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 has been widely used for large-scale data analysis in the Cloud. The system is well recognized for its elastic scalability and fine-grained fault tolerance although its performance has been noted to be suboptimal in the database context. According to a recent study [19], Hadoop, an open source implementation of MapReduce, is slower than two state-of-the-art parallel database systems in performing a variety of analytical tasks by a factor of 3.1 to 6.5. MapReduce can achieve better performance with the allocation of more compute nodes from the cloud to speed up computation; however, this approach of "renting more nodes" is not cost effective in a pay-as-you-go environment. Users desire an economical elastically scalable data processing system, and therefore, are interested in whether MapReduce can offer both elastic scalability and efficiency.

In this paper, we conduct a performance study of MapReduce (Hadoop) on a 100-node cluster of Amazon EC2 with various levels of parallelism. We identify five design factors that affect the performance of Hadoop, and investigate alternative but known methods for each factor. We show that by carefully tuning these factors, the overall performance of Hadoop can be improved by a factor of 2.5 to 3.5 for the same benchmark used in [19], and is thus more comparable to that of parallel database systems. Our results show that it is therefore possible to build a cloud data processing system that is both elastically scalable and efficient.

1. INTRODUCTION

In cloud systems, a service provider delivers elastic computing resources (virtual compute nodes) to a number of users. The details of the underlying infrastructure are transparent to the users. This computing paradigm is attracting increasing interest from both academic researchers and industry practitioners because it enables users to scale their applications up and down seamlessly in a pay-as-you-go manner.

To unleash the full power of cloud computing, it is widely accepted that a cloud data processing system should provide a high degree of elasticity, scalability and fault tolerance. In this paper, we argue that besides the above three features, a cloud data processing system should also deliver efficiency. Even though better performance can be achieved by "renting" more compute nodes from the cloud to speed up computation, this solution is not really cost effective and may in fact offset the benefits of cloud computing. An ideal cloud data processing system should offer elastic data processing in the most *economical* way.

MapReduce [13] is recognized as a possible means to perform elastic data processing in the cloud. Compared to other designs, MapReduce provides a few distinguished features such as:

- The programming model of MapReduce is simple yet expressive. Although MapReduce only provides two functions `map()` and `reduce()`, a large number of data analytical tasks can be expressed as a set of MapReduce jobs, including SQL query, data mining, machine learning and graph processing. The programming model is also independent of the underlying storage system and is able to process various types of data, structured or unstructured. This storage-independent design is considered to be indispensable in a production environment where mixed storage systems are deployed [14].
- MapReduce achieves elastic scalability through block-level scheduling. The runtime system automatically splits the input dataset into even-sized data blocks and dynamically schedules the data blocks to the available compute nodes for processing. MapReduce is proven to be highly scalable in real systems. Installation of MapReduce on a shared-nothing cluster with 4,000 nodes has been reported in [3].
- MapReduce provides fine-grained fault tolerance whereby only tasks on failed nodes have to be restarted.

With the above features, MapReduce has become a popular tool for processing large-scale data analytical tasks. However, the performance of MapReduce is still far from ideal in the database context. According to a recent benchmarking study, Hadoop [4], the open source implementation of MapReduce, is slower than two parallel database systems by a factor of 3.1 to 6.5 [19]. MapReduce can deliver better performance with the use of more compute nodes [14]; however, this scheme is not really cost effective in a pay-as-you-go cloud environment. Users want a data processing

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

system which is not only elastically scalable but also efficient. An interesting question is thus whether MapReduce must trade off performance to achieve elastic scalability.

In this paper, we conduct an in-depth performance study of MapReduce. Following [19], we choose Hadoop as the target MapReduce system for evaluation since it closely resembles Google's original MapReduce implementation. We aim to address the following two questions:

- What factors affect the performance of MapReduce?
- Can we eliminate the negative impact of these factors by proper implementation, and what performance improvement can we gain as a result?

To answer the first question, we consider the impact of the architectural design of MapReduce, including programming model, storage-independent design and scheduling. In particular, we identify five factors that affect the performance of MapReduce: I/O mode, indexing, data parsing, grouping schemes and block-level scheduling. To answer the second question, we conduct extensive experiments on a 100-node cluster of Amazon EC2 with various levels of parallelism using the same benchmark published in [19] and a new benchmark tailored for our comparison purposes. Our findings are summarized as follows:

- While MapReduce is independent of the underlying storage system, it still requires the storage system to provide I/O interfaces for scanning data. We identify two types of I/O modes: direct I/O and streaming I/O. Benchmarking on HDFS (Hadoop Distributed File System) shows that direct I/O outperforms streaming I/O by 10%.
- Like existing database systems, MapReduce can actually exploit the index to improve the performance of data processing. In particular, MapReduce can efficiently process three kinds of index structures: sorted files with range-index, B^+-tree files and database indexed tables. Our experiments show that by enabling an index, the performance of MapReduce improves by a factor of 2.5 in the selection task and a factor of up to 10 in the join task, depending on the selectivity of filtering conditions.
- Record parsing is not the source of the poor performance of MapReduce. Our finding is somewhat different from those reported in [19], [20] and [10]. In this study, we evaluate two kinds of record decoding schemes: mutable decoding and immutable decoding. We find that only immutable decoding introduces high performance overhead. To handle database-like workload, MapReduce users should strictly use mutable decoding. We show that a mutable decoding scheme is faster than an immutable decoding scheme by a factor of 10, and improves the performance of MapReduce in the selection task by a factor of 2.
- The MapReduce programming model focuses on data transformation. Data transformation logic is specified in the `map()` and `reduce()` functions. However, the programming model itself doest not specify how intermediate data produced by `map()` functions are grouped for `reduce()` functions to process. The default sort-merge grouping algorithm is not efficient for certain kinds of analytical tasks such as aggregation and join. We observe that different grouping algorithms can significantly improve performance, but the Hadoop core may need to be modified for an efficient grouping algorithm to be implemented.
- Block-level scheduling incurs considerable overhead in MapReduce. In general, the more data blocks to schedule, the higher the cost the scheduler will incur. A micro-benchmark shows that the processing time for scanning a 10GB file with 5GB block size is more than thrice faster than scanning the same file with 64MB block size. Furthermore, the scheduling algorithm, which determines the assignment of map tasks to the available nodes, also affects performance. The current scheduling strategy in Hadoop is sensitive to the processing speed of slave nodes, and may slow down the execution time of the entire job by 20%∼30%.

In summary, by carefully tuning the above factors, the overall performance of MapReduce (Hadoop in this paper) can be improved by a factor of 2.5 to 3.5. This means that, contrary to some recent studies, MapReduce-based systems are not inferior to parallel database systems in terms of performance; instead, they can offer a competitive edge as they are elastically scalable and efficient.

The rest of this paper is organized as follows: Section 2 discusses the factors that affect the performance of MapReduce. In Section 3, we show the combinations of factors that we will evaluate in the experiments. Section 4 presents the implementation details. We present our benchmark results in Section 5. We briefly review related work in Section 6, and we conclude the paper in Section 7.

2. PERFORMANCE FACTORS

In this section, we conduct a performance profile for MapReduce. We start our analysis by reviewing the MapReduce programming model and the execution flow of a MapReduce job. Then, from an architectural perspective, we figure out what design factors may hurt the performance and discuss possible fixes. Implementation details of our improvement on Hadoop will be presented in Section 4.

