MRBench : A Benchmark for Map-Reduce Framework

Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin-gyu Kim, Hyungsoo Jung, Heon Y. Yeom
School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
{kykim, khjeon, hhyuck, sgkim, jhs, yeom}@dcslab.snu.ac.kr

Abstract

MapReduce is Google's programming model for easy development of scalable parallel applications which process huge quantity of data on many clusters. Due to its conveniency and efficiency, MapReduce is used in various applications (e.g., web search services and online analytical processing.) However, there are only few good benchmarks to evaluate MapReduce implementations by realistic testsets.

In this paper, we present MRBench that is a benchmark for evaluating MapReduce systems. MRBench focuses on processing business oriented queries and concurrent data modifications. To this end, we build MRBench to deal with large volumes of relational data and execute highly complex queries. By MRBench, users can evaluate the performance of MapReduce systems while varying environmental parameters such as data size and the number of (Map/Reduce) tasks. Our extensive experimental results show that MRBench is a useful tool to benchmark the capability of answering critical business questions.

1 Introduction

As the quantity of data to process is increasing in many areas such as web search service and scientific research, it is a challenging problem to process large amounts of data efficiently. Existing DBMSs are too generalized for some specific applications such as web search engine[6]. To reduce unnecessary overhead of DBMS for some specific applications, we have to find another approach to deal with large amounts of data. Recent work[9] shows that it is better to deal large amounts of data with lots of clusters than with some multiprocessor server with shared memory in terms of performance and cost. However, it is very difficult to process data with many clusters due to complexity of data distribution, parallel processing, results collection with fault tolerance. There are some solutions such as MPI and Grid for parallel computing, but it is not easy to solve problems with both efficiency and convenience.

MapReduce[7] is Google's programming model for scalable parallel data processing, and various implementations[7, 11, 1] of MapReduce are used to solve problems in many areas (i.e., web search service.) MapReduce uses two functions - Map and Reduce - from the parallelizable functions. Users implement only Map function and Reduce function to convert their programs to parallelized versions easily. There are many implementations of MapReduce such as Map-Reduce-Merge[6], MapReduce for Multi-core and Multiprocessor systems[11], Google MapReduce, and Apache Hadoop[1].

Even if MapReduce implementations are already being used widely, there are only a few ways to measure performance of MapReduce implementations. There are some researches[7, 11, 1] which tried to benchmark MapReduce with many testcases such as word-count, sort, grep, and matrix multiply. However, they do not have enough workloads to benchmark MapReduce. Since MapReduce's improvement includes relational algebra[6, 10, 2], we believe that benchmarking MapReduce with relational algrebra will be necessary in industry and academia.

To this end, we adopt TPC-H[3] that is an audited benchmark for decision support system since TPC-H evaluates database system with business oriented database and 22 ad-hoc queries and each query represents realistic complex queries. In this paper, we represent MRBench, which is a kind of MapReduce benchmark based on TPC-H. We converted each queries into MapReduce jobs. Each jobs consists of several steps of MapReduce. To implement SQL queries with MapReduce, we implemented basic SQL features (SELECT, FROM, WHERE, and JOIN) into MapReduce tasks. Each step's result data is treated as input data of next step of MapReduce except the last step of MapReduce. The last step of MapReduce collects result data into one file to represent overall result of a query.

The main contributions of this paper follow:

- **A benchmark design for general MapReduce system.** We designed all queries of TPC-H into MapReduce model by default MapReduce model[7].
- **A benchmark implementation for MapReduce system.** We implemented MRBench with Java targeting Hadoop system, but source code can be easily converted into other MapReduce systems.
- **A flexible testing environment with various configurations.** Options that can be configured are the size of database, the number of Map tasks, and the number of Reduce tasks.

The rest of this paper is organized as follows. Section 2 describes related works of this paper. Section 3 gives an overview of MRBench. In Section 4, we will discuss detailed implementation issues of MRBench. Section 5 describes MRBench experiments with various environments. Section 6 discusses summary and future work.

