

# **An Investigation into whether Shared Cyclical File Scanning Improves the Performance of Concurrent Processing Jobs within the MapReduce Distributed Computing Framework**

A dissertation submitted in partial fulfilment  
of the requirements for the Open University's  
Master of Science Degree  
in Networks and Distributed Systems

Russell Martin  
X977594X

**4 March 2012**

Word Count: **14998**

## **Preface**

My thanks and appreciation is given to Dr Robert Walker, the project supervisor. His advice and feedback was always welcome and proved invaluable from start to finish.

I would also like to thank Robert Drake for proof-reading this dissertation as well as my wife Wendy and daughter Lauren for their patience and understanding throughout the MSc.

## Table of Contents

Preface.....	i
List of Figures.....	v
List of Tables.....	vi
Chapter 1 Introduction .....	1
1.1 Background to the Research .....	1
1.2 Aims and Objectives of the Research Project .....	2
1.3 Overview of the Dissertation .....	4
Chapter 2 Literature Review .....	5
2.1 An Overview of MapReduce.....	5
2.1.1 The MapReduce Programming Model .....	6
2.1.2 MapReduce Distributed File Systems .....	8
2.1.3 MapReduce Model and DFS Summary.....	9
2.2 Measuring Performance Improvement.....	10
2.2.1 Measuring Performance.....	11
2.2.2 Job Types and Input Data .....	12
2.2.3 Cluster Specification.....	15
2.2.4 Experimental Technique Summary .....	17
2.3 Evaluating MapReduce Enhancements.....	18
2.3.1 Optimising MapReduce .....	19
2.3.2 Model Function Changes.....	22
2.3.3 Stage Barrier Reduction .....	23
2.3.4 Summary of Qualitative Evaluation Criteria .....	26
2.4 Shared File Scanning.....	27

2.4.1	Shared Scanning in MapReduce.....	27
2.4.2	Shared Scanning in Database Systems.....	30
2.4.3	The Impact of Storage Media.....	31
2.4.4	Shared Scanning Summary.....	34
2.5	Literature Review Summary.....	35
Chapter 3	Research Methods .....	36
3.1	Re-Examining the Research Question .....	36
3.2	Research Techniques.....	37
3.3	Research Plan .....	42
3.3.1	Experiment Design .....	42
3.3.2	Prototype Development.....	46
3.4	Summary .....	49
Chapter 4	Data Collection .....	50
4.1	Cluster Configuration .....	50
4.2	Experimental Configuration .....	51
4.2.1	Experiment 1: Varying Degrees of Concurrency .....	51
4.2.2	Experiment 2: Slow Running Jobs .....	52
4.2.3	Experiment 3: Varying Input File Sizes .....	52
4.3	Summary .....	53
Chapter 5	Results .....	54
5.1	Impact of Increasing Client Numbers.....	54
5.2	Impact of Slow Running Jobs.....	56
5.3	Impact of Increasing File Sizes.....	58
5.4	Evaluation of SCFS .....	60
5.5	Addressing the Hypotheses and Research Question .....	62

5.6 Research Evaluation .....	64
5.7 Summary .....	66
Chapter 6 Conclusions .....	67
6.1 Project Review .....	68
6.2 Future Research .....	70
References.....	74
Bibliography.. .....	80
Index.....	81
Appendix A - Extended Abstract .....	82
Appendix B - Source Code .....	88

## List of Figures

Figure 1: Jobs Utilising a Shared Cyclical Scan .....	3
Figure 2: The Functional Definition of MapReduce .....	6
Figure 3: Mappers and Reducers Executing within a Cluster .....	7
Figure 4: The Architecture of HDFS.....	9
Figure 5: Asymmetric Join on MapReduce .....	20
Figure 6: The Impact of Tuning MapReduce .....	21
Figure 7: MapJoinReduce Performance against TPH-H Q4.....	23
Figure 8: Performance Impact of Stage Barrier Reduction.....	25
Figure 9: Scans Sharing a Cache .....	28
Figure 10: Non-Cyclical Shared Scans .....	29
Figure 11: The Elevator Scan Strategy .....	31
Figure 12: The Impact of Simultaneous Access to Larger Files.....	33
Figure 13: Steps for Conducting Controlled Experiments.....	42
Figure 14: Sample Input Data.....	45
Figure 15: The SCFS MapReduce Prototype.....	48
Figure 16: Average Job Completion Time (Experiment 1) .....	56
Figure 17: Average Job Completion Time (Experiment 2) .....	58
Figure 18: Average Job Completion Time (Experiment 3) .....	60

## List of Tables

Table 1: Experiment 1 Results (Varying Concurrent Clients) .....	55
Table 2: Experiment 2 Results (Introducing a Slow Job) .....	57
Table 3: Experiment 3 Results (Varying Input Size) .....	59

## **Abstract**

MapReduce enables computation to be conducted on a massive scale by harnessing a cluster of computers to solve a particular problem, managing issues like result aggregation and fault tolerance on behalf of application developers.

This research investigates an enhancement to MapReduce known as Shared Cyclical File Scanning (SCFS). SCFS attempts to reduce the overhead incurred by MapReduce when conducting intensive I/O operations, when data is read from relatively slow media. This is achieved by ensuring that files are only being read once at any one time. This approach is similar in principle to existing research. Agrawal et al. (2010), for example, adjust the MapReduce job scheduling algorithm so that jobs requiring access to the same file execute simultaneously and can share non-cyclical scans.