2.1 MapReduce Background

According to [13], MapReduce is a programming model for processing large-scale datasets in computer clusters. The MapReduce programming model consists of two functions, `map()` and `reduce()`. Users can implement their own processing logic by specifying a customized `map()` and `reduce()` function. The `map()` function takes an input key/value pair and produces a list of intermediate key/value pairs. The MapReduce runtime system groups together all intermediate pairs based on the intermediate keys and passes them to `reduce()` function for producing the final results. The signatures of `map()` and `reduce()` are as follows [13]:

$$\begin{array}{lll} \text{map} & \texttt{(k1,v1)} & \rightarrow \texttt{list(k2,v2)} \\ \text{reduce} & \texttt{(k2,list(v2))} & \rightarrow \texttt{list(v2)} \end{array}$$

A MapReduce cluster employs a master-slave architecture where one master node manages a number of slave nodes. In the Hadoop, the master node is called JobTracker and the slave node is called TaskTracker (since the evaluation is performed on Hadoop, we will use Hadoop's terms in the remaining of the paper). Hadoop launches a MapReduce job by first splitting the input dataset into even-sized data blocks. Each data block is then scheduled to one TaskTracker node and is processed by a map task. The task assignment process is implemented as a heartbeat protocol. The TaskTracker node notifies the JobTracker when it is idle. The scheduler then assigns new tasks to it. The scheduler takes data locality into account when it disseminates data blocks. It always tries to assign a local data block to a TaskTracker. If the attempt fails, the scheduler will assign a rack-local or random data block to the TaskTracker instead. When `map()` functions complete, the runtime system groups all intermediate pairs and launches a set of reduce tasks to produce the final results. We investigate three design components of MapReduce: programming model, storage independent design and runtime scheduling. The three designs result in five factors that affect the performance of MapReduce: grouping schemes, I/O modes, data parsing, indexing, and block-level scheduling. We shall next describe each of them in detail.

2.2 Programming Model

The MapReduce programming model mainly focuses on enabling users to specify data transformation logic (via `map()` and `reduce()` functions). The programming model itself does not specify how intermediate pairs produced by `map()` functions are grouped for `reduce()` functions to process. In the MapReduce paper [13], the designers considered that specifying a data grouping algorithm is a difficult task and such complexity should be hidden by the framework. Therefore, a sort-merge algorithm is employed as the default grouping algorithm and few interfaces are provided to change the default behavior.

However, sort-merge algorithm is not always the most efficient algorithm for performing certain kinds of analytical tasks, especially for those which do not care about the order of intermediate keys. Examples of such tasks are aggregation and equal-join. Therefore, in these cases, we need to switch to alternative schemes. In this paper, we evaluate the performance of adopting alternative grouping schemes to perform aggregation and join tasks. Exhausting all possible grouping algorithms is impossible. Our aim is to study whether "typical" grouping strategies can be efficiently implemented within the MapReduce framework.

For the aggregation task, we evaluate a fingerprinting based grouping algorithm. In particular, a 32-bit integer is generated as the fingerprint of the key for each intermediate key/value pair. When a map task sorts the intermediate pairs, it first compares the fingerprints of keys. If two keys have the same fingerprint, we will further compare the original keys. Similarly, when a reduce task merges the collected intermediate pairs, it first groups the pairs by the fingerprints and then for each group, the key-value pairs are merged via the original keys.

For the join task, we evaluate different join algorithms. Details of these algorithms can be found in Section 5.11.

We find that it is hard to efficiently implement fingerprinting based grouping in `map()` and `reduce()` interfaces. Our final implementation modifies the Hadoop core. The implementation details are described in Section 4. Our experiences show that MapReduce may need to extend its programming model to allow users to specify their customized grouping algorithms.

2.3 Storage Independence

The MapReduce programming model is designed to be independent of storage systems. Namely, MapReduce is a pure data processing system without a built-in storage engine. MapReduce reads key/value pairs from the underlying storage system through a *reader*. The reader retrieves each record from the storage system and wraps the record into a key/value pair for further processing. Users can add support for a new storage system by implementing a corresponding reader.

This storage independent design is considered to be beneficial for heterogenous systems since it enables MapReduce to analyze data stored in different storage systems [14]. However, this design is quite different from parallel database systems. All the commercial parallel database systems are shipped with both a query processing engine and a storage engine. To process a query, the query engine directly reads records from the storage engine. No cross-engine calls are therefore required. It appears that the storage independent design may hurt the performance of MapReduce, since the processing engine needs to call the readers to load data. By comparing MapReduce and parallel database systems, we identify three factors that may potentially affect the performance of MapReduce: 1) I/O mode, the way of a reader retrieving data from the storage system; 2) data parsing, the scheme of a reader parsing the format of records; and 3) indexing.

2.3.1 I/O mode

A reader can choose two modes to read data from an underlying storage system: 1) direct I/O and 2) streaming I/O. Using direct I/O, the reader reads data from a local disk. In this case, the data are directly shipped from the disk cache to the reader's memory though DMA controller. No inter-process communication costs are required. On the other hand, a reader can also adopt the streaming I/O scheme. In this case, the reader reads data from another running process (typically the storage system process) through certain inter-process communication schemes such as TCP/IP and JDBC.

From a performance perspective, we would expect direct I/O to be more efficient than streaming I/O when a reader retrieves data from the local node. This could be a possible reason on the choice of the design of most parallel database systems and the fact that they are shipped with both a query engine and a storage engine. In parallel database systems, the query engine and storage engine run inside the same database instance and thus can share the memory. When a storage engine fills its memory buffer with the retrieved data, it directly passes the memory buffer to query engine for processing.

On the other hand, streaming I/O enables MapReduce execution engine to read data from any processes, such as distributed file system processes (e.g. DataNode in Hadoop) and database instances (e.g. PostgreSQL). Furthermore, if a reader needs to read data from a remote node, streaming I/O is the only choice. The features collectively make MapReduce storage independent.

In this paper, we enhance the HDFS with a direct I/O support and benchmark the reading performance of different I/O modes when a reader reads data from the local node. This benchmark serves for the purpose of quantifying the performance impact of different I/O modes.

2.3.2 *Data Parsing*

When a reader retrieves data from the storage system, it needs to convert the raw data into the key/value pairs for processing. This conversion process is known as data parsing. The essence of data parsing is to decode the raw data from their native storage format and transform the raw data into data objects which can be processed by a programming language, e.g. Java. Since Java is the default language of Hadoop, our discussion is therefore Java specific. However, the insights should also apply to other languages.

There are two kinds of decoding schemes: immutable decoding and mutable decoding. The immutable decoding scheme transforms raw data into immutable Java objects. Immutable Java objects are read-only objects and cannot be modified. One example is Java's string object. According to SUN's document, setting a new value to a string object causes the original string object to be discarded and replaced by a new string object. Hence, in immutable decoding scheme, a unique Java object is created for each record. Therefore, parsing four million records generates four million immutable objects. Since the Java string object is immutable, most text record decoders adopt the immutable decoding scheme. By default, the Google's protocol buffer also decodes records as immutable objects [2].

An alternative method is the mutable decoding scheme. Using this approach, a mutable Java object is reused for decoding all records. To parse the raw data, the mutable decoding scheme decodes the native storage format of a record according to the schema and fills the mutable object with new values. Thus, no matter how many records are decoded, only one data object is created.