2 Background and Related Work

2.1 MapReduce

MapReduce[7] is a programming model from Google for simplified data processing on large clusters. It has pretty simple structure by Map and Reduce. The overall process of MapReduce works as follows. When MapReduce function is called, MapReduce program first splits user's input data into M pieces and starts with many copies of program with one piece each. One of the copy becomes master program that manages whole execution. M map tasks and R reduce tasks are being assigned. Map task reads one piece of input data and generates key/value pairs from data. Reduce tasks read values with same key, process values and create result data. The master program returns results from reduce tasks. When one of the worker fails, master re-execute failed tasks on another worker node.

Phoenix[11] is an implementation of MapReduce for shared-memory system. The authors of Phoenix evaluated MapReduce on Multi-core and Multiprocessor systems with Phoenix. Hadoop[1] is Apache's implementation to process vast amounts of data with support of MapReduce and Hadoop Distributed File System. Hadoop can be deployed easily by configuring some variables - some paths and nodes. Hadoop defines one master node that manages all systems and jobs, and other worker nodes. Prototyped implementation of MRBench is based on Hadoop since it is a kind of open source software. HBase[2] is Hadoop's extension for database system with Hadoop. HBase is modeled from BigTable[5]. It provides Hql to support SQL queries, but Hql cannot support complex queries which can be found often in TPC-H queries yet.

Map-Reduce-Merge[6] is a new model from MapReduce model. Map-Reduce-Merge adds Merge phase to support relational heterogeneous datasets. It is showed that Map-Reduce-Merge can process relational database operators such as join, selection, and Cartesian product. This work motivates our research.

2.2 Benchmark

Benchmark is running a number of programs/operations to obtain the performance of an object. In [12], application-specific benchmark can be categorized into three categories - vector-based, trace-based, and hybrid methodology. Vector-based methodology characterizes an underlying system by a set of testcases that consist of some system vectors. Each vector's characteristic can be found by some of testcases and results of other vectors. Trace-based methodology characterizes the trace rather than characterizing an underlying system, and uses the characterization to generate benchmark that is similar to actual usage. Hybrid methodology combines two methodologies - vector-based and trace-based. We focus on trace-based benchmark.

TPC-H[3] is a benchmark in TPC for decision support system. It consists of industry related data with 8 tables and 22 business-oriented queries. TPC-H benchmarks decision support system with massive data that executes query with high degree of complexity, and generates complex results. The performance of TPC-H result is measured by QphH@Size(Query-per-hour). Performance can be evaluated on same size of datasets. Many clusters are compared with performance in [3] by QphH@Size and Price/QphH.

STMBench7[8] is a candidate benchmark to evaluate software transactional memory (STM) implementations. STMBench7 is a good benchmark example which is developed from other existing benchmark. STMBench7 got its base from OO7 benchmark[4] which benchmarks object-oriented database systems. STMBench7 converted OO7 to fit STM implementations, and added some characteristics for better STM benchmarking. STMBench7 is well designed and implemented to benchmark various STM implementations.

3 Overview

In this section, we will give an overview of TPC-H benchmark structure and MRBench design.

3.1 TPC-H overview

TPC-H consists of 8 tables for dataset design and 22 queries for process workflow. Each table attribute represents data for industrial resource management, and each query represents business-oriented ad-hoc query that is complex and possibly produces a lot of data as results. We will see more specific structure below.