SCFS works by distributing the data from a single scan of the file to all currently executing jobs that require it rather than each job reading from the same file independently. Jobs arriving partway through an existing scan receive data from the point at which they join the scan. Once the scan has reached the end of the file it restarts. This allows these jobs to receive the portion of the file that they did not receive initially.

It is shown that comparing the performance improvement produced by different MapReduce enhancements is difficult given the variation in the approaches to benchmarking. This is partly mitigated by the development of qualitative criteria against which a MapReduce enhancement can be assessed.



A prototype system is then developed that implements SCFS in MapReduce. Experiments are conducted to benchmark the prototype's performance under different conditions and the results compared to the performance of an unmodified MapReduce system. These experiments involved varying the number of concurrent jobs, introducing slow running jobs and consuming larger volumes of input data. Finally, SCFS is compared to the criteria established to evaluate MapReduce enhancements.

These experiments showed that there are circumstances where SCFS improves performance. On this particular cluster the prototype outperformed the baseline system when there were eight or nine simultaneous jobs sharing the same scan. Outside of this range the prototype performed worse. Furthermore, a single slow job significantly degraded the prototype's performance. Increasing the size of the input files when there were only a small number of concurrent jobs did not affect the relative performance of the two systems.

The overall conclusion is that whilst SCFS can improve the performance of MapReduce, the improvement is far from universal with the majority of circumstances examined yielding performance degradation. The only circumstances that led to an improvement were high numbers of simultaneous scans of the same file. This may not be experienced frequently in practice.

## **Chapter 1 Introduction**

### **1.1 Background to the Research**

Modern organisations are increasingly reliant upon analysing large volumes of data to provide critical business insights and gaining a competitive edge (Gupta et al., 2010). Facebook and the New York Stock Exchange are just two organisations that generate vast quantities of data, much of which requiring intensive analysis to identify trends or support conclusions (White, 2009).

Processing increasing volumes of data requires increasingly large amounts of computing power. Michael et al. (2007) investigated different means of increasing computing power based on parallelism - that is, increasing the number of processors assigned to a task. The two classifications of these techniques are 'scaling up' and 'scaling out'. Scaling up equips a single machine with more and/or better processors, whilst scaling out utilises an increasingly large cluster of independent machines. Michael et al. (2007) concluded that scaling-out offered indisputable performance and cost advantages. However, utilising clusters is not trivial with issues such as distributed data management, fault tolerance and job scheduling needing to be resolved (Liu et al., 2010).

MapReduce addresses many of these issues (Dean and Ghemawat, 2008). It is based on the observation that many parallelisable tasks can be divided into the same two abstract operations, Map and Reduce. Developers only have to provide application-specific logic for the Map and Reduce functions. The MapReduce framework itself

deals with the periphery issues that result from distributed processing. This means that it is possible to develop distributed applications with little knowledge of distributed systems. There are numerous implementations of MapReduce, including Apache's "Hadoop", Nokia's "Disco" and AsterData's "MapReduce Appliance" (Monash, 2008).

Since the data volumes and the demands of users will only increase, improving the performance of MapReduce is a topic of interest in industry and academia. It is hoped that this research will contribute to this field.

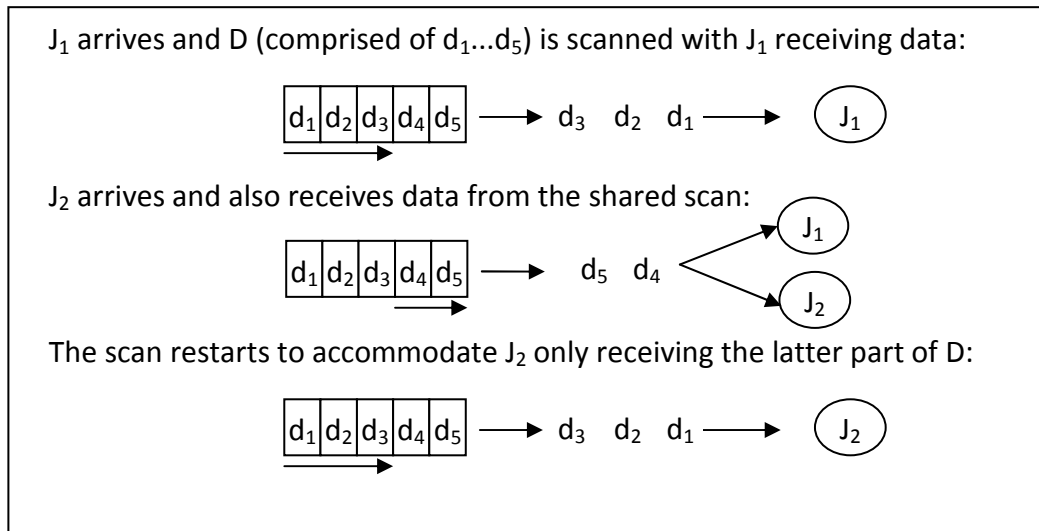
### **1.2 Aims and Objectives of the Research Project**

The aim of this research is to investigate whether a particular enhancement, Shared Cyclical File Scanning (SCFS), can improve the performance of a MapReduce cluster. This enhancement allows multiple jobs to share the same input stream rather than each job accessing the file system independently.

MapReduce is designed to run on commodity hardware where I/O is typically a bottleneck, with the CPU often waiting for input from the relatively slow media (Qiao et al., 2008). As many MapReduce jobs are I/O (rather than CPU) bound (Agrawal et al., 2008), there is a large potential for improvement if the cost associated with reading in data could be reduced. SCFS, which is illustrated in Figure 1, aims to reduce the time spent conducting I/O by sharing this overhead amongst multiple jobs.