We note that the poor performance of record parsing observed in [19], [20] and [10] is most likely due to the fact that all these studies adopt the immutable scheme for decoding text records. The immutable decoding scheme is significantly slower than the mutable decoding scheme as it produces a huge number of immutable objects in the decoding process. Creation of those immutable objects incurs high overheads on CPUs. In this study, we design a micro-benchmark to quantify the performance gap between the the two decoding schemes.

2.3.3 *Indexing*

The storage independent design implies that MapReduce doest not assume the input dataset to have an available index. At the first glance, MapReduce may not be able to utilize indexes [19]. However, we found that there are three methods for MapReduce to utilize indexes for speeding up data processing.

First, MapReduce offers an interface for users to specify the data splitting algorithm. Therefore, one can implement a customized data splitting algorithm, which applies the index to prune the data blocks. In Hadoop, this can be achieved by providing a specific `InputFormat` implementation. This customized data splitting technique can be used in two scenarios: 1) if the input of a MapReduce job is a set of sorted files (stored in HDFS or GFS), one can adopt a range-index to prune redundant data blocks; 2) if each file name of the input files follows certain *naming* rule, such naming information can also be used for data splitting. An example of this scheme is presented in [14]. Suppose a logging system periodically rolls over to a new log file and embeds the rollover time in the name of each log file, then to analyze logs within a certain period, one can filter unnecessary log files based on the file name information.

Second, if the input of MapReduce is a set of indexed files (B^+-tree or hash), we can efficiently process these input files by implementing a new reader. The reader takes certain search condition as input (e.g. a date range) and applies it to the index to retrieve records of interest from each file.

Finally, if the input of MapReduce consists of indexed tables stored in n relational database servers, we can launch n map tasks to process those tables. In each map task, the `map()` function submits a corresponding SQL query to one database server and thus transparently utilizes database indexes to retrieve data. This scheme is first presented in the original MapReduce paper [13] and subsequently demonstrated in a recent work [10].

2.4 Scheduling

MapReduce adopts a runtime scheduling scheme. The scheduler assigns data blocks to the available nodes for processing one at a time. This scheduling strategy introduces runtime cost and may slow down the execution of the MapReduce job. On the contrary, parallel database systems benefit from a compiling-time scheduling strategy [20]. When a query is submitted, the query optimizer generates a distributed query plan for all the available nodes. When the query is executed, every node knows its processing logic according to the distributed query plan. Therefore, no scheduling cost is introduced after the distributed query plan is produced. In [20], the authors note that the MapReduce's runtime scheduling strategy is more expensive than the DBMS's compiling-time scheduling. However, the runtime scheduling strategy enables MapReduce to offer elastic scalability, namely the ability of dynamically adjusting resources during job execution.

In this paper, we quantify the impact of runtime scheduling cost through a micro-benchmark and a real analytical task, i.e., Grep. We find that runtime scheduling affects the performance of MapReduce in two ways: 1) the number of map tasks need to be scheduled and 2) the scheduling algorithm. For the first factor, it is possible to tune the size of data blocks to alleviate the cost. For the second factor, more research work is required to design new algorithms. Detailed discussions will be presented in Section 5.

3. PRUNING SEARCH SPACE

Different possible combinations of the above five factors result in a huge search space. Moreover, as all benchmarks are conducted on Amazon EC2, the budget for the use of EC2 also imposes a constraint on us. Thus, we narrow down the search space into a representative but tractable set.

First, we limit the performance evaluation of MapReduce on two kinds of storage systems, a distributed file system (namely HDFS) and a database system (namely PostgreSQL). These two storage systems should sufficiently represent the typical input data sources of MapReduce applications. For HDFS, we report the performance of all tasks in a designed benchmark. For PostgreSQL, we only perform a subset of the tasks, as loading data into PostgreSQL and building cluster index consumes a lot of time. We only run the tasks on those datasets whose loading time is within three hours.

Second, we choose proper record formats and decoding schemes for the evaluation. For text records, we only consider one encoding format where each record occupies one line and the fields are separated by a vertical bar, i.e., "`|`". This record format is used with both mutable and immutable decoding schemes. For binary records, we consider three popular record formats: 1) Hadoop's `Writable` format, 2) Google's protocol buffer format [2], and Berkeley DB's record format [8]. We conduct a micro-benchmark to evaluate the performance of parsing records encoded in these three formats with different decoding schemes. The combination of the record format and the decoding scheme that achieves the best performance is chosen for the final benchmark.

Finally, we implement a `KeyValueSequenceFile` data structure to store binary records in HDFS files. We do not use Hadoop's `SequenceFile`, since it only stores records in `Writable` format. We do not consider data compression in this paper as Hadoop does not support compression well at this moment. The impact of data compression to the performance of MapReduce will be investigated in the future work.

4. IMPLEMENTATION DETAILS

We use Hadoop v0.19.2 as the code base[1]. All MapReduce programs are written in Java. The source code is available in our project's website [6].

4.1 Handling Databases

Hadoop provides APIs (e.g. `DBInputFormat`) for the MapReduce programs to analyze data stored in relational databases. However, HadoopDB [10] is more efficient and easy to use. Therefore, HadoopDB is adopted for illustrating the performance of MapReduce on processing relational data stored in database systems. We tune HadoopDB based on the settings described in [10]; we have also benefited from pointers given by two of the authors through email exchanges.

4.2 Direct I/O in HDFS

HDFS stores a large file among the available DataNodes by chopping the file into even-sized data blocks and storing each data block as a local file in one DataNode. We enhance HDFS with the support of direct I/O when a reader reads data from a local DataNode. In the HDFS, a data accessing process works as follows. A reader first calls `open()` to inform the HDFS which file it intends to read, and then it invokes `seek()` to move to the file position where data retrieval starts. Following that, the reader invokes a sequence of `read()` to retrieve data.

When the reader opens a file, it contacts the NameNode for gathering information of the data blocks belonging to that file. Then, the reader checks which data block contains the data of interest based on the file position provided by `seek()`. Finally, the reader makes the connection to the DataNode storing the data block and begins data streaming. To support direct I/O, we only change the final step. When a DataNode receives a data retrieval request, it first checks whether the request is from a local reader (by examining IP address). If this is the case, the data node passes the local file name of the data block to the reader. The reader, then, reads data from local disk directly. If the request is not from a local reader, the DataNode applies the default streaming I/O mode to handle the data retrieval request. Using this approach, we implement a hybrid I/O mode in HDFS: direct I/O for local reads and streaming I/O for remote reads.

[1]The Hadoop (v0.19.2) we evaluated in this paper is slightly different from the Hadoop (v0.19.0) studied in [19]. There is no difference between the two versions in performance. Compared to v0.19.0, Hadoop v0.19.2 only fixes some bugs.

4.3 Data Parsing

For the immutable decoding scheme, we use the same code released in [19] for decoding text and `Writable` records. We also use the compiler shipped with Google's protocol buffer for generating the record decoder. According to the protocol buffer's document, the decoder decodes a record as an immutable record.

We implement mutable decoding schemes for text, `Writable`, protocol buffer and Berkeley DB's record format. For the text decoding, the decoder treats the input record as a byte array. It iterates each byte in the array to find the field delimiters and converts bytes between two successive field delimiters into the corresponding data type defined by the schema. To avoid using a Java string (as it is an immutable object) as the target data type, we implement a `VarCharWritable` data structure, which is used for decoding text strings.

