A Comparison of Distributed Data Processing Systems

Vincent M Chen
Information and Computer Science Department
University of Oregon
applekey@cs.uoregon.edu

Abstract—

This survey paper is intended to compare traditional distributed DBMS systems with newer MapReduce frameworks. The paper will first give a brief description of the general similarities of both frameworks. The paper will then delve into the areas that MapReduce fails in comparison to distributed DBMS systems. The paper will finally cover the additional research that has being released in order to address these short comings.

I. Introduction

The explosion of data in the past decade has lead to many questions in finding the most efficient method in order to analyze the data being generated. Best put by Google's CEO, Eric Schmidt, "There were 5 exabytes of information created by the entire world between the dawn of civilization and 2003. Now that same amount is created every two days.", more precisely a survey conducted in 2003 found that 6.8 exabytes were being created every 2 days[1]. With such a large volume of throughput, trivial tasks such as locating a users' account or finding all users within the same group can't be found in meaningful time with a single computers nor using sequential algorithms.

In order to solve this problem, parallel and distributed computation frameworks leverage the power of multiple processors in order to resolve queries that work on data is independent from each other. For example, different users on an on-line shopping platform where every shopper has their own independent profile, shopping cart and payment information. Independence within the data means that parallel computation can be used, where instead of a single computer processing data sequentially, the job is divided into many independent tasks and processed in parallel. This allows the computation of a large data set to be divided among the processing power of multiple computers.

Parallel computation systems have existed since the 1980's, the most common of which are database systems. In the time since, many database systems have evolved into both parallel and distributed varieties. More recently, newer frameworks such as MapReduce or Apache Spark have adopted and re-invented the parallel and distributed computation model of past distributed and parallel database systems. The goal of this survey paper is to firstly introduce these different data query systems and their variants. Secondly, discuss the common approaches all of them use to speed up computation, and finally highlight unique aspects of each system.

II. Basic Description of Functionality

A data processing framework should provide the following fundamental features:

1. Guarantee high degree of fault tolerance.

2. Compute a user defined query in a reasonable time.

A. Fault Tolerance

Fault tolerance guarantees the reliability of data being processed. Without this guarantee, the data being returned from any query cannot be used for any meaningful analysis. Fault tolerance covers errors from a few different sources.

1. Network fault - communication between nodes is unreliable and data is lost in between.

2. System fault - OS failure.

3. Hardware fault - memory, hard disk, component heat death.

4. User fault - user query has code that leaves query engine in a bad state.

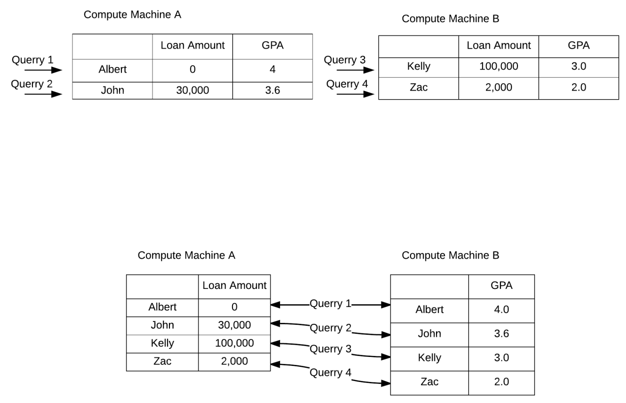
In order to speed up a user's query running on a sequential database system, there are two categories of parallelization that a database system can leverage.

B. Parallel Computation

The framework can process independent queries in parallel. For example, searching for all students who owe more than 10,000 dollars in student loans can be queried in parallel, each student's data can be treated independently.

C. Distributed Computation

The framework is able to distribute the data set across multiple compute machines. Each machine might contain independent data, for example, one machine contains all students with first names starting A to J and another from K to Z. Another case is multiple machines hold data that is related to one another. For example in figure 1, machine A might hold all student's GPA and Awards and another machine holds the student's student loan amount. If the query were to find all students that had more than 10,000 in student loans but also a 4.0 GPA in order to give them a financial award, then each query per student would need to look at the data in both compute machines.



Compute Machine A

	Loan Amount	GPA
Albert	0	4
John	30,000	3.6

Compute Machine B

	Loan Amount	GPA
Kelly	100,000	3.0
Zac	2,000	2.0

Compute Machine A

	Loan Amount
Albert	0
John	30,000
Kelly	100,000
Zac	2,000

Compute Machine B

	GPA
Albert	4.0
John	3.6
Kelly	3.0
Zac	2.0

Fig. 1. Distributed data dependency

III. Databases Management Systems

Databases management systems are frameworks that are designed to store data in tabular tables with a set schema. DBMS systems use relational queries to perform CRUD (Create, read, update and delete)operations on the stored data.

A. Parallel DBMS

Relational queries are ideal suited to run parallel since each operation declares independent operations on independent data sets [2]. By partitioning the data among multiple processors, a single query can be carried out in parallel and the result merged into a final step. In figure 2, an independent scan (ex. look for key word) is applied and the results are sorted by binning by first letter. The final result of each independent operation is merged into the final result.