Consider a data file  $D$  required by jobs  $J_1$  and  $J_2$ .  $J_1$  arrives and begins to read  $D$ .  $J_2$  arrives whilst  $J_1$  is part-way through executing.  $J_2$  could either wait for the existing scan

to finish (causing higher latency) or start another scan (causing less effective I/O) - both inefficient. SCFS allows  $J_2$  to start receiving its scan of D part-way through the existing scan for  $J_1$ . Then, once the end of file is encountered, the scan then restarts to provide the remaining data to  $J_2$ . Additional jobs requiring the data in D can essentially hook into the shared-scan at any point.



**Figure 1: Jobs Utilising a Shared Cyclical Scan**

Whilst it may be intuitively obvious that sharing overheads between jobs will lead to a performance improvement, there are situations where this turns out not to be the case (Johnson et al., 2007). For instance, when evaluating the benefits of sharing memory scans in a multi-CPU system, the performance benefit did not keep pace with the number of processes sharing the scan and eventually deteriorated (Qaio et al., 2008).

Therefore, the specific research question which this dissertation seeks to address is:

*"To what extent can shared cyclical file scanning improve the performance of concurrently executing jobs within the MapReduce distributed computing framework?"*

This dissertation has two main objectives. First, it aims to evaluate SCFS in the context of the wider body of knowledge available about MapReduce and related topics. Second, it aims to demonstrate whether, and in what circumstances, SCFS can improve MapReduce's performance.

### **1.3 Overview of the Dissertation**

This dissertation starts with a literature review in Chapter 2 which provides an overview of MapReduce and examines existing framework enhancements. This provides a means of evaluating SCFS relative to other enhancements and helps establish the research methodology. The literature review also examines the theory behind shared scanning in general and proposes several hypotheses concerning the performance of MapReduce when SCFS is introduced.

The remainder of the dissertation addresses the research question by investigating these hypotheses and evaluating SCFS. The chosen research method is explained in Chapter 3 with Chapter 4 documenting its execution. The results are presented in Chapter 5. Finally, the project is concluded in Chapter 6 where ideas for further work are presented along with a review of the project.

## **Chapter 2 Literature Review**

To investigate whether SCFS can improve MapReduce it is first necessary to understand the framework itself. Section 2.1 enables this by presenting an overview of MapReduce and its underlying Distributed File System (DFS).

Previous attempts to improve MapReduce's performance are analysed in section 2.2, although the primary purpose of this section is to examine the experimental methodologies employed during previous research. Section 2.3 continues to analyse existing enhancements but with the primary focus of establishing qualitative criteria against which SCFS can be assessed.

Finally in section 2.4, the focus shifts to investigating the concept of shared file scanning. This leads to the proposal of a set of hypotheses that predict the impact of SCFS on the performance of MapReduce.

### **2.1 An Overview of MapReduce**

MapReduce is a model for writing software suited to tasks involving large quantities of data as its highly-parallel nature allows for massive scalability across potentially tens of thousands of machines (Lam, 2011). The framework deals with distribution issues such as fault tolerance (Dean and Ghemawat, 2008) and is explained in section 2.1.1.

In most implementations MapReduce uses a DFS to allow data to be distributed among multiple nodes but accessible as though it were stored locally on each node. This is described in more detail in section 2.1.2.

### 2.1.1 The MapReduce Programming Model

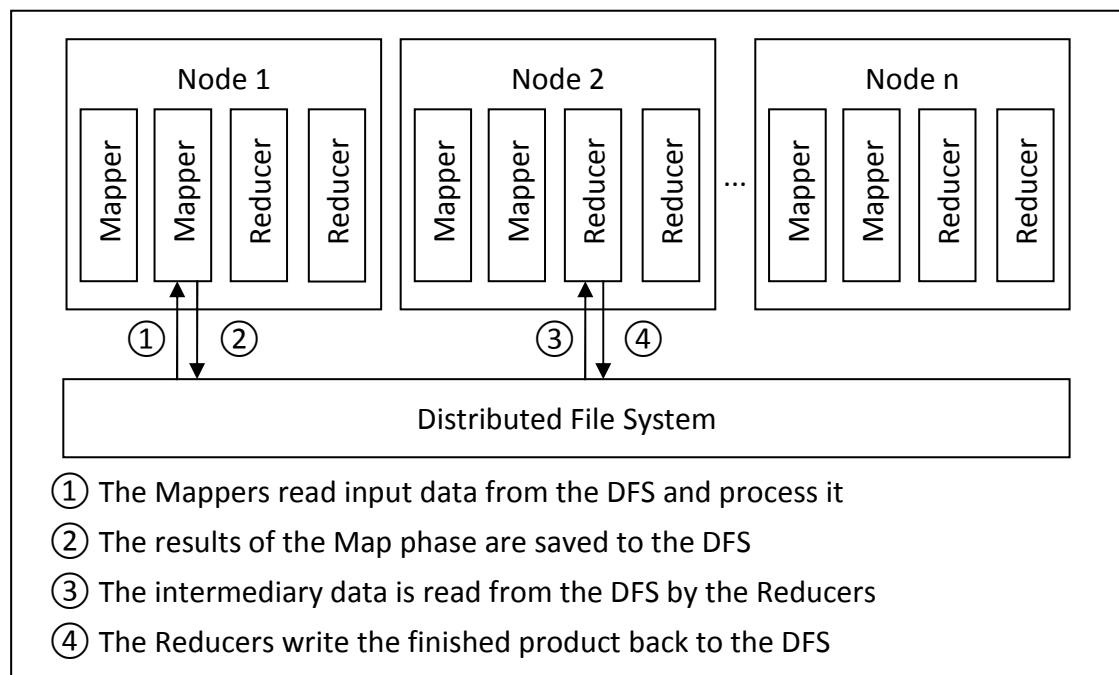
Creating a MapReduce application requires the implementation of two functions, Map and Reduce, presented formally in Figure 2. Map and Reduce are commonly considered as 'filter' and 'aggregate' operations respectively (Pike et al., 2005) and it is the developer's job to express their application in terms of these functions.