Binary records are stored in our own data structure, `KeyValueSequenceFile`, which is a read-optimized row store [16][17]. `KeyValueSequenceFile` stores records sequentially, one after another, in a set of pages. The page size is 128KB with a five-byte header recording the offsets of the first and last record in the page. Our `KeyValueSequenceFile` is efficient for data splitting since the splitting boundary only occurs at multiples of page size (as `KeyValueSequenceFile` stores records in fixed-size pages).

4.4 Indexing

We evaluate the performance of MapReduce on processing two kinds of index structures: 1) sorted files with range index and 2) indexed database tables. For the second case, we apply HadoopDB as the data storage engine.

We implement a simple range-indexing scheme for sorted files. Suppose a file stores sorted key/value pairs, the indexing scheme creates an index entry for every B-bytes data page where B is defined by the user and is not necessarily equal to the block size of HDFS. 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 page, and D_s is the offset of the data page in the HDFS file. Index entries are sorted on k_b and stored in an index file.

To utilize the range-index for processing sorted files, we develop a data splitting algorithm in our own `InputFormat` implementation. The data splitting algorithm takes a key range as input and searches the index entries to locate which data pages contain the records of interest. For each data page, the splitting algorithm fills a Hadoop's `InputSplit` data structure with offset and length of that data page. The `InputSplit`s are then collected and will be used by the JobTracker to schedule map tasks to process, one `InputSplit` for each map task. Using our approach, only data pages containing the records of interest will be processed. Other data pages are discarded after the data splitting process.

We also plan to evaluate the performance of MapReduce on processing B$^+$-tree files. In this setting, we store Berkeley DB's B$^+$-trees in HDFS files, one file for each B$^+$-tree. Each file will then be processed by one map task. We implement a Berkeley DB reader which accepts a filtering condition and uses Berkeley DB's API to retrieve qualified records. However, in each launch, Berkeley DB automatically checks the input file's integrity for recovery purposes. This checking incurs huge runtime overhead. Thus, the result of this scheme is intentionally omitted in this paper.

4.5 Fingerprinting Based Grouping

It is challenging to efficiently implement fingerprinting based grouping in `map()` and `reduce()` functions. In principle, one can emit original key k and its fingerprint p as a compound intermediate key $I = (k, p)$ and provide a special `compare()` function to the system for comparing intermediate keys. The `compare()` function first compares fingerprints of two intermediate keys and if the two fingerprints are the same, a further comparison of the original keys is performed.

However, it is not very efficient since to compare two intermediate keys, a `compare()` functions must be invoked and the function must deserialize the intermediate keys I into k and p for comparison. The costs of function invocation and deserialization are not negligible when we have to sort millions of intermediate pairs.

To alleviate this problem, we modify Hadoop's `MapTask` implementation. When the original key k is emitted by the map function and is placed in the `MapTask`'s output buffer, we store the fingerprint p in an additional array. When `MapTask` sorts intermediate pairs in the output buffer, we compare fingerprints stored in the additional array correspondingly. The fingerprinting function we used is *djb*2 described in [7].

5. BENCHMARKING

In this section, we report our performance evaluation results. Our benchmarking entails the evaluation of seven tasks. Four tasks are the same tasks studied in [19]. The other three tasks are designed for our comparison purposes.

5.1 Benchmarking Environment

The benchmarking is conducted on Amazon EC2 [1]. We use EC2's large instance as our compute node. Each large instance is equipped with 7.5 GB memory, 2 virtual cores and 2 disks with 850 GB storage (2 × 420 GB plus 10 GB root partition). The operating system is 64-bit Fedora 8. The large instance automatically mounts one 420 GB disk to "`/mnt`", when the system is booted. We manually mount the other 420 GB disk to the system to improve the I/O performance.

After some experiments, we find that the I/O performance of EC2 is not stable. According to `hdparm`, the throughput of buffered reads ranges from 100∼140 MB/sec (when the instance is launched at off-peak hours) to 40∼70 MB/sec (when the instance is launched at peak hours). In order to make the benchmark results consistent, we launch instances at off-peak hours over the weekends. The average disk performance that we measured is approximately 90∼95 MB/sec when conducting the experiments. The network bandwidth that we measured is approximately 100MB/s. As presented in Section 4, we use Hadoop v0.19.2 for all the experiments. For the Berkeley DB and protocol buffer, we use their Java Edition v4.0.9 and v2.3.0, respectively. The JVM version is version 1.6.0_16.

5.2 Hadoop Settings

We have tested various configurations of Hadoop to choose the one with the best observed performance. Our configurations are as follows: 1) The JVM runs in the server mode with maximal 1024 MB heap memory for either the map or reduce task; 2) We enable the JVM reuse. 2) The I/O buffer is set to 128 KB; 3) The available memory for map-side sorting is 512 MB; 4) The space ratio regarding to the metadata of the intermediate results is set to 0.25; 5) The merge factor is 300.

Besides, we also set the block size of the HDFS to 512 MB, which is different from the setting (256MB) in the previous study [19][10]. Section 5.5 will illustrate why we choose this setting. In the experiments, each TaskTracker is allowed to run two map tasks and one reduce task concurrently. To speed up the reducer-side merge, we use all the heap memory of a reducer to hold the merged results. We store data in the HDFS with no replication. For all the analytical tasks, we configure HDFS to run in hybrid I/O mode, namely direct I/O for processing local reads and streaming I/O for handling remote reads.

For the HadoopDB settings, we strictly follow the guidelines given in [10]. Thus, we use PostgreSQL version 8.2.5 and store data in PostgreSQL without compression. The shared buffer of PostgreSQL is set to 512MB, while the working memory size is set to 1GB. Furthermore, we did not manage to utilize two disks for PostgreSQL. For HadoopDB experiments, the number of concurrent map tasks in each TaskTracker node is set to one according to [10].

We run all the experiments on an EC2 cluster with one additional node acting as the JobTracker and NameNode. For micro-benchmark (e.g. I/O modes, scheduling and record parsing), we only run one slave node. For analytical tasks, we evaluate the performance of MapReduce programs by varying the cluster size from 10 nodes to 100 nodes. One exception is the Grep task where we only report the results of Grep task on a 100-node cluster. As an EC2's node may fail at anytime, we only report the results when all the nodes are available and operate correctly. Each experiment is run for three times and we report the average result.

5.3 Datasets

For analytical tasks, we use three datasets (`Grep`, `Rankings`, and `UserVisits`) which are identical to [19]. The schemas of these three datasets are as follows.

```
CREATE TABLE Grep(
   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 );
```

We also use the same data generator released in [19] for data generation. For the `Grep` dataset, we generate a 1TB dataset for the 100-node cluster. For `Rankings` and `UserVisits` datasets, we generate 18 million records (∼1GB) and 155

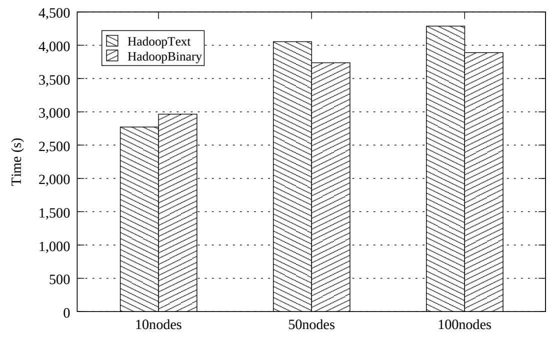

Figure 1: Loading Times – `UserVisits` Dataset

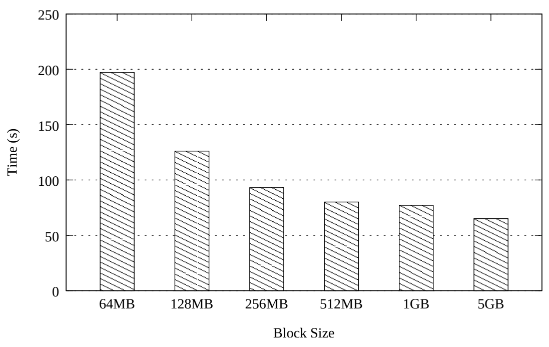

Figure 2: Results for Scheduling Micro-benchmark

million records (∼20GB) respectively in each slave node. These settings are also identical to those used in [19]. The generated datasets are stored as plain text files in the slave nodes.

