own. Figure 4 demonstrates the results for 535MB/node category. From the results we can see that the DBMS caches significantly shorten the processing time of HadoopDB. But without the caches, the processing time of the three systems are close; and Hadoop with Berkeley

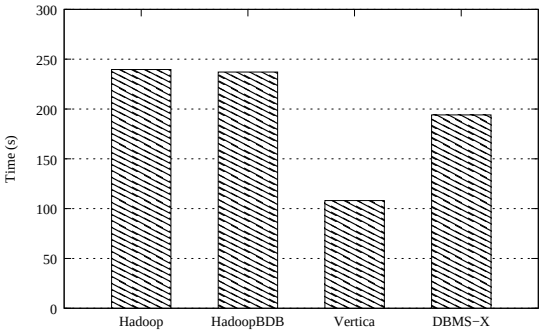

Figure 5: Grep Task Results – 1TB/cluster

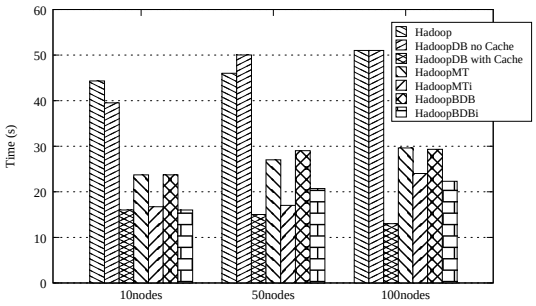

Figure 6: Selection Task Results

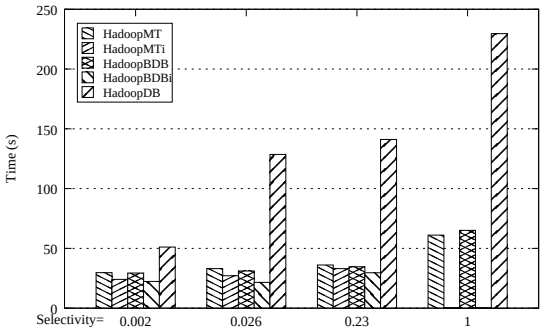

Figure 7: Selection Task with Various Selectivities

DB parser shows slight improvements than the other two.

For 1TB/cluster data set, each node processes 10GB of records. We run the original Hadoop and Hadoop with Berkeley DB parser. Figure 5 shows the results. We also reproduce the performance of Vertical and DBMS-X presented in [15] in the result figure for comparison. In such large dataset, the performance of all four systems are close.

5.7 Selection Task

The selection task operates on Rankings dataset. The Rankings table is initially generated on each node, which consists of 18 million of records, approximately 1GB. There are three columns in this table, and each attribute is separated by a vertical bar. In this task, the pageURL with its corresponding pageRank higher than a user-specified threshold will be selected. The benchmark sets the threshold to 10, which will select approximately 36,000 records out of each node.

To process such dataset, Hadoop needs to do an entire data scan in order to select the corresponding results. Different from Hadoop, the database system in HadoopDB can take advantage of its index, and identify the valid records. Same as [6], we build a clustered index on the pageRank column. We have also implemented Hadoop with mutable text decoding (HadoopMT), and Hadoop with Berkeley DB parser (HadoopBDB); each of the two approaches also comes with an alternative implementation with the indexing features that we have discussed in Section 2.2 (we denote them as HadoopMTi and HadoopBDBi respectively).

We have conducted the experiments on 10, 50, and 100 node clusters on Amazon EC2. Figure 6 illustrates the results. We notice that the original Hadoop has least satisfactory result, due to its inefficient immutable text decoding techniques. For HadoopDB, even though it has built-in indexing features, the performance is close to Hadoop when we clean the DBMS's cache every time. However, when the cache is kept, the performance of HadoopDB improved significantly. If there is nothing in the cache, the DBMS need to scan the data on the disk to generate the result set, which is similar to the execution in Hadoop; but when the result is cached, the DBMS does not need to scan the disk to get the result. We have recorded the average time for map tasks [3] on 100-node cluster, the value is 33 seconds for the execution without cache, but only 3 seconds for the one with cache.