$\text{Map: } (\text{key}_1, \text{value}_1) \rightarrow \text{list}(\text{key}_2, \text{value}_2)$ $\text{Reduce: } (\text{key}_2, \text{list}(\text{value}_2)) \rightarrow \text{list}(\text{value}_3)$
---

**Figure 2: The Functional Definition of MapReduce**

The Map phase is conducted by several 'Mappers', with each working on a subset of the overall problem, such as part of the input data. They conduct processing that can be conducted independently from other Mappers which may be executing elsewhere in the cluster, as shown in Figure 3. The results from the Map phase are then transferred to the Reducers ready for the Reduce phase to start. This transfer takes place in an intermediary step known as the 'shuffle', which is a cluster-wide synchronisation point where the output from the Map phase is committed to the DFS.

This introduces a performance bottleneck (Verma et al., 2010) but without it a cluster failure may require entire queries to be restarted. This would be problematic for long-running queries taking several weeks to execute (Xie et al., 2010).



**Figure 3: Mappers and Reducers Executing within a Cluster**

The performance of MapReduce has come under considerable criticism, especially when compared to SQL-based databases. Hadoop can be 2-3 times slower than many major database products when conducting word-count tasks and slower still for many complex analytics (Xu, Y et al., 2010). However, the approach of MapReduce is to sacrifice per-node efficiency in favour scalability, meaning performance can be bolstered by simply adding hardware (Shafer et al., 2010).

Databases are underpinned by indexes, allowing rapid retrieval of data, whilst data analysis with MapReduce is scan-intensive. This means in situations where only small subsets of the data are relevant, considerable time may be spent examining irrelevant data (An et al., 2010). It is this scan-intensive nature of MapReduce, demonstrated by many jobs spending over 20% of their time loading input data during the Map phase (Wang et al., 2011), that SCFS intends to exploit.

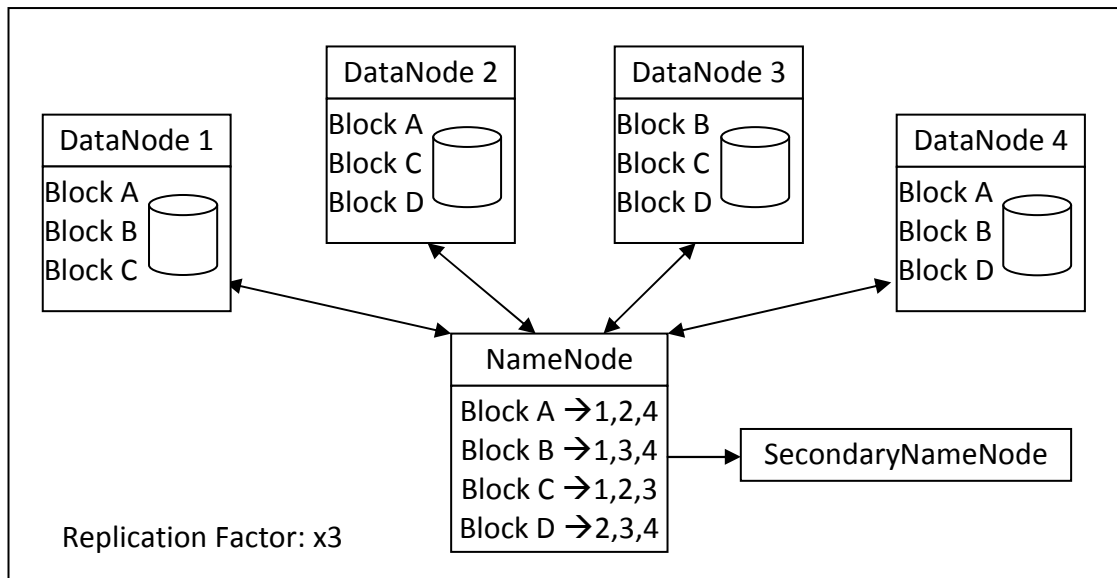


### 2.1.2 MapReduce Distributed File Systems

MapReduce is underpinned by a DFS that allows files to be stored on nodes across the cluster and accessed from any other node, even if they must be read via a network connection. There are many types of DFS available ranging from the Network File System (NFS), which implements a simple client-server model (Callaghan et al., 1995) to Lustre, which allows data to be distributed across thousands of nodes (Schwan, 2003).

However, not all types of DFS are suitable for MapReduce. The DFS must be highly scalable, fault tolerant and support extremely large files (Moise et al., 2010). Google's proprietary MapReduce implementation uses the Google File System (GFS) whilst by default Hadoop uses the Hadoop Distributed File System (HDFS). These have been used to manage hundreds of terabytes of data across large clusters at Google and Yahoo! Inc. (Ghemawat et al., 2003; Shvachko et al., 2010). GFS and HDFS share many similarities (Thanh et al., 2008) including their fault tolerance capabilities and use of large block sizes to facilitate high-throughput data transfers by minimising disk seek times.