5.4 Data Loading

This Section describes the procedures for loading the datasets. We first copy the raw data files from each slave node to the HDFS. This simple loading scheme is sufficient for most of the MapReduce programs that we will evaluate in this paper. For `Rankings` and `UserVisits` datasets, to evaluate the performance of MapReduce on processing sorted files with range index, we sort the datasets and build a range-index on them. We develop a data loader to perform this task.

The data loader is a MapReduce job which takes a partition column p and an index column i as input and partitions the input dataset into n partition files. For each partition file the loader sorts the records and builds a range-index on the index column. For `Rankings` dataset, we partition the dataset on `pageURL` column and build a range-index on `pageRank` column. For `UserVisits` dataset, we partition the dataset on `destURL` column and build index on `visitDate` column. We adopt the range-index scheme described in Section 4.4 and set data page size B to 128MB. Figure 1 reports the elapsed time for loading `UserVisits` dataset of two formats: text format (HadoopText) and Berkeley DB's record format (HadoopBinary). The index file size generated is rather small: 0.15KB per partition file for `Rankings` and 1.12KB per partition file for `UserVisits`.

We only load `Rankings` dataset to HadoopDB. The loading process is separated into three phases. First, HadoopDB's Global Hasher is launched to partition the dataset into n files (n is the cluster size) via the `pageURL` column. Then, each slave node downloads a unique partition file. Finally, in each slave node, we load the partition file into PostgreSQL and build a cluster index on `pageRank`. The last phase is performed by first loading the raw partition file into a temp table T in PostgreSQL by a SQL COPY command. Then, we sort T by `pageRank` and store the results as the `Rankings` table. Finally, we build a B^+-tree index on the `pageRank` column of the `Rankings` table[2].

5.5 Scheduling Micro-Benchmark

It is not trivial to directly measure the cost of Hadoop's scheduling. Therefore, we estimate the cost of scheduling by measuring the performance of running a dummy MapReduce job with different numbers of map tasks. The dummy MapReduce job scans through the input dataset without further processing. We implement a special reader for the dummy job. The reader repeatedly reads data from the input dataset until all data are consumed. Each time, the reader reads 128KB data from the HDFS into an array L and generates a random integer I for L. Then, the key/value pair (L, I) is used as the input of the `map()` function. The `map()` function simply discards the input pair, i.e., a no-op operation. There is no `reduce()` function involved in the dummy job.

We generate a 10GB file using `TestDFSIO`, a tool shipped with Hadoop for writing random bytes to HDFS. The data file is split based on different block sizes, from 64MB to 5GB. This strategy results in the number of data chunks (also the number of map tasks) that the scheduler needs to manage ranges from 160 (64MB of each chunk) to 2 (5GB of each chunk). Since the TaskTracker can run two concurrent map tasks, in the 5GB block size setting, the scheduler can schedule two map tasks in one phase. Therefore, the scheduling cost is minimized. We can therefore estimate the scheduling cost by comparing the dummy job's execution time on other settings with this 5GB setting. Hadoop's streaming I/O mode dose not support streaming a data chunk beyond 5GB. Thus, we run the benchmark in direct I/O mode.

Figure 2 shows the execution time of scanning the 10GB file with different block sizes. A larger block size will lead to less number of map tasks. From Figure 2, we clearly see that the performance of the dummy job is significantly affected by the total number of map tasks. The execution time (197s) of processing the 10GB file with 160 map tasks (64MB block size) is more than three times longer than the execution time (65s) of scanning the same file with two map tasks (5GB block size). Increasing the block size improves the scanning performance, but it may also lead to a long failure recovery process. In Figure 2, 512MB block size achieves a good balance. Further increasing the block size will not improve the performance significantly. Therefore, we use 512MB block size in the remaining benchmark studies.

5.6 Comparison of Different I/O Modes

In this experiment, we evaluate the performance of different I/O modes in HDFS. All data are stored on a unique DataNode. We use `TestDFSIO` to generate four datasets: 5GB, 10GB, 20GB, and 40GB. We launch the dummy MapReduce job described above on the four datasets and measure

[2]The procedure of last loading phase was suggested by the HadoopDB's authors.

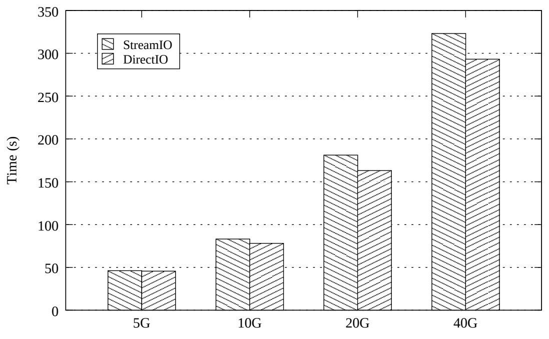

Figure 3: I/O Micro-benchmark Results

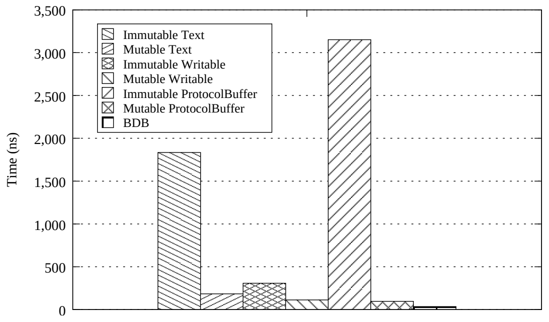

Figure 4: Decoding Results – `Rankings` Dataset

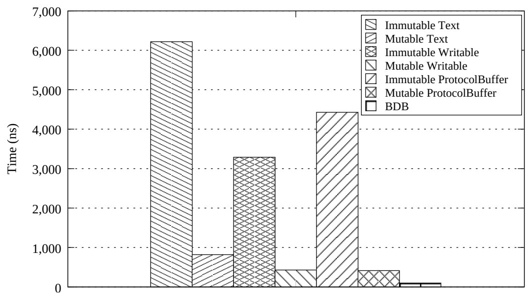

Figure 5: Decoding Results – `UserVisits` Dataset

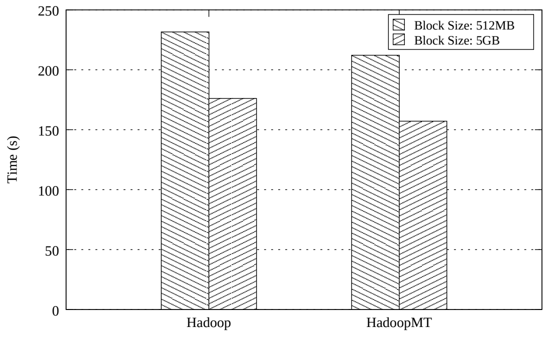

Figure 6: Grep Task Results – 1TB/cluster

the scanning performance.