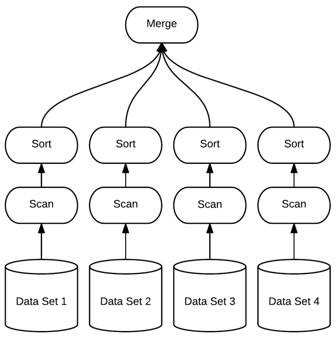


Fig. 2. Independent query operations followed by a merge

B. Distributed DBMS

Distributed DBMS systems are an additional layer of abstraction on top of Parallel DBMS. Where in parallel DBMS systems, all queries are run on the same machine sharing the same memory space. Distributed DBMS systems access memory through a shared memory controller. This memory controller is often not local and instead accessed through a router. Distributed DBMS systems need to consider the additional factor of latency when access data that does not reside in the current compute machine. A large factor in determining the performance of a distributed DBMS system is how efficient the memory layout is such that network traffic between different nodes is minimized.

Generally speaking, there are three main types of memory architectures for distributed DBMS systems, Shared-Nothing, Shared-Memory, Shared-Disk.

1. Shared-Memory: All compute machines share access to a global memory as well as global disk storage.

2. Shared-Disk: Each compute machine has their own memory, but share a global disk.

3. Shared-Nothing: Each compute machine owns their own memory and disk.

Share-Nothing architectures have being adopted with wide popularity due to its characteristics of minimizing resource sharing. In addition, the lack of a Shared-Memory or hard disk means that commodity interconnection networking switches can be used. This characteristic of minimal dependence between nodes and the lack of an interconnect fabric enable Shared-Nothing architectures to have many backup and duplication layers allowing for better reliability. In summary, Shared-Nothing architectures enable scaling to happen at a massive scale. For example, in the 1990's the largest Shared-Nothing system was Intel's 2000 node Hypercube, while many competing Shared-Memory multiprocessors were limited to only about 32 processors.

Shared-Memory and Shared-Disk systems do not scale as well due to excessive inter-node network traffic. During a computation, the shared network switch quickly becomes a bottleneck, any advantage gained from accessing shared memory quickly dissolves as the network struggles to keep up with all compute node's requests [3]. There have being many attempts to implement affinity scheduling much like processor affinity to a particular processor, but these attempts have not yielded any favorable results.

C. Data Partitioning

Data Partitioning is a Shared-Nothing splitting strategy that splits a large dataset across multiple disks. This strategy is used to utilize the I/O bandwidth of multiple disks' read and written in parallel for data independent queries. There are three common methods, the goal of all three is to minimize the amount of un-necessary communication between disks by placing data needed by independent compute nodes locally within their assigned disk. Figure 3 gives a visual representation of each.

1. **Round Robin**
Round Robin is ideal if all compute nodes are accessing data sequentially.

2. **Hash Partitioning**
Hash Partitioning is ideal if the compute nodes access data sequentially or associatively. Data is assigned to each disk by an associate hash transform such that each compute node does not need to look to other data partitions.

3. **Range Partitioning** Range Partitioning is ideal for sequential and associate data access. One major issue with this partition scheme is if the data is not fairly partitioned, data skew can arrise where all data is in one partition.

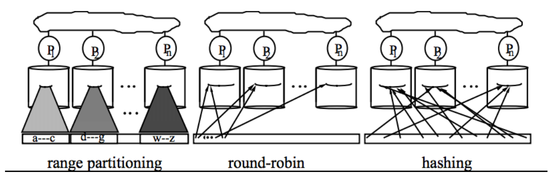


Fig. 3. Three different types of data partitioning used by Shared-Nothing DBMS [2]

D. Fault Tolerance

DBMS systems have very mature and well-studied failure mechanisms. For the purpose of this survey paper, the exact details of each will not be discussed. DBMS fault tolerance follow the general fault tolerance categories that were described in the Basic Description of Functionality section. DBMS systems take advantage of the schema and transactions to implement restart at different granularity levels. By using these different transaction levels, DBMS systems are able to take advantage of the work that has already being completed and restart at the most optimal point before the crash.

E. Optimizations

DBMS systems use table schemas and indexing in order to allow the relational SQL query language to both logically analyze(static optimization) as well as dynamically adjust query execution based on the layout and size of the stored data per table. Take for example, the query III-E, given the tables:

1. Sailors (sid, sname, rating, age)
2. Boats (bid, bname, color)
3. Reserves (sid, bid, day, rname)

```
SELECT S.sid, S.sname, S.age
FROM   Sailors S, Boats B, Reserves R
WHERE  B.bid = R.rid AND B.bid = R.bid AND
B.color = "Red" AND S.age < 30;
```

This query can be compiled into four different execution plans, these are show in figure 4.