The architecture for HDFS is illustrated in Figure 4. Data is replicated to provide fault tolerance with periodic 're-balancing' to maintain the desired level of replication. Metadata is stored on a master node (or 'NameNode') and is queried by slaves to determine the location of a particular file. A 'SecondaryNameNode' can be used to backup the metadata, since its loss would be catastrophic to the file system.



**Figure 4: The Architecture of HDFS**

The relatively poor performance of MapReduce often results from the underlying DFS. Although HDFS is used by default in Hadoop it does not always suit the requirements of the cluster or applications being executed. Hadoop supports a wider range of file systems, such as local disk access or an alternative DFS (Dadan et al., 2009). This allows the DFS be tailored for the specific circumstances involved, for example, by removing the replication overhead when there is less need for fault tolerance (Shafer et al., 2010). This is the approach being employed in this research - attempting to improve performance by amending the underlying DFS, in this case to support new functionality.

### 2.1.3 MapReduce Model and DFS Summary

MapReduce is an increasingly popular platform to manage distributed computation across a cluster. However, performance is often sacrificed in favour of fault tolerance and scalability. As much of the time taken to complete a job is spent reading data, SCFS

is targeting part of the lifecycle that could yield a significant improvement. The scan-centric nature of MapReduce jobs means that SCFS can potentially offer a means of improving the performance of the underlying DFS, and hence the framework overall.

## **2.2 Measuring Performance Improvement**

The previous section provided a basic understanding of MapReduce and showed that there is potential for SCFS to improve its performance. This section analyses how SCFS's impact on performance can be measured by examining previous attempts to improve the framework's performance. The enhancements examined in this section all relate to the lower-level issue of job scheduling - deciding when and where tasks are executed. As well as examining the experimental techniques employed, this approach helps put any performance improvement into context, allowing a more informed judgement of whether any improvement resulting from SCFS is significant.

Section 2.2.1 starts by establishing what metrics can be collected to measure performance and different techniques to collect them. Section 2.2.2 examines the type of jobs being executed and the input data being processed during experiments. Finally, section 2.2.3 examines the different clusters that are used to run experiments. All three of these sections relate to decisions that will need to be taken if SCFS is to be evaluated, regardless of the precise research method chosen. The end result is a set of recommendations to provide the most robust and comparable experimental results possible.

### 2.2.1 Measuring Performance

SCFS alters the operation of the underlying MapReduce DFS. It is therefore logical to consider measuring the impact of SCFS in terms of the throughput that the DFS can achieve. For example, if SCFS can serve 5 concurrent jobs at a rate of 2Mb/second, then the effective throughput of data (10Mb/second) might be considered greater than if these 5 jobs can each achieve a rate of 1Mb/second independently.

One technique to measure this is to record file system events, such as files being open and closed for reading. These events can then be aggregated across the DFS (Baker et al., 1991). The problem with this approach is that data is not necessarily being transferred whilst files are open. Therefore these events do not measure the actual data throughput of the MapReduce cluster effectively.

Another technique for recording performance metrics within a MapReduce DFS is the use of 'synthetic micro-benchmarks' (Moise et al., 2010). This involves creating a custom DFS client which directly accesses the storage layer functionality. The client is then able to record metrics such as throughput and latency itself. This is a step closer to measuring the impact that the DFS changes have on the MapReduce cluster it serves. However, it is still not possible to easily relate this to what most MapReduce users would care about - the time taken to complete jobs.

The most common way of measuring the impact of the change is therefore to simply measure the time taken to perform a set of jobs before and after the enhancement

has been implemented. This approach is used by most of the researchers who investigate specific performance enhancements, for example, Verma et al. (2010), Jiang and Wang (2010) and Xie et al. (2010).

An exception to this is Dhok et al. (2010), who investigated an enhancement that predicted how demanding a job will be and rejects/defers jobs that will lead to the cluster being overstretched. Rather than expressing performance in terms of time taken to complete jobs, they quote the proportion of jobs completed within the user-specified deadline. Under the new admission control policy, the number of jobs completed before the deadline rises from less than 2% to 87% for a particular set of jobs. Whilst this might be considered a performance improvement, at least for customers whose jobs were not rejected, it is difficult to relate this metric to any of the other performance enhancements assessed.

Measuring performance in terms of the time taken to complete a set of jobs is, in most circumstances, both the easiest to measure and easiest to relate to existing research. It is therefore recommended to use this as a performance metric.

### **2.2.2 Job Types and Input Data**

As performance will be measured by recording how long jobs take to complete, it is necessary to consider what these jobs will be and what input data they will process.

Some researchers, e.g. Babu (2010), utilise standardised benchmarking jobs, such as the TeraSort (Nyberg et al., 2012). This is a benchmark used to assess a cluster's ability

to sort large volumes of data. Unfortunately, these types of benchmark can be very demanding and are designed for high-end systems backed by well resourced organisations. The "TeraByte Sort" metric, the competition for which the TeraSort job was written for, is now deprecated with the 100TB "GraySort" taking its place. This particular benchmark may be out of scope for the clusters used to test many MapReduce enhancements. For example, Zhang et al. (2009) only had 7 nodes totalling 0.5TBs of storage space.

Despite the formal deprecation, it is still possible to publish metrics against the TeraSort and other lower-grade benchmarks. These help provide a consistent means of comparison between enhancements which is lost when non-standard proprietary jobs and datasets are employed exclusively. It is therefore recommended to utilise one of these benchmarks when assessing an enhancement's impact on performance.