Figure 3 shows the result of this benchmark. The performance gap between direct I/O and streaming I/O is small, approximately 10%. We conclude that the I/O mode is not the major factor of the poor performance of MapReduce.

5.7 Record Parsing Micro-Benchmark

We now evaluate the performance of parsing different records formats with different decoding schemes. The datasets that we use are `Rankings` and `UserVisits`. For each dataset, we consider four record formats, text format, `Writable` format, protocol buffer and Berkeley DB's record format. For the first three formats, we evaluate both immutable decoding and mutable decoding schemes. For the last format, i.e., Berkeley DB's record format, we only report results of mutable decoding as Berkeley DB does not support immutable decoding scheme. Thus, we evaluate seven decoders in total.

Unlike the other benchmark tasks studied in this paper, we run the record parsing benchmark using a stand-alone Java program instead of a MapReduce job. This is because we need to eliminate other additional overheads introduced by the MapReduce framework. In addition, we read the whole datasets into memory for parsing to eliminate I/O cost. Thus, we finally parse roughly 512MB raw data for each dataset, which is respectively equivalent to 4 millions `UserVisits` records and 8 millions `Rankings` records.

The average time of decoding a record of both datasets is respectively shown in Figure 4 and Figure 5. The results indicate that despite of the record format, the performance of the mutable decoding scheme is always faster than the immutable decoding scheme, by a factor up to ten. Profiling the decoding program further reveals that the large performance gap between immutable decoding and mutable decoding is due to the CPU overhead. Using immutable decoding scheme, the CPU spends 80∼90% time on object creations. For the four mutable decoders, we observe that the performance difference between the text decoder with the other three binary decoders is small, within a factor of two. We conclude that record parsing is not the source of the poor performance of MapReduce. Parsing a text record is not significantly slower than parsing a binary record. The key point for efficient decoding is using the mutable decoding scheme.

5.8 Grep Task

Our first analytical task is the Grep task. For this task, we scan through the input dataset searching for a three-character pattern "XYZ".

We generate a 1TB Grep dataset for the 100-node cluster, namely 10GB data per node. The pattern appears once in every 10,000 records. Therefore, the performance of this work is mainly limited by the sequential scan speed of the data processing system. The MapReduce job that performs Grep task only involves the `map()` function, which searches the pattern in the input value string and outputs the matching records as results.

We evaluate two implementations of the above MapReduce job. One is the original implementation described in [19]. In this implementation, a reader reads a line from the input file and wraps the offset and contents of the line as a key/value pair. In the alternative implementation, as each Grep record is of 100-byte length, the reader reads a 100-byte string from the file and applies the offset and the string as a key/value pair for further processing.

We measure the performance of the above two MapReduce implementations in two settings: 1) 512MB block size and 2) 5GB block size. Using the 5GB setting, the input file in

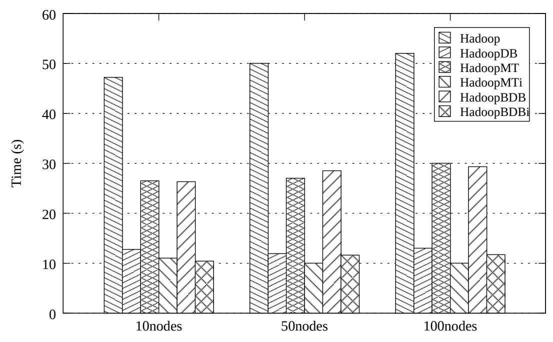

Figure 7: Selection Task Results

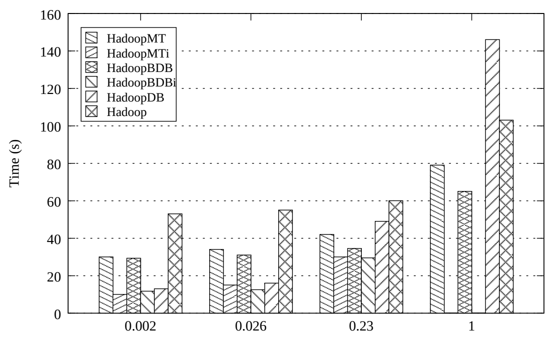

Figure 8: Selection Task with Various Selectivity

each node can be processed by two concurrent map tasks in one phase, resulting in the minimal scheduling cost. We are therefore free to understand the scheduling cost in the real analytical workload.

Figure 6 shows the results of this benchmark. Our alternative MapReduce implementation (e.g. HadoopMT) is slightly faster than the original implementation (e.g. Hadoop) used in [19]. This is because our implementation directly wraps a 100-byte string instead of a line as a record. The latter case requires the reader to search for the end of line character, which incurs additional runtime cost. However, the performance difference between the two implementations is not significant, only 8%. The scheduling cost affects the performance significantly, 30% in Hadoop and 35% in HadoopMT. The scheduling costs are introduced by two factors 1) the number of map tasks to schedule; and 2) the scheduling algorithm. The first factor is studied in Section 5.5, and we shall only discuss the second factor here. When the TaskTracker informs the JobTracker that it is idle, Hadoop's scheduler will assign a map task to that TaskTracker if it holds any unassigned map tasks. However, in the cloud, the processing speed of compute nodes may not be the same. A fast node which completes its tasks quickly will be assigned data blocks from the slow nodes. In this case, the fast node will contend the disk bandwidth with the running map tasks on the slow nodes since it needs to read data from the slow nodes. This resource contention slows down the whole data processing, by 20% (Hadoop) and 25% (HadoopMT) in this task. The scheduling problem we found is similar but not identical to the straggler problem presented in [13]. We turn on the Hadoop's speculative execution strategy, but enabling this option fails to solve the problem. On the contrary, the observed performance is even worse since scheduling many speculative tasks results in more resource contention.

5.9 Selection Task

In the second analytical task, we need to find records in the `Rankings` dataset (1GB per node) whose `pageRank` is above a given threshold. The threshold is set to 10, which results in approximately 36,000 records out of 18 millions records in each node. The SQL command of this task is as follows:

```
SELECT pageURL, pageRank
FROM Rankings
WHERE pageRank > 10
```

The corresponding MapReduce job only involves the `map()` function, which produces qualified records as the output. Three implementation schemes are compared in the experiment: 1) Hadoop, the scheme described in [19]; 2) Hadoop with mutable decoding; 3) Hadoop with mutable decoding and index pruning.

For 2) and 3), we consider processing text record format (HadoopMT and HadoopMTi) and Berkeley DB's record format (HadoopBDB and HadoopBDBi). We also evaluate the performance of HadoopDB on this task. To perform selection task, the `map()` function of HadoopDB's program pushes the SQL query to the PostgreSQL instances for execution and retrieves the results from PostgreSQL instances through JDBC. Therefore, we evaluate the performance of six implementations in total.

Figure 7 illustrates the results of this benchmark. The original Hadoop has the worst performance, due to its inefficient immutable decoding scheme. By adopting mutable text decoding, HadoopMT improves the performance of Hadoop by a factor of two. The same performance is also observed in HadoopBDB. Since the filtering condition is highly selective (0.002), by using index pruning, HadoopMTi and HadoopBDBi improves the performance of their non-index counterparts by a factor of 2.5 to 3.0. The performance of HadoopDB is similar to HadoopMTi and HadoopBDBi.