The Rankings table uses vertical bar as separator, and by adopting mutable text decoding, the performance of Hadoop (HadoopMT) is improved significantly. Similar performance improvements can be achieved by adopting Berkeley DB's parser (HadoopBDB). When we use indices on the two systems, the performance has another improvement, as HadoopMTi and HadoopBDBi show. This time the performance is gained by reducing the number of data chunks that need to be scanned.

The original benchmark sets the threshold to 10, which approximately selects 36,000 out of 18,000,000 records on each node (selectivity=0.002). This selectivity is very high, making index in database efficient for this kind of processing. However, with the selectivity becomes lower, more records will be selected, causing the cache insufficient to hold all the results. Moreover, we intend to examine whether the different systems are scalable when the selectivity becomes lower. Therefore, we choose selectivity to 0.026, 0.23, 1, and redo the experiments. We observe that when selectivity reaches 23%, the influence of database cache can be neglected. Therefore for HadoopDB we only recorded the results without cache. Figure 7 shows the results. We can see that both HadoopMT and HadoopBDB, together with their corresponding implementation with indices, are scalable when selectivity becomes lower. But for HadoopDB, the execution time of the database decreases greatly when the result sets become larger, causing HadoopDB less scalable than the other systems. Note that when selectivity=1, the entire dataset is selected; therefore HadoopMTi and HadoopBDBi also need to do full data scan. The index structure cannot be used to prune any unnecessary data chunks; thus we do not record their execution time.

5.8 Aggregation Task

This Section evaluates the performance of MapReduce system on calculating aggregations on the UserVisits table. This analytical task consists of two subtasks called Large Aggregation Task and Small Aggregation Task respectively. Regardless of the number of nodes in the cluster, Large Aggregation Task always produces 2 millions records and Small Aggregation Task produces 2,000 records. The SQL commands of Large Aggregation Task and Small Aggregation Task is as follows:

Large:
```
SELECT sourceIP, SUM(adRevenue)
  FROM UserVisits GROUP BY sourceIP;
```

Small:
```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
  FROM UserVisits GROUP BY SUBSTR(sourceIP, 1, 7)
```

[3] in HadoopDB, since the execution is pushed to DBMS, the map time is close to database execution time

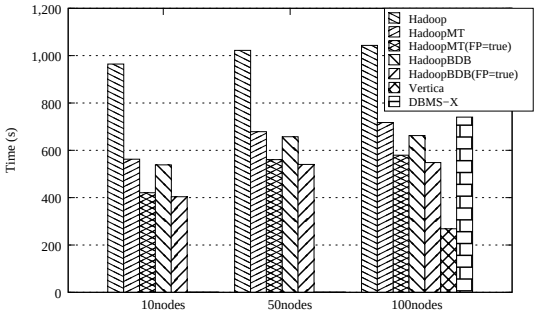

Figure 8: Large Aggregation Results (2.5 million Groups

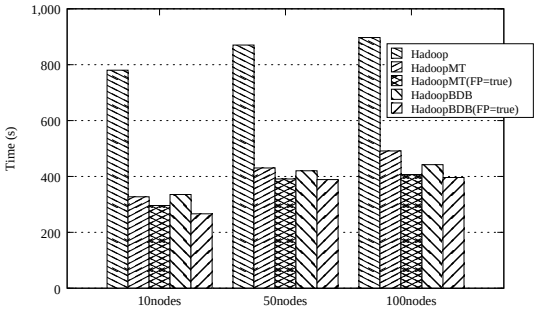

Figure 9: Small Aggregation Results (2,000 groups)

The MapReduce program performing this task consists of both a map function and a reduce function. The map function iterates each record in the UserVisits table and emits the sourceIP (Large Aggregation Task) or its first seven bytes (Small Aggregation Task) and adRevenue field as an intermediate record (key-value pair). The reduce function merges all intermediate records and sums up all adRevenues for each key. To reduce the intermediate records that will be shuffled, a standard practice in MapReduce programming is using a combiner function in map-side to perform a partial aggregation.