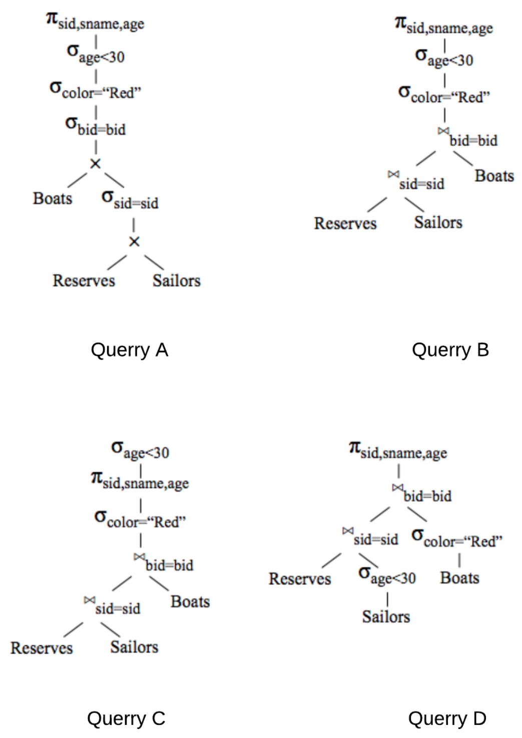


Fig. 4. Same query represented by four different execution plans [4]

The DBMS engine can then use a combination of information on the size of each leaf, which is provided through table schema, with dynamic programming in order to find the best execution plan.

IV. MapReduce

MapReduce is a general distributed computation programming model introduced by Google in 2004 [5]. MapReduce is different from distributed DBMS systems in that it is a much more general. MapReduce is not limited by the query language of a rigid schema as is the case with DBMS systems and does not need to operate on data that follows a specific table schema. MapReduce, like DBMS systems have fault tolerance mechanisms but they operate differently than DBMS systems due to the architectural differences.

A. *Mapper and Reducer*

The execution of any MapReduce program includes two steps, the map step and then the reduce step. In between the map and the reduce step, there is a barrier in that the reduce step will not start until all mappers have completed. In the map stage, the data is partitioned equally among N mappers. Input data arrives to the mapper as a list of (key, value) pairs, the mapper applies a user-defined map operation to these (key, value) pairs. The map operation produces an intermediate (key, value) pairs which are the results of the map operations, these results are written and saved to the file system.

The reducer is again a user defined operation that processes the intermediate values from the mapper grouped by key. The reducer can produce zero or an aggregate of all values that map to the same key.

Figure 5 gives an execution diagram of the MapReduce framework.

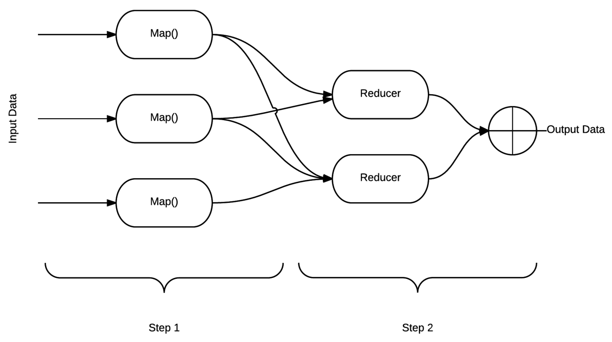


Fig. 5. MapReduce Program Execution

B. *Hadoop*

There are two very popular frameworks that implement the MapReduce programming model, these are the closed source Google MapReduce and open source version Hadoop MapReduce. This paper will focus on Hadoop instead of Google MapReduce since the complete details of Google MapReduce and relatively unknown, however both frameworks function very similarly.

Hadoop consist of three layers:

1. Application layer

2. MapReduce layer

3. HDFS layer

The HDFS layer is Hadoop's storage layer. The HDFS structures data in blocks that is managed by a master node, when a MapReduce program executes, the master node will assign each individual mapper data chunks as broken down by the HDFS. More importantly, the HDFS layer is one major component Hadoop MapReduce's mechanism for fault tolerance. The application layer sits on top of the MapReduce layer. The application layer consists of several additions to the standard MapReduce framework, these include high-level languages, database adaptations and data flow optimizations. These application layer additions will be discussed later on in this survey paper.

C. *Parallel and Distributed Computation*

The main advantage of the MapReduce programming model is that there are no dependencies between the mappers or reducers. MapReduce therefore is extremely scalable, adding additional nodes either through adding additional cores or compute nodes is a relatively simple task. This design is very similar to that of the Shared-Nothing architecture found in distributed DBMS systems.

D. *Fault Tolerance*

Hadoop's fault tolerance mechanism is extremely basic, it achieves fault tolerance through replication. When a MapReduce job is invoked, the input data is duplicated three times on the HDFS. This duplication process occurs for intermediate and final results as well. When the master node, who is responsible for keeping track of map and reduce jobs notices that a node has failed, the master node will reschedule the map and reduce job with the input or intermediate data. Hadoop MapReduce's failures can be categorized into three categories.