We are also interested in the performance of HadoopDB. Although the SQL query can be efficiently processed by PostgreSQL, the `map()` function still needs to retrieve the resulting records through JDBC. Unfortunately, the JDBC parses the returned records (encoded in database's native format) as immutable Java objects. Therefore, we expect the performance of HadoopDB to degrade when many records are returned. Consequently, we vary the selectivity of the filtering condition from 0.002 to 1.0 and rerun the experiment to study the scalability of the six implementations. When the selectivity is 1.0, we intentionally remove the results of HadoopMTi and HadoopBDBi since it is impossible to speed up data processing through index in this setting.

Figure 8 summarizes the results for the scalability benchmark. As expected, the performance of HadoopDB is affected by the filtering selectivity. When the selectivity is set to 1, namely the whole dataset is returned, the performance of HadoopDB is worse than that of Hadoop by a factor of 40%. The main reason for this result is that HadoopDB only launches one map task in each node and thus limits the degree of parallelism (e.g., only one out of the two disks on the EC2 node is used for I/O). However, in the long term, we expect HadoopDB can solve the "multiple concurrent map tasks" problem since this issue is not fundamental to HadoopDB. In an additional micro-benchmark, we configure

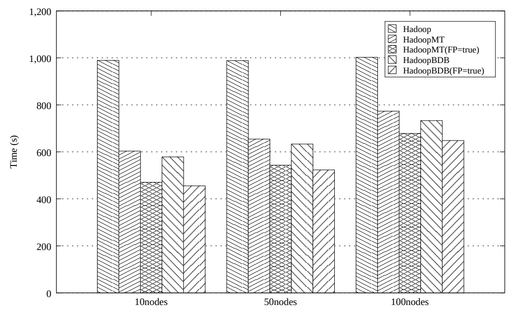

Figure 9: Large Aggregation Results (2.5 million Groups)

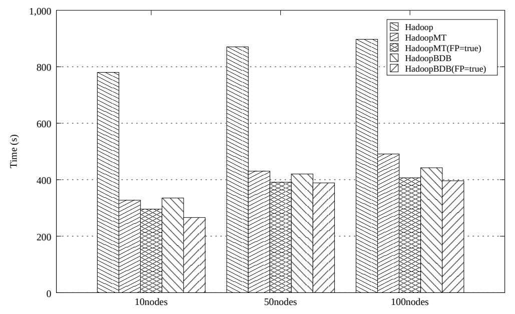

Figure 10: Small Aggregation Results (2,000 groups)

both Hadoop and HadoopDB to launch a single map task to perform the selection task, the performance gap between the two systems is small, only 7%. However, in this setting, HadoopDB is still slower than HadoopMT by a factor of 30%. We believe the cost of JDBC (particularly decoding cost) contributes a lot to the performance gap between HadoopDB and HadoopMT.

5.10 Aggregation Task

In the third analytical task, we use MapReduce to calculate the total `adRevenue` for each unique `sourceIP` in the `UserVisits` dataset. This analytical task consists of two subtasks, large aggregation and small aggregation. The large aggregation task generates 2 million unique groups and the small one generates 2,000 unique groups. The SQL commands of this benchmark are 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)
```

The MapReduce program of this task consists of both a `map()` function and a `reduce()` function. The `map()` function generates an intermediate key/value pair with the `sourceIP` (large Task) or its seven-character prefix (small task) as the key and `adRevenue` as the value. The `reduce()` function sums up `adRevenue`s for each unique key. We also enable a combiner function to perform map-side partial aggregation. The logic of the combiner is identical to the `reduce()` function.

For this task, we compare three implementation schemes: 1) Hadoop, the scheme implemented in [19]; 2) Hadoop with mutable decoding schemes; and 3) Hadoop with mutable

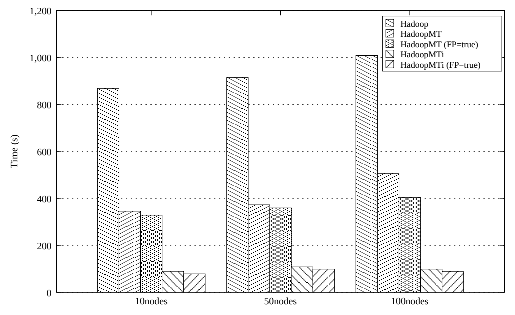

Figure 11: Large Aggregation Results with a Date Range

decoding schemes and fingerprinting based grouping. For 2) and 3), we consider both text record format and Berkeley DB's record format. We set the number of reducers to be the same as the number of slave nodes in the cluster.

Figure 9 and Figure 10 present the results of this benchmark. We found that record parsing contributes a large part of the overall execution cost in Hadoop. The mutable Hadoop implementation (HadoopMT) runs faster than Hadoop by 63%. Different grouping schemes lead to different performance results. In Figure 9, the fingerprinting based grouping implementation (HadoopMT(fp=ture)) runs faster than mutable Hadoop implementation by a factor 20% to 25% in the large task. One interesting observation is that the record format (text or binary) does not affect the performance of MapReduce significantly. This observation holds for the results of all the analytical benchmarks in this paper.

We also design an additional benchmark to study the performance of MapReduce-based aggregation by using indexes. This task calculates the total `adRevenue` for each unique `sourceIP` in a given date range. The date range that we select yields 10% of records in the `UserVisits` dataset, which is roughly 2GB in each node. The SQL command is as follows:

```
SELECT sourceIP, SUM(adRevenue) FROM UserVisits
WHERE visitDate BETWEEN Date('1984-1-1')
          AND Date('1987-1-1')
GROUP BY sourceIP;
```

We only consider text record format in this task. Other settings are similar to those in the large and small tasks. Figure 11 summarizes the results of this benchmark. The results show that index based implementations (HadoopMTi and HadoopMTi(fp=true)) outperform their non-index counterparts (HadoopMT and HadoopMT(fp=true)) by a factor of 3.7 to 4.5. Since only a small number of data are required to be processed, the performance gain from fingerprinting based grouping is not significant.

5.11 Join Task

The join task computes the average `pageRank` of the pages accessed by the `sourceIP`, which generates the most revenue during the given date range. The date range is set to (2000-01-15, 2000-01-22), which yields 134,000 records from the `UserVisits` table in each node. The SQL query for this task is 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') AND
```