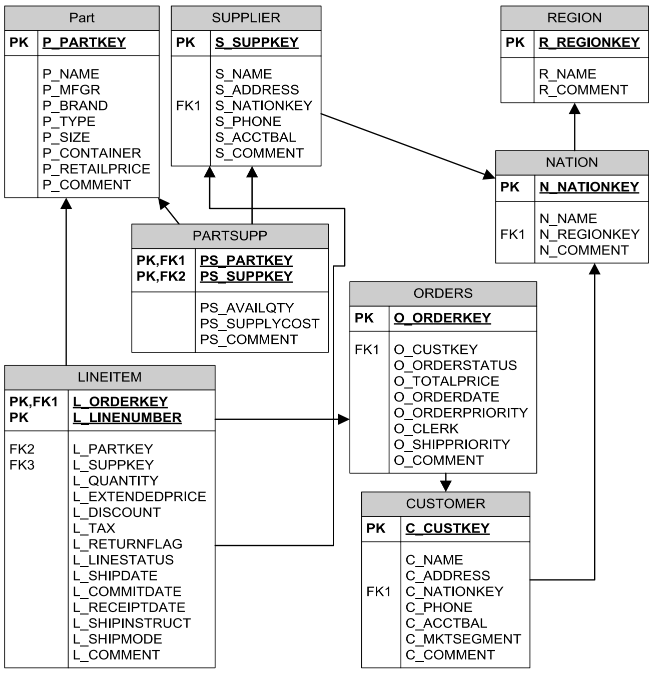

Figure 1. TPC-H's table relationship

Data relationship overview. Fig 1. represents table design of TPC-H including relations with foreign-key relationship. Each table has at least one relationship with other tables. Each table has 3-16 attributes which consists of some primary/foreign keys and other specific data. There are some constraints on these attributes, such as positive key value and date format restrictions. Generated data shows patterns similar to real-life data. For example, there are many "dead data" customers (who has no orders so will be dropped in table join) in customer table and table sizes are well-balanced with certain ratio except that nation table has 25 fixed nations and region table has 5 fixed regions.

Query overview. TPC-H has 22 queries with following characteristics.

- A high degree of complexity
- Various access to database
- Test of a large part of the available data
- Ad-hoc manner
- Varying query parameters

Queries in TPC-H consist of SQL operations such as select, join, group by, and order by. In addition, the queries use views as well as tables. Most queries combine and filter data from many tables.

Fig 2. is an example of query with table joins.

3.2 MRBench design

How to convert each SQL features to MapReduce framework's primitives, Map and Reduce, will be described in this section. In the end, we will present an example of implementing TPC-H's query 9 with Map and Reduce.

SELECT. Select operation in SQL is equivalent to projection, a relational algebra operation. Projection selects required attributes from tuples. In MapReduce framework, one can perform projection in the Map phase by emitting required attribute values as intermediate data. Select implementation can be described briefly as follows:

- **Map**: for each tuple, produce (key, selected attribute values)
- **Reduce**: for each unique key-value pair, emit (key, value)

Join. A Join operation combines data from two or more tables by certain conditions. In a TPC-H benchmark environment, tables are associated to each other by foreign keys. Join is usually performed on the foreign key relationship. Join can be implemented on MapReduce framework by using attributes on the join condition as map phase keys. Join procedure is outlined as follows:

- **Map**: for each tuple, produce (join condition attribute, other attributes)
- **Reduce**: for each unique key value, concatenate attributes from join tables.

Outer join can be performed in a similar manner.

GROUP BY. A GROUP BY statement in SQL is used to specify that a SELECT operation returns a list that is grouped by one or more attributes. Usually, some sort of aggregate function comes with GROUP BY statement. Brief Map/Reduce procedure of GROUP BY operation is as follows:

- **Map**: for each tuple, produce (GROUP BY attributes, other attributes)
- **Reduce**: for each key, compute the aggregate function on data collected by the group-by attributes.

If there are more than one GROUP BY attributes, the values of attributes are concatenated and act like a single key value.

VIEW. View creates new virtual table from existing tables. A view is defined by a query expression, which consists of some relational operations: join, projection, aggregation, etc. In MRBench, a view is just another query, which executes before the main query starts. There is just one query (query 15) in TPC-H which employs a view.

```
select
        nation,
        o_year,
        sum(amount) as sum_profit
from
(
        select
                n_name as nation,
                extract(year from o_orderdate) as o_year,
                l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
        from
                part,
                supplier,
                lineitem,
                partsupp,
                orders,
                nation
        where
                s_suppkey = l_suppkey
                and ps_suppkey = l_suppkey
                and ps_partkey = l_partkey
                and p_partkey = l_partkey
                and o_orderkey = l_orderkey
                and s_nationkey = n_nationkey
                and p_name like '%:1%'
) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;
```

Figure 2. Query 9 of TPC-H

Example : Query 9. In this section we will present an example of MRBench design converted from a TPC-H query. Fig 2. represents original query 9 of TPC-H. Query 9 joins six tables and processes data by grouping results of embedded query. To join six tables, we perform five join operations in three steps of MapReduce. Each join operation is performed with a key which is the primary key to one table. After projection and filter processes in Map phase, Reduce phase processes join tables. These joins can be performed in parallel by treating in same Map/Reduce tasks.