We evaluate the performance of MapReduce programs with five settings. First, the original MapReduce program described in [14] is used as a baseline. This MapReduce program adopts immutable text decoder for parsing UserVisits recrods and is denoted as Hadoop in the results figure. Second, we evaluate the performance of a MapReduce program using mutable text decoder. This program is denoted as HadoopMT. Third, we evaluate the performance of MapReduce program which utilizes BDB API for decoding binary records. This setting is denoted as HadoopBDB. For HadoopMT and HadoopBDB, we also evaluate two additional settings by enabling fingerprinting based key-comparison. These two additional settings are denoted as HadoopMT(FP=ture) and HadoopBDB(FP=true) respectively. For all five settings, we set the number of reducers to be equal to the number of salve nodes in the cluster.

Figure 8 and Figure 9 present the results for this task. In Figure 8, for a 100 nodes cluster, we also reproduce the performance numbers of Vertical and DBMS-X from [15] as a comparison. From the Figure 8 and Figure 9, we can clearly see that data parsing/decoding contributes a large part of overall execution time of orignal MapReduce job. By removing the cost of data decoding, the job performs almost two times faster. We also see that fingerprinting based sorting can even reduce the total execution time by 20% to 25% in Large Aggregation Tasks. However, for small aggregation task, fingerprinting based sort only contributes a small performance improvements since at this case the total number of unique keys is small. Thus fingerprinting based comparison could not reduce the cost key comparison too much. Again, we found Hadoop's scheduling strategy introduces non-trivial performance overhead. The scheduler is too aggressive for assigning new map tasks to TaskTrackers and thus is sensitive to processing speed of slave nodes. We found that when the MapReduce job is approaching to end, Hadoop aggressively assigns unprocessed data stored on slow nodes to fast nodes who have finished processing their own data. This aggressive assignment strategy results in a lot of non-local maps. Regardless of the number of nodes in the cluster, data-local maps of HadoopMT or HadoopBDB with fingerprinting enabling are able to complete in 20 seconds. However, non-local maps may take 160 seconds to finish. These outliers (non-local maps) slow down the whole job by approximately 30%.

5.9 Join Task

Join task is the final data analytical task that we used for this performance study. This task consists of two subtasks. The SQL commands that described this task are as follows.

```
SELECT INTO Temp sourceIP, AVG(pageRank) as avgPageRank,
            SUM(adRevenue) as totalRevenue
FROM Rankings as R, UserVisits as UV
WHERE R.pageURL = UV.destURL
          AND UV.visitDate BETWEEN Date('2000-01-15')
                       Date() GROUP BY UV.sourceIP;

SELECT sourceIP, totalRevenue, avgPageRank
   FROM Temp
   ORDER BY totalRevenue DESC LIMIT 1;
```

For this task, we evaluate the performance of MapReduce jobs with three different implementations. Again, the MapReduce program presented in [14] is used as a baseline and is denoted as Hadoop in the results figure. This implementation performs the join task through three separated MapReduce jobs. The first MapReduce job filters UserVisits records that are outside of the desired data range and joins the qualified UserVisits records with Rankings records on pageURL field. The second MapReduce job takes the output of the first MapReduce job as input and compute aggregation. Finally, the third MapReduce job sorts the output of the second job and produces the final result.

In addition to this baseline implementation, we also evaluate two alternative implementations which utilize range index. These two additional MapReduce implementations only differs in decoding scheme. The first implementation adopts mutable text record decoding is denoted as HadoopMTi in the results figure. The second implementation uses BDB API for record decoding and is denoted as HadoopBDBi. These two additional implementations use the same execution strategy and launch one MapReduce job to perform the join task. The map function of the MapReduce job first utilizes a range-index built on visitDate field of UserVisits table for filtering UserVisits records and load qualified records into in-memory hash table. Then the map function iterates each Rankings record and joins the Rankings