1. **Task Failure**
Task failures occur either on the mapper or reducer. These failures can occur from (1) malformed input data, (2) contention of shared resources such as file servers or network switches. When a task failure occurs, workers mark the running task as a failure and notify the master scheduler. The master scheduler will restart the same task on the same node twice before either skipping the bad data section, or restarting the whole computation.

2. **Worker Failure**
Worker failures happen when the worker stops responding to the master node. Worker failures can occur from (1) hardware failures such as a broken hard disk or (2) network switch failures such that the worker node cannot be contacted.

3. **Master Failure**
Hadoop's master node is usually duplicated by multiple instances. Since this is a single point of failure and easily de-

tectable, when the master node fails, the responsibility is simply given to any secondary duplicated instance.

V. DBMS vs MapReduce

MapReduce and distributed DBMS systems are very similar in that both are distributed data processing frameworks, both support roughly the same feature sets, performance, fault tolerance, and user-customizability. However MapReduce is far more flexible, where as DBMS require data to be organized into schema, MapReduce can be applied to any variety of data set. This flexibility is not without its trade-offs, in general it is found that MapReduce is about 15 - 20 percent slower than DBMS in a variety of general tasks [6]. This section will cover the differences between the two and present a performance study between the two.

A. MapReduce Advantages

1. **Simplicity in Framework**
Programming MapReduce is easier to use than query languages such as SQL for two reasons. Firstly, there are no complex language constructs as is the case with SQL, and secondly, the programmer does not need to understand a table's schema and data layout.

2. **Flexibility**
The lack of schema's allow data to be organized in any fashion. The programmer does not need to massage the input data into a rigid format.

3. **Storage Agnostic**
Since there is no set schema, data can easily be moved from one storage framework or representation to another with very little modification. For DBMS systems, there would have to be a translation job between two different storage frameworks to ensure that the integrity of the data is maintained.

B. MapReduce Disadvantages

1. **Lack of High Level Language**
While MapReduce is extremely easily to program due to the simplistic nature of the logical map and reduce function, the lack of a high level language means that there is very little in terms of portability when multiple MapReduce stages are chained together. Take for example, a very common query in SQL, select * from table where column 1 > value 1 and column 2 > value 2. In MapReduce, there would be two stages, the first stage would filter by column 1 and the second column 2. The programmer would need to know to pass data from column 1 in first, then pass in data from column 2. The programmer cannot pass in an entire row with multiple columns because MapReduce only supports single (key,value) pairs. In this regards, MapReduce has often being compared with assembly programming, it is difficult to re-use a pre-existing MapReduce stage without knowing entirely the execution flow.

2. **Rigid Data Flow**
MapReduce runs in two discrete steps, the reduce step does not start until all the mapper steps have completed. Since MapReduce is a distributed system running on a variety of network, the possibility that there is a single straggler node is not out of the ordinary. In this situation, all compute nodes would have to wait on the straggler node. DBMS systems do not have this constraint, using the Schema and query logic, the scheduler will schedule as many local reduces as possible from free compute nodes, this results faster execution times [7].

3. **Lack of Schema or Indexes**
One of MapReduce's biggest advantages, flexibility is also one of its largest disadvantages. While not having a schema means that MapReduce jobs have very fast initiation times, DBMS systems will need time initially to massage data into appropriate acceleration structures. MapReduce's lack of these acceleration structures means that searching operations are considerable slower. A lack of schema also means that between multiple MapReduce stages, data has to be parsed again and again. In addition, the lack of a schema means that static compile time optimizations cannot be made, such as reductions in logic between multiple MapReduce stages.

C. Difference in Programming Model

MapReduce and DBMS represent an old discussion on two different views in methods to access data [6]. These are:

1. Relational view, state what you want rather than presenting an algorithm; this represents MapReduce.

2. Codasyl, an algorithm for data access; this represents query languages such as SQL.

The outcome of this discussion concluded that the relational view is easier to understand and modify. The Codasyl method was criticized for being "too assembly similar to assembly".

VI. MapReduce versus DBMS Performance

Hadoop MapReduce and Parallel DBMS systems compared in performance for three category of tasks on a 100 node cluster. The execution times between the two are presented in table III.

A. System Configurations

Both MapReduce and DBMS systems were deployed on a 100 node cluster. Each node has a single 2.40 GHz Intel Core 2 Duo with 4GB of ram and two 250GB SATA-I hard disks. All 100 nodes were connected using two gigabyte 128 Gbps Ethernet switch with 50 nodes per switch.

B. Grep Task

The first task benchmarked is the original grep task that was presented in the original Google MapReduce paper. This task involves searching for a specific keyword that occurred once every 1000 (key, value) pairs. The total data set consisted of 1 TB split amount 100 nodes. Per node, 10 GB of records consisting of 100B records was partitioned to assigned map tasks. This test was designed to measure pure IO throughput of the two systems. This grep task is designed such that both Hadoop MapReduce and DBMS-x are not able to take advantage of indexes. In addition, there is no reduce step in this experiment since when the specific keyword is found, it is outputted as the result.