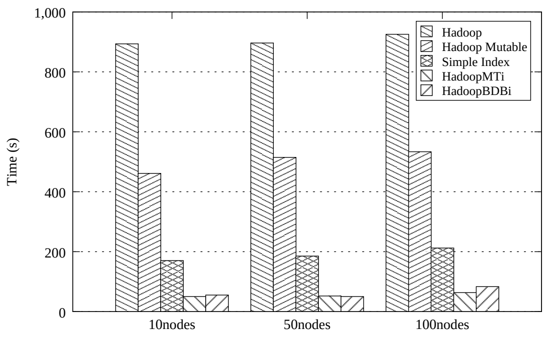

Figure 12: Join Task Results

```
      Date('2000-01-22')
GROUP BY UV.sourceIP;

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

Four implementation schemes are compared for this task.
Hadoop: This scheme is the same one presented in [19].
Hadoop Mutable: The processing logic of this scheme is identical to Hadoop's. The difference is that we use the mutable text decoding scheme to decode `Rankings` and `UserVisits` records. The join task is evaluated by two MapReduce jobs. The first job joins the two tables in the reducer side and outputs a joined record in the key/value format with the `sourceIP` as the key and the record (`pageURL`, `pageRank`, `adRevenue`) as the value. The second MapReduce job computes the total `adRevenue` and the average `pageRank` for each unique `sourceIP` and outputs the record with the largest `totalRevenue`. Only one reducer is launched in the second MapReduce job.
Simple Index: The evaluation process of this scheme is identical to the Hadoop Mutable. The only difference is that we use index to prune redundant data pages of `UserVisits` in the first MapReduce job.
Partition Join: This scheme performs the join task in one MapReduce job. This is achieved by utilizing the partition information. Recall in Section 5.4, the data loader partitions the `Rankings` and `UserVisits` datasets into the same number of partition files based on `pageURL` and `destURL` columns respectively. These partition files follow the same naming rule and are stored in two directories: one for `Rankings` and the other for `UserVisits`. Therefore, for each pair of `Ranking` and `UserVisits` partition files, the records in the two files can be joined only if the two partition files share the same file name.

We launch map tasks for the partition files of `Rankings`. The `map()` function reads qualified data pages from the `UserVisits` partition file with the same file name through the range-index built on `visitDate` column. The `map()` function loads the qualified records of `UserVisits` into an in-memory hash table. Then, it joins records from the two datasets and outputs the joined records. The `reduce()` function is identical to the one adopted in the second job of the Hadoop Mutable. Also, only one reducer is launched. For the Partition Join, we evaluate both text record format (HadoopMTi) and Berkeley DB's record format (HadoopBDBi).

Figure 12 describes the performance of all five implementations. From Figure 12, it can be clearly observed that significant performance improvement is achieved by adopting 1) mutable decoding schemes, 2) indexing, and 3) an efficient grouping algorithm (partition map-side join). Overall, with all three optimizations, the performance of MapReduce (HadoopMTi and HadoopBDBi) improves the original implementation (Hadoop) by a factor of more than 14.

Table 1: Indirect Comparison with Parallel Database Systems

	DBMS-X	Vertica	HadoopOpt
Grep	1.5x	2.6x	1.47x
Aggregation (Large)	1.6x	4.3x	1.54x
Join	36.3x	21.0x	14.68x

5.12 Comparison to Parallel Database Systems

We have no parallel database systems on hand and cannot conduct a direct performance comparison between those systems and Hadoop. Therefore, we only compare the performance of the two types of systems in reference to what have been reported in [20]. In [20], the authors report the performance advantages of two parallel database systems (DBMS-X and Vertica[9]) over Hadoop in three analytical tasks on a 100-node cluster. Due to the similar settings we used, we indirectly compare our results with the two parallel database systems in terms of improvement ratio over the original Hadoop implementation used in [19] and [20].

Table 1 presents the performance gains for reference purposes only. The numbers of DBMS-X and Vertica are extracted from [20]. The number of HadoopOpt is calculated by Hadoop/Hadoop-X, where Hadoop is the performance of original implementation and Hadoop-X is the performance of our best implementation on a 100-node cluster. The indirect comparison results shown in Table 1 reveal that by tuning the factors studied in this paper, the performance of Hadoop/MapReduce is fairly similar to DBMS-X's on Grep and Aggregation tasks, and approaches to Vertica's on the Join task. Another point to note is that both parallel database systems enabled data compression, which improves the performance significantly [20]. However in all our benchmarking experiments, we did not use compression since the current version of Hadoop cannot support compression efficiently. Suppose full-fledged compression is supported in Hadoop, we would expect the performance gap between Hadoop and parallel database systems to be even smaller.

6. RELATED WORK

MapReduce was proposed by Dean and Ghemawat in [13] as a programming model for processing and generating large datasets. The ability of using MapReduce as a data analysis tool for processing relational queries has been demonstrated in [22][21][18][12][15].

In [19] and [20], the authors compared the performance of MapReduce with two parallel database systems. The authors noted that while the process to load data into DBMSs and the tuning of DBMSs incurred much longer time than a MapReduce system, the performance of parallel DBMSs is significantly better. This work is closely related to ours. However, our study focuses on identifying the design factors that affect the performance of a MapReduce system and examining alternative solutions.

In [14], the authors described the differences between MapReduce and parallel database systems. They also presented some techniques to improve the performance of MapReduce, including using a binary record format, indexing, merging the results etc. This work is also closely related to ours. Compared to this work, our study covers more factors such as programming model and scheduling. Furthermore, we also provide benchmarking results to quantify the impact of the factors that we have examined.

In [11], an extensive experiment was performed to study how the job configuration parameters affect the observed performance of Hadoop. This work is nevertheless complementary to ours. Our work focuses on the architectural design issues and possible solutions while [11] focuses on the system parameter tuning. The results from both work (ours and [11]) can indeed be combined to improve the overall performance of Hadoop.

Some issues and techniques presented in this paper have also been studied in the literature. [23] investigated the scheduling algorithm of Hadoop and proposed a LATE scheduling algorithm which improves Hadoop response times by a factor of two. We do not evaluate this scheduling algorithm since the implementation is not available to us. Data partition is also adopted in Hive [21]. However, Hive does not support the partition join technique in current version. The issue of record decoding has also been examined in the Hadoop community recently [5]. The users are advised to avoid using a text record format, and reuse the `Writable` objects for parsing. The suggestion is similar to the immutable decoding scheme that we evaluated. However, we show that the performance of record parsing is not very related to the record format. It is also efficient to decode a text record with the mutable decoding scheme.

7. CONCLUSIONS

In this paper, we have conducted an in-depth performance study of MapReduce in its open source implementation, Hadoop. We have identified five factors that affect the performance of MapReduce and investigated alternative implementation strategies for each factor. We have evaluated the performance of MapReduce with representative combinations of these five factors using a benchmark consisting of seven tasks. The results show that by carefully tuning each factor, the performance of Hadoop can approach that of parallel database systems for certain analytical tasks. We hope that the insights and experimental results presented in this paper would be useful for the future development of Hadoop and other MapReduce-based data processing systems.

8. ACKNOWLEDGEMENTS

The work described in this paper is part of the *epiC* project [6]. It was in part supported by the Singapore Ministry of Education Grant No. R-252-000-394-112 under the project name of Utab. We would also like to thank the AWS (Amazon Web Services) in Education for a research grant that supports our experiments on EC2, Kamil Bajda-Pawlikowski and Daniel Abadi for sharing with us the details of HadooopDB, and VLDB10 reviewers for their insightful comments.

9. REFERENCES

[1] Amazon elastic compute cloud (Amazon EC2) http://aws.amazon.com/ec2/.

[2] http://code.google.com/p/protobuf.

[3] http://developer.yahoo.net/blogs/hadoop/2008/09/.

[4] http://hadoop.apache.org.

[5] http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

[6] http://www.comp.nus.edu.sg/∼epic/.

[7] http://www.cse.yorku.ca/ oz/hash.html.

[8] http://www.oracle.com/database/berkeley-db/je/index.html.

[9] http://www.vertica.com.

[10] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.

[11] S. Babu. Towards automatic optimization of mapreduce programs. In *SoCC*, pages 137–142. ACM, 2010.

[12] 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.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[14] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[15] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *Proc. VLDB Endow.*, 1(1):28–41, 2008.

[16] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498. VLDB Endowment, 2006.

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

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

[19] 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*, pages 165–178. ACM, 2009.

[20] 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.

[21] 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. *PVLDB*, 2(2):1626–1629, 2009.

[22] 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.

[23] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.