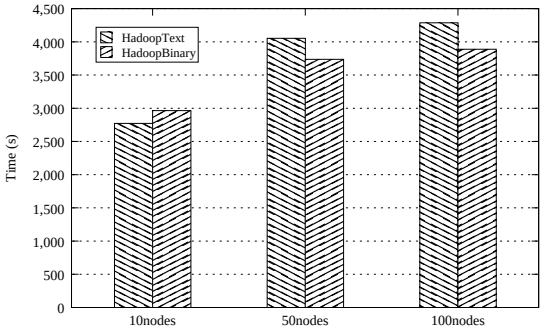

Figure 10: Index Building Times

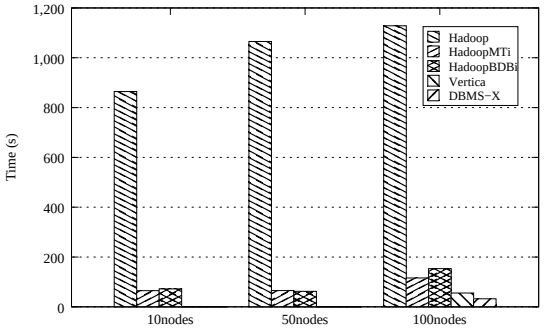

Figure 11: Join Task Results

record with UserVisits record. Finally, the map function emits the joined record as intermediate output and shuffle them to reduce functions. The reduce function then simply perform aggregation on the joined records and output the record with the largest totalRevenue. Figure 10 reports the time for building range-index on visitDate field for UserVisits dataset. Figure 11 shows the performance of all three implementations.

6. RELATED WORK

MapReduce is first proposed by Dean and Ghemawat in [8] as a programming model for processing and generating large data sets. MapReduce is original designed for construction of the inverted index. The ability of using MapReduce as a data analysis tool for processing relational queries is demonstrated in [17][16][13][7][10].

In [14] and [15], the authors compared MapReduce with a traditional data analysis tool, namely a Parallel Database. This work compares two kinds of systems from a number of dimensions, including schema support, programming model, flexibility, execution strategy, fault tolerance and so on. To the end, the authors introduced a benchmark consisting of five data analytical tasks for evaluation the performance of the two kinds systems. The authors found although the process to load data into and tune the execution of Parallel DBMSs took much longer time than MapReduce system, the performance of Parallel DBMSs is stinkingly better. The authors also speculate the possible causes of the performance gap. For example, record parsing is recognized as a contributing factor [15]. Since many speculated causes are related to the architectural design of MapReduce, e.g., record parsing, the natural question is that whether the poor performance of MapReduce is due to the fact that MapReduce is designed to be flexible and scalable and yet it definitely tradeoffs performance. Our work is mainly motivated from this question. According to our study, the flexible and scalable design does not put fundamental obstacles on the performance of MapReduce. However, the individual components have to be properly implemented.

During preparation of the paper, we notice the work in [9] published in Jan 2010. This work also figures out three cases that MapReduce is able to utilize index. This idea is identical to us. However, our work is an independent work and we backup out claim with benchmark data. This also discusses the problem of record parsing. It suggests MapReduce users to avoid using text format and prefers protocol buffers for encoding and decoding binary structured records. However, according to our study, the problem of records parsing actually does not stems from record storage format, text or binary. The problem is actually closely related to the decoding scheme. Even protocol buffer is adopted, as the default output of protocol buffer's compiler is an immutable record parser, the performance is still slow. If the user needs to process millions of records, he/she should use mutable decoder no matter whether the record is encoded as text or binary layout.

7. CONCLUSION

This paper conducts an in-depth performance study of MapReduce on its open source implementation, Hadoop. We figure out four factors that affect the performance of MapReduce significantly and investigate alternative implementation strategies for each factor. To the end, we evaluate the performance of MapReduce with a representative combinations of those four factors on a benchmark consisting of four kinds of analytical tasks. The results show that with proper implementation, the performance of MapReduce can be significantly improved by a factor of 2.5 to 3.5 and approaches to Parallel databases. This result indicates that a system that achieves scalability and flexibility does not necessarily sacrifice performance. We believe the experimental results would be useful for the future development of MapReduce based data processing systems.

8. REFERENCES

[1] http://code.google.com/p/protobuf.
[2] http://developer.yahoo.net/blogs/hadoop/2008/09/.
[3] http://www.cse.yorku.ca/ oz/hash.html.
[4] http://www.oracle.com/database/berkeley-db/je/index.html.
[5] http://www.vertica.com.
[6] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *VLDB*, Lyon, France, 2009.
[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.
[9] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[10] D. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *VLDB*, 2008.

[11] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[12] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 1(1):502–513, 2008.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[14] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*. ACM, June 2009.

[15] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wychoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[17] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.