One would expect Hadoop to perform well in comparison to DBMS-x since Hadoop has a much simpler initialization mechanism, simply load data via (key, value) pairs. However, referring to table III, Hadoop was 1.5 slower. This slow down can be attributed to the need for repetitive record record parsing. Since Hadoop does not have any special storage formats, such as DBMS data indexing and messaging, it is up to the use's code to read in and write out data in the appropriate format. In Hadoop's native language Java, (key, value) pairs are stored as immutable strings. This means anytime any modification on any (key,value) pairs are done, the original string is deleted and re-created in memory.

Secondly, DBMS systems are well matured products with highly efficient data compression algorithms. In the study, it was found that enabling compression on DBMS systems enabled performance by a factor of two to four III. On the other hand, there has not being a lot of work on Hadoop and data compression, at most a 15 percent performance boost was seen [5].

C. Web Log Task

The second task is a SQL aggregation, "GROUP BY" clause on a table of User Visits in a Web server log. The data set used is a 2TB data set with a total of 155 million records distributed over 100 nodes, translating to 20 GB of data per node. Each node runs the group by and records the total ad revenue for each IP address. Similar to the previous Grep experiment, all records must be scanned and there no advantage gained from indexing. This experiment involves a reduce step in order to collect all visits from different IP addresses.

Again, Hadoop is considerably slower than DBMS-x in performance. The addition of a reduce step, where (key,value) immutable string pairs have to be serialized and de-serialized may be the reason why this task has worse performance than the Grep task.

D. Join Task

The final task is a situation where MapReduce programming model is inefficient in handling. The task consist of a complex join operation over two tables which require both an aggregation and then filtering. The data set utilizes the User Visits from the previous join task joined with another 100 GB table of PageRank values that consist of 18 million URL's. The first part consists of the system locating the IP address that generates the most revenue within a set range in the User Visits data set. The intermediate result records are then joined with the PageRank table and used to calculate the average Page Rank of all the pages visited during this interval. Referring to table III, Hadoop is considerably, 36.3x slower than DBMS systems.

	Hadoop	DBMS-x	Hadoop/DBMS-x
Grep	284s	194s	1.5x
Web Log	1,146s	740s	1.6x
Join	1,1158s	32s	36.3x

TABLE I
HADOOP VS DBMS-X PERFORMANCE

VII. JOINS IN DBMS VS MAPREDUCE

The comparison of join tasks between Hadoop and DBMS-x demonstrates a large difference in execution times where Hadoop loses considerably. This section will give some insight as to why there is such a large performance difference. Take an example data set consisting of two tables, Employees and Department

Name	Age	Dept Id
Alex	26	2
Ben	24	2
Sarah	34	5

TABLE II
EMPLOYEES TABLE

Department Id	Name
5	Marketing
2	Engineering
3	Sales

TABLE III
HADOOP VS DBMS-X PERFORMANCE

Suppose the following Join was carried out, " Select * from Employees join Department on Employees.DepartmentId = Department.DepartmentId". When this query is run on a DBMS system, the system has two options, in most situations it will use the data's indexes to its advantage and carry out a hash join which is $O(1)$ time or at worst carry out a index join,$O(logN)$time. Also, all data representations are internally maintained in the database and no extraneous conversions are carried out.

The MapReduce execution of this query follows a similar path to that of DBMS, first a filter then an aggregate. However, since MapReduce's internal representation of the data has no indexes, hashing and divide and conquer techniques cannot be applied. Every operation between the two tables is an $O(n^2)$ operation [8]. In addition, there are extraneous serialization and de-serialization steps in between the map and filter steps. The following section gives an example flow diagram as well as mapper and reduce pseudo code to demonstrate the join.

A. *MapReduce Join*

Joins in MapReduce are run in $O(n^2)$ time due to lack of help from indexes or any table schema. In the first stage, the mapper maps both tables using the join column as the key, this produces the intermediate results as shown in figure 6, stage 1. In stage 2, the mapper's local aggregation pairs all values per key, now both employees and department records share the same key. In stage 3, the reducer runs a double for loop, looping through each employee record in the outer loop and then each department record in the inner loop, if the value's tag is not the same tag as the outer loop's tag, this means that these two values need to be joined. The reducer's code for stage is include in code listing VII-A, take note of line 5 as well line 6, where the tag comparison for tags as well join of values are run, these are expensive Java string operations.

```
1. reduce (K dept_id, list<tagged_rec> tagged_recs)  {
2.   for (tagged_rec : tagged_recs) {
3.     for (tagged_rec1 : taagged_recs) {
4.         if (tagged_rec.tag != tagged_rec1.tag) {
5.           joined_rec = join(tagged_rec, tagged_rec1)
6.   }
7.   emit (tagged_rec.rec.Dept_Id, joined_rec)
8.   }
9. }
```