Another problem with using the TeraSort is that it only measures one single activity - sorting. Real life MapReduce clusters will have to conduct a wider variety of tasks ranging from I/O-intensive tasks where the actual processing conducted is relatively trivial to CPU-intensive tasks where the processing is more intensive. An alternative benchmark known as MRBench (Kim et al., 2008) utilises a Transaction Processing Performance Council (TPC) benchmark that has been adapted for MapReduce. The TPC is a database industry benchmarking organisation and produce benchmarks to independently rate the performance of database products. MRBench has adapted the 'H'-series benchmark, which includes a set of test data and queries to execute on it, for

MapReduce. This benchmark tests a range of functionality including searching, sorting, parsing and merging of datasets.

Although TeraSort and MRBench are rarely employed when assessing MapReduce enhancements, they offer an authoritative performance benchmark given their independence. It is therefore recommended that, if possible, they are included in any benchmarking exercise.

Many researchers use proprietary jobs rather than relying upon industry standard benchmarks. Tian et al. (2009) and Aboulnaga et al. (2009) both investigated improvements that relied upon examining whether jobs were I/O or CPU-intensive and used this information to influence when and where they were executed. For example, by pairing a CPU-bound and I/O-bound task together there is less contention for the same hardware resources on each node.

These enhancements were both tested by submitting a range of I/O and CPU-bound tasks to the cluster simultaneously. These jobs included sorting, searching and word counts. The best result from these experiments reduced completion time for a set of jobs by 26%, from 350s to around 260s. However, this could be considered artificial as the range of I/O and CPU-bound tasks would have allowed complementary pairs of tasks to be formed more readily than might be the case in practice.

This demonstrates that some improvements favour jobs dominated by I/O-bound work, others favour CPU-bound work and others favour a mixture of the two. SCFS will

most likely favour I/O-bound work. The requirements of real life MapReduce deployments will also vary, with some conducting mostly I/O-bound work, some mostly CPU-bound work and some a mixture. The best test of an enhancement is therefore to use different sets of jobs. Some should be dominated by I/O-bound work, some by CPU-bound work and some more equally balanced. This will ensure that the performance improvement quoted is more likely to be achieved in practice.

The enhancements investigated so far have revealed a variation in the choice of job used and input data used to benchmark the performance. This is a recurring theme with there being very few genuinely comparable enhancements.

### **2.2.3 Cluster Specification**

Along with the types of job to be run, another important decision when measuring a performance improvement is what cluster any experimentation should be run on. As scalability is an important property of MapReduce (Dean and Ghemawat, 2008), the enhancement should in general be tested on a large a cluster as possible to ensure that it does not constrain a MapReduce cluster's scalability.

Just as there was a large variation of job types, there is an equally large variation in the size and specification of test clusters. This ranged from the use of a 100 node cluster (Jiang et al., 2010) to only 5 nodes (Xie et al., 2010). The specification of the nodes also varied considerably ranging from powerful machines with Quad-Core CPUs and 16GB RAM (Verma et al., 2010) to machines with single cores and only 2GB RAM (Zhang et



al., 2009). This variability further compounds the inability to directly compare different MapReduce enhancements.

Limited by the available equipment, some researchers have utilised virtualisation to increase the number of nodes. One physical machine running a 'hypervisor' such as Xen (Barham et al., 2003) can support many guest machines. As MapReduce is designed to run on large clusters, this offers a means of providing resource-constrained researchers the ability to create a cluster with a large number of nodes.

When benchmarking MapJoinReduce, Jiang et al. (2010) simply rented a 100-node cluster from the Amazon cloud, saving them from having to purchase equipment. Chen et al. (2010) used a virtualised cluster of 10 machines running upon 5 physical hosts when benchmarking a modified job scheduler, SAMR, which takes into account a more detailed view of a node's specification when assigning tasks.

The use of virtualisation by Jiang et al. (2010) and Chen et al. (2010) is questionable. Hosting multiple nodes on a single physical machine artificially inflates the node count without increasing the actual resources available. In the case of the Amazon cloud, one has little awareness of the true load of the underlying hardware or even the relative locations of each node. This limits the repeatability of the experiments. In the case of SAMR, the contention for resources could impact upon the algorithm that attempts to determine the capability of the nodes. For example, a node could believe it has more RAM than it effectively has because the physical resource is shared with another node.

It is therefore recommended that virtualisation should not be used when benchmarking a MapReduce enhancement.

#### **2.2.4 Experimental Technique Summary**

This section has examined the experimental technique employed by others. It has lead to several recommendations to minimise the variation between results produced in this research and that of existing research. These recommendations also ensure that the methodology employed is as robust as possible:

1. Performance should be measured by recording the time taken to complete a defined set of jobs. This metric is the easiest to relate to existing research.
2. Standardised benchmarks, such as the TeraSort, with pre-defined data sets should be employed to maximise comparability with other enhancements.
3. Experimentation should examine how the enhancement performs with I/O-bound jobs, CPU-bound jobs and a mixture of the two.
4. Virtualisation should be avoided as it artificially inflates the cluster size causing more contention for the same hardware.

The actual performance improvements encountered were significant in many cases, including a best-case 26% reduction in job completion times from the scheduler amendments implemented by Abounaga et al. (2009). However, the most notable issue was the large variations in experiment execution. This includes different cluster

sizes, node specification, job types and input data. This limits the extent to which any performance improvement can be compared between enhancements.

### **2.3 Evaluating MapReduce Enhancements**

The previous section showed that it is not possible to conduct meaningful comparisons between different enhancements because of variations in the experimental stages of the research. This is a recurring issue with few enhancements being truly comparable. This section therefore establishes more general qualitative criteria against which SCFS can be assessed by examining further enhancements that have not yet been discussed. These criteria offer an additional means of evaluating MapReduce enhancements and take into account a wider range of issues - not just the performance improvement itself.