Overall process is as follows. In the first step, six tables are joined into three tables - part / partsupp, supplier / nation, and lineitem / orders. Map phase loads six tables, performs projection, and creates key/value pairs. Key has table type and join key of table, and Value has projected attributes which is needed by query. Reduce phase joins values with same key.

Fig 3. explains our design of first step's join for part table and partsupp table. Map phase performs filtering and projection for two tables, and creates key/value pairs, which key is partkey and value is each table's projected attributes. Reduce phase performs join with key. To perform join, reduce phase generates Cartesian product between two tables with non-key attributes. Generated results are part/partsupp table's tuples. Results of first step will be three joined tables - part / partsupp, supplier / nation, and lineitem / orders.

Second step joins p/ps table and s/n table, and no process for l/o table as same methods with first step. Third step joins p/ps/s/n table and l/o table. Fourth step groups joined table's tuples with n_name and year of o_orderdate as o_year and orders by n_name in ascending order, and o_year in descending order. Grouping and ordering are performed using Map keys. By using map keys as n_name and o_year, grouping can be performed, and ordering can be performed with defining comparator of keys.

Fig 4. shows overall process of MRBench's Query 9 design.

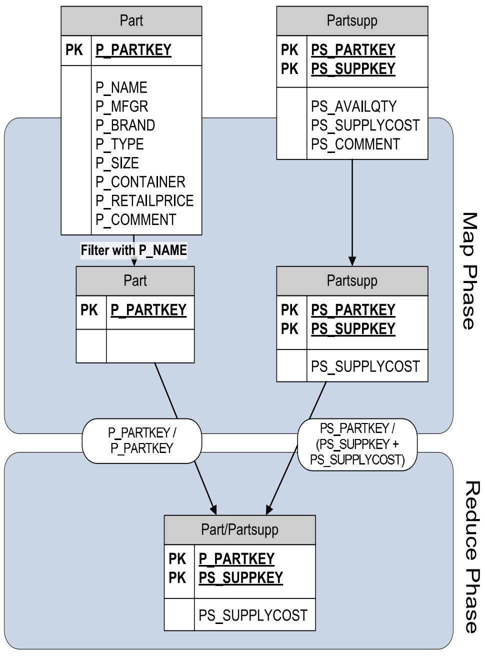

Figure 3. MRBench Join of Part and Partsupp in the first step of Query 9

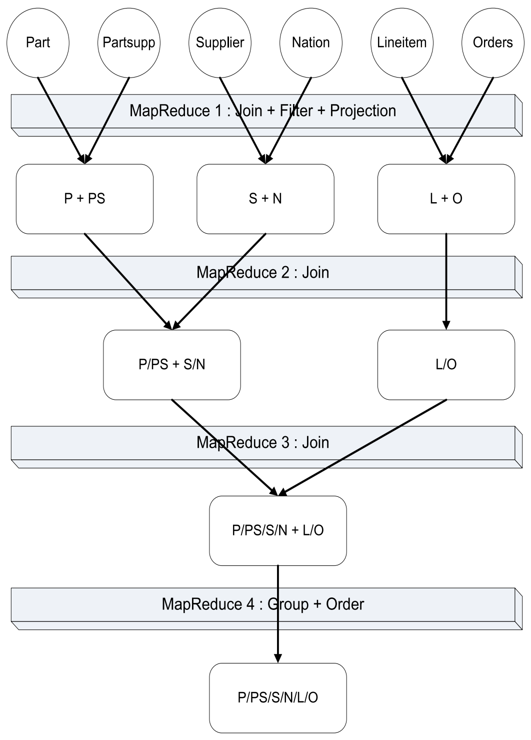

Figure 4. MRBench Design of Query 9

4 Implementation issues

In this section, we will discuss some issues arised during the implementation. We wrote 8,000 lines of Java code to implement MRBench. Our implementation is based on Hadoop 0.14.4 with Hadoop Distributed File System (HDFS). Every test database is stored in HDFS. All intermediate data and results are also stored on HDFS.