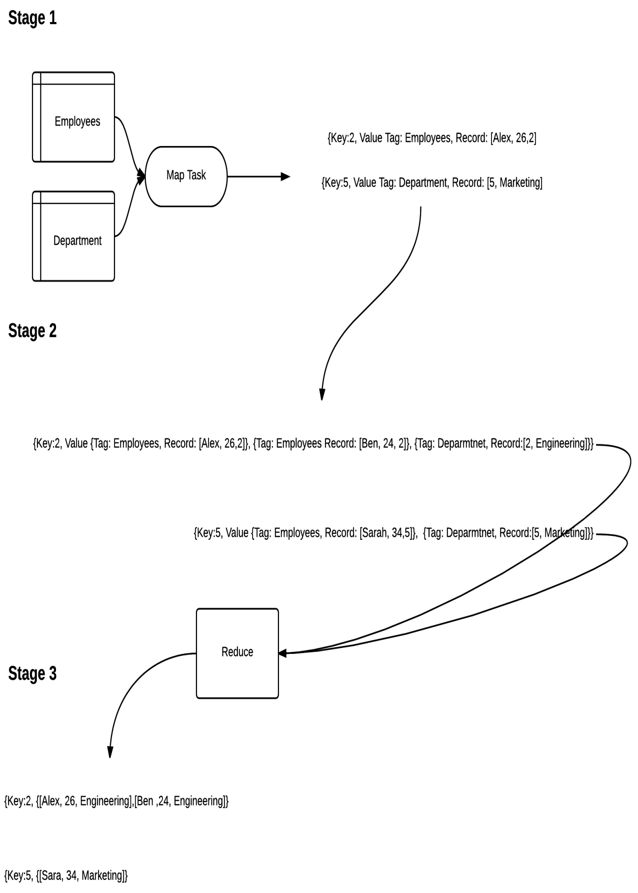

Stage 1
Employees
Department
Map Task
{Key:2, Value Tag: Employees, Record: [Alex, 26,2]
{Key:5, Value Tag: Department, Record: [5, Marketing]
Stage 2
{Key:2, Value {Tag: Employees, Record: [Alex, 26,2]}, {Tag: Employees Record: [Ben, 24, 2]}, {Tag: Deparmtnet, Record:[2, Engineering]}}
{Key:5, Value {Tag: Employees, Record: [Sarah, 34,5]}, {Tag: Deparmtnet, Record:[5, Marketing]}}
Reduce
Stage 3
{Key:2, {[Alex, 26, Engineering],[Ben ,24, Engineering]}
{Key:5, {[Sara, 34, Marketing]}

Fig. 6. Join stages in MapReduce

VIII. MapReduce Applications

Since the introduction of the MapReduce programming model and its widespread adoption there have being many application level frameworks designed to address the several issues covered earlier.

A. *High Level Languages*

There have being several efforts in order to incorporate a relational language on top of MapReduce. Higher level languages will enable a whole host of optimizations DBMS systems currently take advantage of. This include static query optimization and easier MapReduce module sharing among developers. These are Microsoft SCOPE[9], Apache Pig [10] and Apache Hive[11].

Pig is an open source project motivated by the Google MapReduce's scripting language built on top of Hadoop. Pig consists of two components, the language called Pig Latin and the execution framework. The language Pig Latin supports nested data models and pre-defined UDF's that can be customized. The Pig execution framework uses a logical analyzer to plan out the best query plan given the Pig Latin query input.

Hive is also an open source project built on top of Hadoop. Hive, like Pig supports a high level language called HiveQL that compiles directly into acylic graph MapReduce jobs. Hive also contains a schema model for the data as well provides run time staticis similar to DBMS.

SCOPE is a Microsoft framework that sits on top of Microsoft's own MapReduce called Cosmos. It provides expressions very similar to SQL as well as C# LINQ expressions. In addition SCOPE is the only framework out of the three that supports custom SQL views, providing an additional layer of flexibility for programmers to interact with data.

B. *Support for Schemas*

The earlier section VI and 6 demonstrated the disadvantages of schema-less systems. Repetitive record parsing and the in-ability to use acceleration data structures to speed up tasks such as joins cause up to a 36 times performance slow down. Though there has being very little research in this area, the earliest suggest that data formats such as Google's Protocol Buffers, XML, JSON, Apache's Thrift, or other formats can be modified in order to very basic data integrity and perhaps more advance data compression [7].

C. *Optimized Data Flow*

Due to the rigid execution model of MapReduce, sequential programs that incorporate loops suffer in performance due to the problem of loading repetitive data over multiple loops. Loops are very common in machine learning algorithms that apply a function repeatably tot he same input, optimizing a single parameter at a time. Take for example a simple program, program VIII-C that invokes MapReduce over a number of iterations.

```
for(int i = 0; i < 10; i++)
{
      MyMapReduceProgram(i);
}
```

For each iteration of the loop, the MapReduce program would have to read in the program's initialization data and output the iteration's results. Even though perhaps only one variable has changed in the data, the entire dataset has to be read in, in its entirety. To solve this problem, there have being several frameworks designed to identify loop invariant data to avoid loading and unloading unnecessary. These are, HaLoop [12], Twister [13], Pregel [14] and Apache Spark[15].