The enhancements encountered in the previous section related to low-level issues such as scheduling and their introduction to MapReduce was mostly transparent. The enhancements examined in this section have a more noticeable effect on issues other than performance. For example, some reduce the framework's scalability. These are therefore more useful for establishing the required qualitative criteria.

These enhancements can be classified into three main groups. The first group, optimisations, do not actually change the framework. Instead they improve performance by tuning the configuration of a cluster or amending how applications achieve their goals. These optimisations are examined in section 2.3.1. Whilst

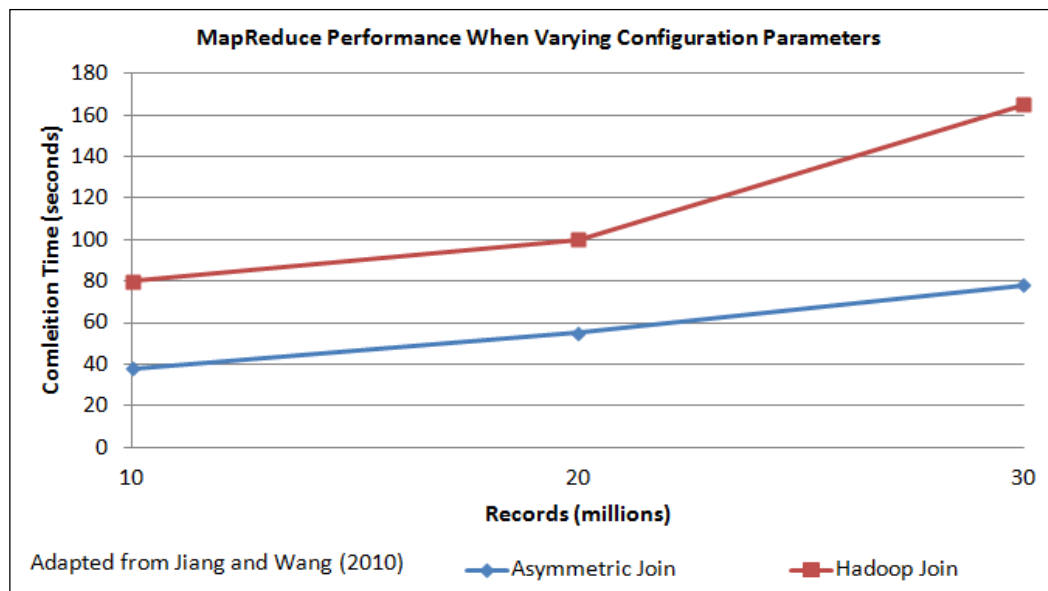
optimisation can yield significant performance improvements, performance is still fundamentally constrained by the MapReduce framework implementation.

The remaining improvements therefore involve modifying the framework. Section 2.3.2 examines the second group, which alter the fundamental definitions of MapReduce (the functions in Figure 2) and have the most profound effect upon the framework. The third group relates reducing the impact of the shuffle operation (defined in section 2.2.1), where data is transferred between stages. These enhancements have a significant impact on a MapReduce job's lifecycle and are examined in section 2.3.3.

### **2.3.1 Optimising MapReduce**

MapReduce is often employed to conduct data processing tasks that it was not specifically intended for, such as merging heterogeneous datasets. The asymmetric join (Jiang and Wang, 2010) improves the performance of this specific operation in MapReduce by caching the smaller of the datasets locally at each node. This data is referred to when needed rather than reading both datasets from the DFS.

In benchmarking experiments, this lead to an improvement of over 100% by reducing the time taken to merge 30 million records from over 160s to less than 80s, as shown in Figure 5:



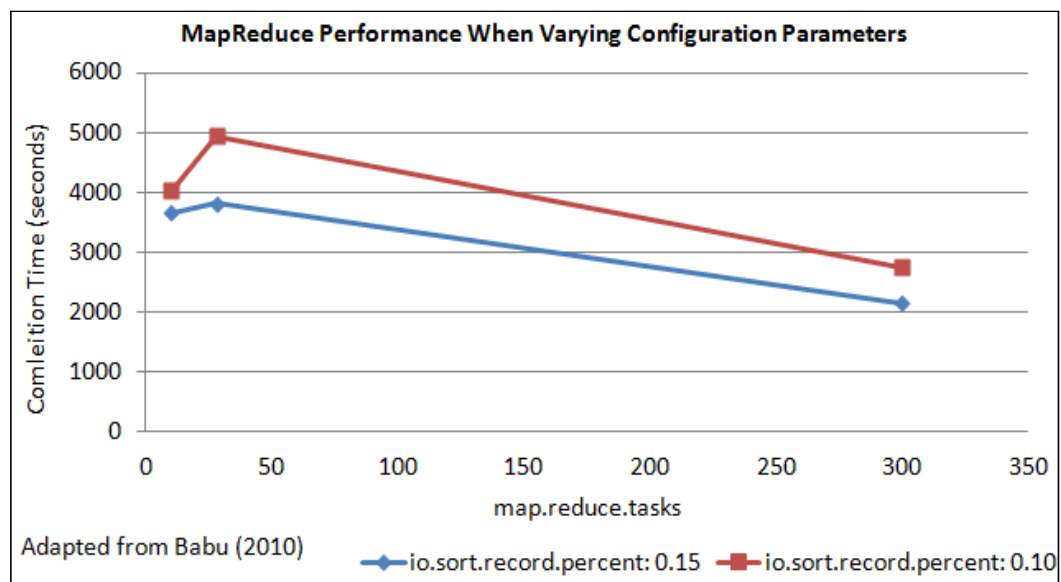
**Figure 5: Asymmetric Join on MapReduce**

Whilst the performance improvement is substantial and does not require any changes to the framework, it is specific to one type of job. Jobs that do not involve merging data or merge two equally large tables would not benefit. This raises the issue of how widely applicable an enhancement is. Ideally, an enhancement should be as widely applicable as possible as MapReduce is intended to be general purpose (White, 2009).