4.1 Implementing MRBench design

To implement MapReduce programs of our design, we need to consider some issues.

Processing tables. To use the test database defined by TPC-H, we directly parse test database files generated by TPC-H data generator which are plain text files where each attribute is separated by '|' and each line corresponds to a tuple. Each file is located in HDFS, and copied to input path when the query requires the table. Copy time will not be included in the measured running time.

Each table is coupled with an appropriate Java class that parses each line of table files and converts the line into the list of attributes. Table parser converts each attributes to its own type. Character and text type is parsed into String type, integer type is parsed into long type. Decimal type is parsed into float type, and date type is also parsed into long type that the value is the number of milliseconds elapsed since midnight UTC of January 1, 1970. Table parser can process a project operation by ignoring not required attribute values.

Join. Joining two tables is implemented as we mentioned in the previous section. We collect all non-join-key table tuples and a join-key table tuple. In TPC-H's query, every join is processed with keys which is one table's primary keys, so we just collected one tuple from join-key table, but it can be implemented to collect both tables' tuples without many modifications. Non-join-key table's tuples are collected until a join-key table tuple is received. After that, the rest of the tuples can be directly handled into output key/value pairs.

Order. Ordering result is easily done in Hadoop. Hadoop sorts the reduce results by the value of map key. By defining map key as attributes to be ordered, and defining map key type's comparator into appropriate comparator, Hadoop gives sorted results for each reduce tasks. By running only one reduce task at the last step, the overall result can be checked in one sorted output file. A simple combiner function also can be implemented to merge some ordered results into one output file because separated results are locally sorted in each file.

4.2 Miscellaneous issues

Followings are issues in implementing MRBench, but not from design issues.

Various configurations. We support three configuration options in MRBench implementation - database size, the number of map tasks and the number of reduce tasks. By managing different database in different directory, database size can be selected while copying table files to input path. Number of map tasks and reduce tasks can be configured by Hadoop's JobConf class. However, the number of map/reduce tasks is only a hint to MapReduce task and the actual number of tasks depends on file splitter's choice.

Intermediate data. When a query is handled with multiple MapReduce phases, intermediate files generated by a phase should be handled properly. MRBench implementation uses output of a phase into input of next phase directly by configuring former phase's output path as next level's input path. Intermediate files are deleted after running all phases, and deletion time will not be included in running time.

4.3 Validation

To validate our implementation, we compared our result with TPC-H benchmark results from Postgresql database. Actually, validation is less important in our paper because our implementation's major target is TPC-H's workload, not TPC-H's results. But, to confirm the correctness of our implementation's workload, we checked every results of queries by TPC-H's results from same query variables and same tables. Some bugs were found - minor bugs such as date calculation bugs and major bugs such as logic bugs - and fixed.

5 Experiments

In this section, we ran some experiments with MR-Bench implementation under various configurations. We tested our implementation by varying the size of database, and the number of map tasks. We also validated that our implementation worked properly as MapReduce benchmark, and characterized MRBench to target the system to use MRBench.

5.1 Experimental Setup

To test MRBench implementations, we used Seoul National University's Supercomputer. The supercomputer is comprised of 480 computing nodes. Each computing node has two PowerPC 970 2.2GHz CPUs, 2GB Memory, and connected with each other by Ethernet and Myrinet. We used only 1Gbps Ethernet for the network communication. Every node uses Suse Linux Enterprise Server 9. Jobs are controlled by LoadLeveler 3.2.1. Home directory is shared by NFS, and each node has own local disk.

We used 16 nodes in our experiments, except for experiments with varying number of nodes. One of these nodes is used as master node, and all of the 16 nodes worked as worker nodes. Hadoop 0.14.4 and Java 1.5.0 is used for all cluster nodes. Hadoop Distributed File System (HDFS) is used to share pre-generated TPC-H data, intermediate data, and result data. Two data sets - 1GB and 3GB - are used to test our implementation. 3GB data set is used by default. In every map task, each splitted file is encouraged to have 64MB size. In each test, every query is executed and used as result.

We analyzed runtime results as three components - System management time, data management time, and processing time. System management time is the time spent on adjusting and maintaining overall system. Data management time is the time spent on managing intermediate data, such as input file split time, file transfer time, and file writing time. Processing time is the time spent on processing data on map phase or reduce phase. We characterized MRBench's runtime with these components.