Both HaLoop and Twister work very similar to each other. These frameworks first identify invariant data between iterations, maintain them in memory. Twister accomplishes this by re-using previously instantiated workers with modified inputs between iterations. HaLoop takes this idea further by also caching input and output data per stage.

Pregel implements the Bulk Synchronous Parallel mode(BSP), in which each node owns its own input and output. BSP assumes that access to each node's own memory is fast and remote memory access is much slower. In this model, only the essential data is transfered between nodes through a communication and sync step.

C.1 Apache Spark

Apache Spark is a twist on the original MapReduce framework. It is intended to MapReduce's repetitive data loading from either loops or joins. Instead Spark tries to keep everything in memory. One of the reason MapReduce has such a rigid flow, with barriers between the map and reduce stage, where input, intermediate and output data is written to disk is that MapReduce has a very simplistic fault tolerance mechanisms. Spark instead takes on a more DBMS approach of categorizing data into read only objects called resilient distributed data sets(RDD). This allows spark to be more flexibility with what needs to be duplicated onto disk in order to ensure fault tolerance. In addition, Spark's language supports the ability for programmers to explicitly control the size and lifetime of RDD's. This gives the programmer a very big tuning knob in order to get the best performance from their programs. More relaxed RDD failure schemes give better performance since most data is in memory, but worse failure recovery time, and more strict RDD failure schemes give the best performance for highly faulty systems. Spark represents perhaps the most drastic over haul of the original MapReduce framework in order to deal with invariant data, and the performance demonstrates the effectiveness of RDD's.

C.2 Apache Spark Performance

A benchmark Spark program that measured the time it takes to complete a logical regression of a dataset was benchmarked versus Hadoop. The program iteratively attempts to find a hyperplane w that best separates two sets of points, figure 7 gives a 2D example. The logical regression accomplishes this through a gradient descent method, starting at a random w and over n iterations moves closer to the true w through finding the gradient. This test benefits greatly from caching input data (the points) since it does not change each iteration.

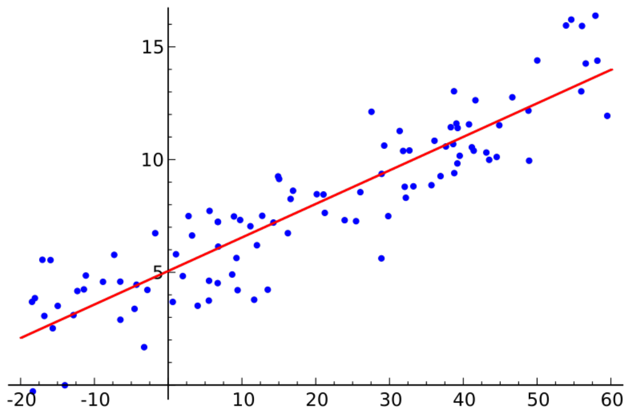


Fig. 7. Finding the optimal hyperplane between a dataset of points

This benchmark was run with a 29 GB data set on 20 "m1.xlarg" EC2 nodes with 4 cores each. From the result, figure 8, there is a clear trend, Hadoop's running time scales almost linearly with number of iterations, while Spark remains constant.

C.3 Application Level Frameworks

Apache Spark is tiered similar to Hadoop, it includes the lower level HDFS which the Spark Engine uses for fault tolerance. On top, there are 4 major application's. These are Spark SQL, Spark's relational query language, like SCOPE, Pig and Hive. Spark streaming, a real time stream data processing engine. MLlib a machine learning library and GraphX, a graph computation library.

IX. MapReduce Fault Tolerance

Hadoop has an extremely naive algorithm in order to handle failed nodes. When the master scheduler detects that a

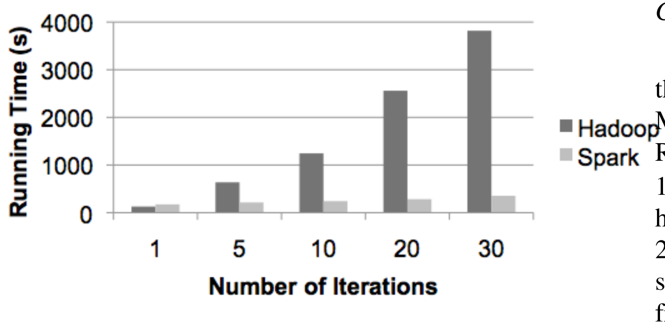


Fig. 8. Logical regression performance between Spark and Hadoop [15]

node has failed, it will simply restart that node in its entirety. Since Hadoop was built with fault tolerance as a first class citizen, failures in nodes are not abnormalities, large Hadoop clusters often experience a multitude of failures over the course of a day [**?**]. Lets take a practical example from Facebook to motivate why Hadoop's naive restart algorithm can become a source for slow down.