5.2 Various Datasets

In this experiment, we compared our results for the different sizes of database: 1GB and 3GB. Fig 5. shows results of query runtime for 1GB and 3GB database. As the numbers of map and reduce tasks, the default values are used. We surmise that data management time and processing time are in proportion to the size of dataset, however, system management time is not.

As a result, every query showed such patterns that the runtime with 3GB dataset is almost three times longer than that of 1GB dataset. As this result shows, MRBench scales well with the sizes of test databases.

5.3 Various Number of Nodes

In this experiment, we compared our results for the different number of computing nodes: 8 nodes and 16 nodes. Fig 6. shows results of query runtime for 8 and 16 nodes. We set the numbers of map and reduce tasks as the Hadoop's default values. We also can verify that MRBench properly shows the effect of processing power.

It showed a pattern different from previous experiments - the speedup is not linear. We surmise that only the processing time is reduced by the increased number of nodes. Data management time and system management time might be increased due to the doubled number of computing nodes. We conclude that the portion of processing time accounts for the difference of runtimes for the different numbers of nodes.

The above two experiments show that MRBench fits test systems for jobs requiring appropriate processing and massive data. As section 5.2 shows, MRBench directly shows the effect of chainging the amount of data. This shows that MRBench scales well with the data size. As this section shows, MRBench is also sensitive to the number of computing nodes, however, not scales linearly. This shows that MRBench also reflects the processing power required by the jobs, but it is only secondary to be concerned.

Thus, we conclude that MRBench can benchmark systems for jobs processing huge data and requiring relatively small processing power. We can select some queries for our needs. For example, to test systems for jobs requiring less processing powers, we can choose queries that are less sensitive to the change of processing powers such as the query 1, 4, and 17.

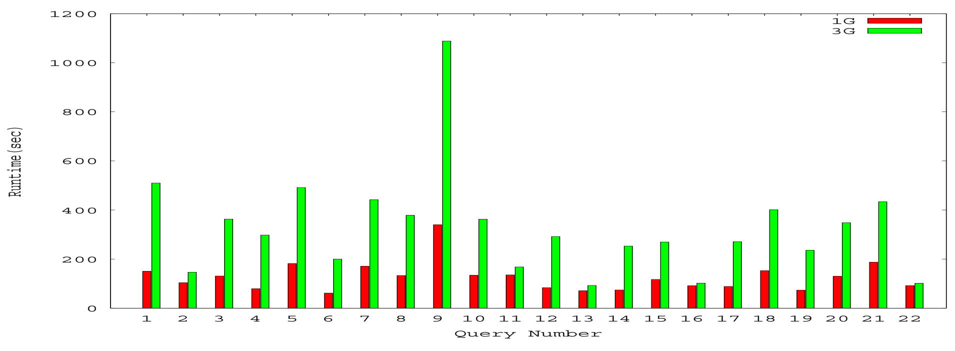

Figure 5. Experiments with various data size

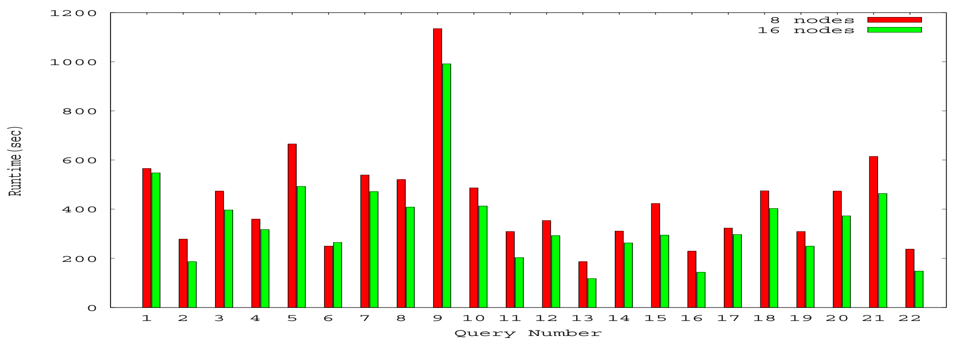

Figure 6. Experiments with various number of nodes

5.4 Various Number of Map Tasks

In this experiment, we evaluated the effects of changing the number of map tasks - 40, 60, 80, and 100 - for 3GB test database. As mentioned above, our configuration on the number of Map/Reduce tasks is only hint for MapReduce system. The file splitter decides the actual number of Map/Reduce tasks. Thus, detailed control is not possible. Actual number of map tasks under configuration of 40 map tasks was 40 to 60 tasks, 100 to 130 tasks for 100 map tasks.