A. *Faulty Facebook Example*

Since failures are very common in MapReduce, most of the times from bad input data. Suppose that each input data has a bad record in the middle of its records. Each mapper will process half its data before it finds a fault and stops responding. Hadoop's default failure mechanisms is to try at least two additional on the same node times before it gives up. This results in a total of 8.5 (half of 17) ÃŮ 2 ÃŮ 200 500 seconds recovery time, about the same time if the no faults had occurred.

B. *Local Checkpoints*

Like DBMS, MapReduce would benefit from local checkpoints so that in the case of failures, a node can pick up a more optimized restart location. Several early attempts have being made in implementing such functionality but have not produced results that dramatically improve MapReduce's restart mechanism [16]. There are three challenges that any solution needs to resolve.

1. Checkpoints relies on the storage solution's replication engine. Increasing the number of checkpoints places additional strain on the storage.

2. Additional communication needs to be relayed intra compute nodes in order to make aware of the additional checkpoints.

3. Recovering from failure requires fetching checkpoint from intermediate storage, which may or may not involve additional network bandwidth.

C. *Checkpoint - Rafts*

The Raft recovery algorithm attempts to solve the above three issues by adopting a hybrid approach in between MapReduces naive implementation and the previous work. Raft provides three features,
1. Local checkpoints, these deal with local task failures are have a very quick recovery scheme.
2. Remote checkpoints, these checkpoints are written to a share file system, these happen when a node failures either from network/hardware or OS failures.
3. Query Meta-data, when a node fails, a restarting node will use this meta data to determine the most optimal restart point.

D. *Performance Benchmark*

A simple aggregation task, listing IX-D was performed. This task was chosen because the group by a considerable amount of intermediate results. The data set UserVisits consists of 50 GB, about 900 M (key, value) tuples.

Figure 9 demonstrates that as the number of bad records per split increases, Raft check pointing is a much more efficient failure mechanism resulting in lower total execution times.

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

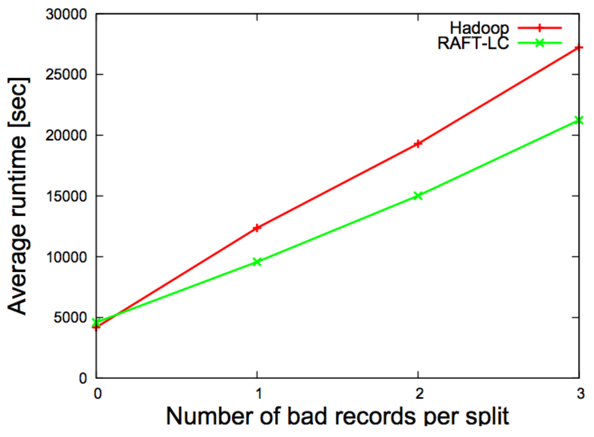


Fig. 9. RAFT-LC results [17]

X. Origins of MapReduce Concept

The ideas of map and reduce are not entirely novel. The strategy of partitioning data sets into smaller concurrent pieces has being around for the last 20 years. It was first proposed in the article, Application of Hash to Data Base Machine and Its Architecture [18]. In a following article, "Multiprocessor Hash-Based Join Algorithms"[19], Gerber describes how partitioning scheme can be extended in parallel to handle joins on a Shared- Nothing architecture.

This technique of partitioning data into independent chunks, Shared-Nothing architectures as well as using memory to speed up computation has being used and refined in commercial DBMS systems since the 1980's.

MapReduce advocates might argue that the move away from a relational SQL like language is what differentiates MapReduce from DBMS systems of yesterday, but POSTGRES systems included support for user-defined mappers and aggregates since the mid 1980's. DBMS systems are not usually associated with the Codasyl method because the Database community as a whole decided that the relational view was easier to understand logically and more portable.

REFERENCES

[1] John Gantz and David Reinsel, "The digital universe decade-are you ready," *IDC iView*, 2010.

[2] David DeWitt and Jim Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[3] Shreekant S Thakkar and Mark Sweiger, "Performance of an oltp application on symmetry multiprocessor system," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE, 1990, pp. 228–238.

[4] Bowers Shaw, "Database management systems," University Lecture, 2009.

[5] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: Simplified data processing on large clusters, osdiâĂŹ04: Sixth symposium on operating system design and implementation, san francisco, ca, december, 2004," *S. Dill, R. Kumar, K. McCurley, S. Rajagopalan, D. Sivakumar, ad A. Tomkins, Self-similarity in the Web, Proc VLDB*, 2001.

[6] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 165–178.

[7] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 1029–1040.

[8] Foto N Afrati and Jeffrey D Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 99–110.

[9] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.

[10] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1099–1110.

[11] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[13] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[14] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, pp. 10–10, 2010.

[16] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel R Madden, "Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 657–668.

[17] Jorge-Arnulfo Quiane-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 589–600.

[18] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, pp. 63–74, 1983.

[19] David J DeWitt and Robert Gerber, *Multiprocessor hash-based join algorithms*, University of Wisconsin-Madison, Computer Sciences Department, 1985.