Fig 7. shows the result of our experiment with various number of map tasks. Each query showed different patterns with various numbers of map tasks. This experiment showed that the optimal number of map tasks increases as the data to process grows. Some experiment showed two optimal number of map tasks. We guess that the result is due to the complex effect of steps with massive data input and steps with small data input. 60 - 80 map tasks might be the best option for 3GB database with data intensive queries. The optimal number of map tasks seems to be related to the size of data and HDFS block size, but more experiments are needed to confirm this. We leave the investigation on this relationship as future work.

6 Conclusion

In this paper, we designed MRBench: a possible benchmark for evaluating MapReduce system. We used TPC-H's database and queries to benchmark complex and heavy workloads. TPC-H's queries are converted into MapReduce tasks based on basic conversions with projection, filtering, join, and so on. We also implemented our design under Hadoop system with some configuration options. Every 22 queries are designed and implemented. We tested our implementation with four hadoop nodes using some experimental variables.

MRBench is only a possible approach to benchmark MapReduce system and many experiments and updates are needed. Our expected future works on MRBench are as follows :

- Query selection is one of the key to use MRBench better. As our experimental results showed, some queries used less data and are inadequate to be used for benchmarking. To remove garbage in benchmark results, query selection is one important key.

- Deciding appropriate number of Map/Reduce

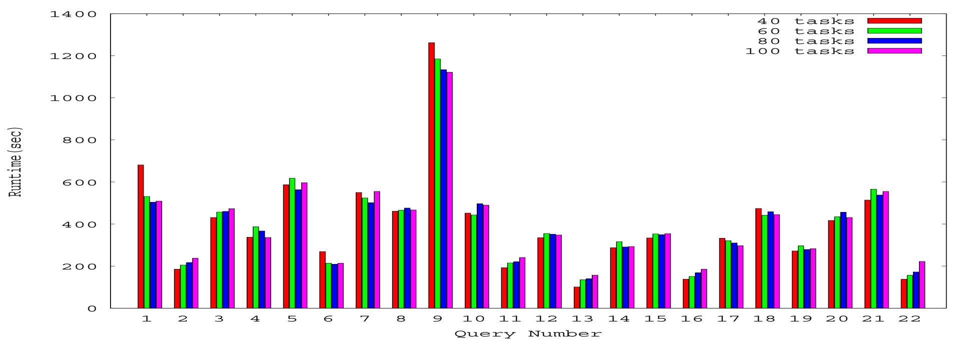

Figure 7. Experiments with various number of map tasks

tasks can be treated under MRBench. There is no general guideline to select the number of Map/Reduce tasks. By using MRBench and varying Hadoop configurations such as HDFS block size and the number of Map/Reduce tasks, a MapReduce system can be tuned under specific size of data.

- Comparing with other testsets can be performed. How MapReduce shows different performance characteristics under different complexity of workloads can be good experiments for proper MapReduce benchmarking.

Acknowledgment

This study was supported by the Seoul R&BD Program (10561), Seoul, Korea. The ICT at Seoul National University provides research facilities for this study.

References

[1] Apache hadoop.
[2] Apache hadoop hbase.
[3] Tpc-h.
[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 12–21, New York, NY, USA, 1993. ACM.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.
[6] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.
[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
[8] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.
[9] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 26-30 March 2007.
[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin : A not-so-foreign language for data processing. In *SIGMOD*, June 2008.
[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.
[12] M. I. Seltzer, D. Krinsky, K. A. Smith, and X. Zhang. The case for application-specific benchmarking. In *Workshop on Hot Topics in Operating Systems*, pages 102–, 1999.