As well as improving how user-defined jobs are written, performance can also be improved by tuning the cluster's configuration so that it is optimised for a particular set of jobs (Babu, 2010). The Hadoop MapReduce implementation has numerous obscure tuneable parameters which can have a significant impact upon performance.

Figure 6 illustrates how the 'mapreduce.reduce.tasks' and 'io.sort.record.percent' parameters impact upon the running time of TeraSort. The first parameter specifies the maximum number of simultaneous Reducers on a single node whilst the other

controls the memory that can be used to sort the Mapper's output. When executing on a 17-node cluster, varying these parameters can reduce execution times from over 80 minutes to less than 40 minutes. However, the interactions between variables are complex with the initial increase in Reducers being detrimental to performance.



**Figure 6: The Impact of Tuning MapReduce**

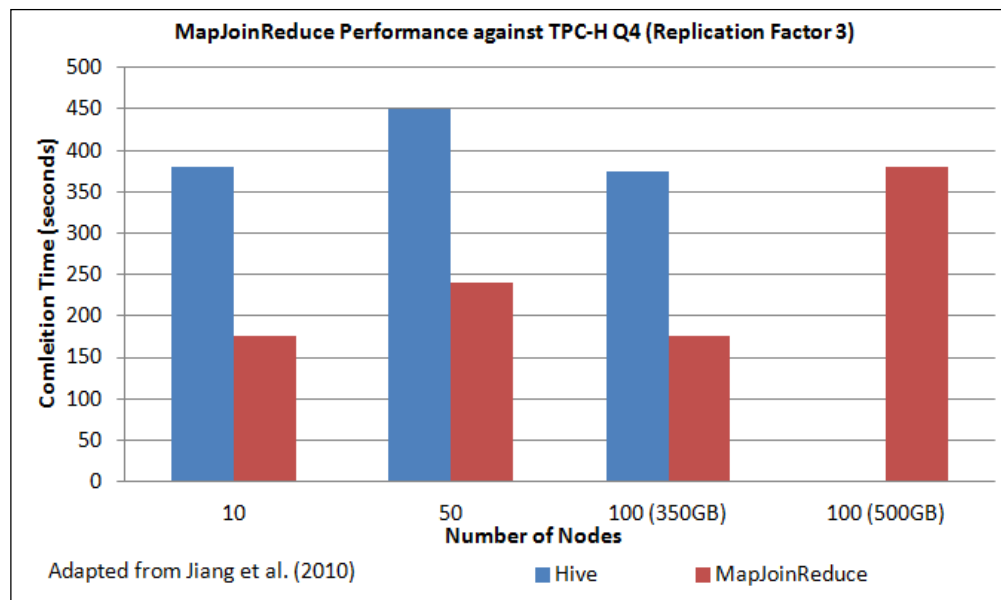
This shows that varying just two parameters can result in a significant performance improvement. Unfortunately, tuning a MapReduce cluster is complex and is often left incomplete (Sharma, 2010). The current trend in MapReduce development is to become more user-friendly via products which sit above MapReduce and shield the users from the underlying complexity. For example, Hive (Hive, 2011) translates high-level SQL statements into low-level MapReduce jobs. Increasing the amount of configuration required will increase the complexity faced by the cluster's users/administrators. The additional configuration introduced by an enhancement should therefore be minimal.

### 2.3.2 Model Function Changes

The enhancements examined so far have largely involved tweaking low-level aspects of the framework, such as the scheduling algorithm or certain configuration parameters. MapJoinReduce (Jiang et al., 2010) and MapReduceMerge (Yang et al., 2007) improve MapReduce's ability to fuse heterogeneous data sources by altering the fundamental definitions of the Map and Reduce functions. This has a profound impact upon the definition of MapReduce itself. As a result, users must now implement three functions, each slightly different to the original functions presented in Figure 2.

When benchmarking MapJoinReduce, Jiang et al. (2010) did not compare it directly to MapReduce but to Hive, the interface that translates SQL instructions into MapReduce jobs. The TPC-H benchmark is employed, leading to a very authoritative benchmark where MapJoinReduce outperforms Hive regardless of the cluster or replication factor used. In fact, Hive could not execute the queries for the full 500GB dataset in some circumstances meaning that it had to be reduced. The results are shown in Figure 7. MapJoinReduce performed some queries in around half the time of Hive with some even achieving up to 3x the speed.

Aside from its scope being limited to relational data processing, this style of enhancement suffers a fundamental problem. By changing the functional definitions the processing model is no longer MapReduce. This means that existing MapReduce jobs will not run in without alteration.



**Figure 7: MapJoinReduce Performance against TPH-H Q4**

MapReduceMerge and MapJoinReduce are relatively minor departures and the effort required to translate existing jobs to the new framework is relatively trivial. Other enhancements are, however, more extreme. For example, GenerateCombine (Sarje and Aluru, 2010) customises the model for processing tree-structured data. It bears little resemblance to the original model, and therefore, is incompatible with existing MapReduce applications. Ideally, a MapReduce enhancement should still be compatible with existing applications, requiring little or no effort to convert them.

### 2.3.3 Stage Barrier Reduction

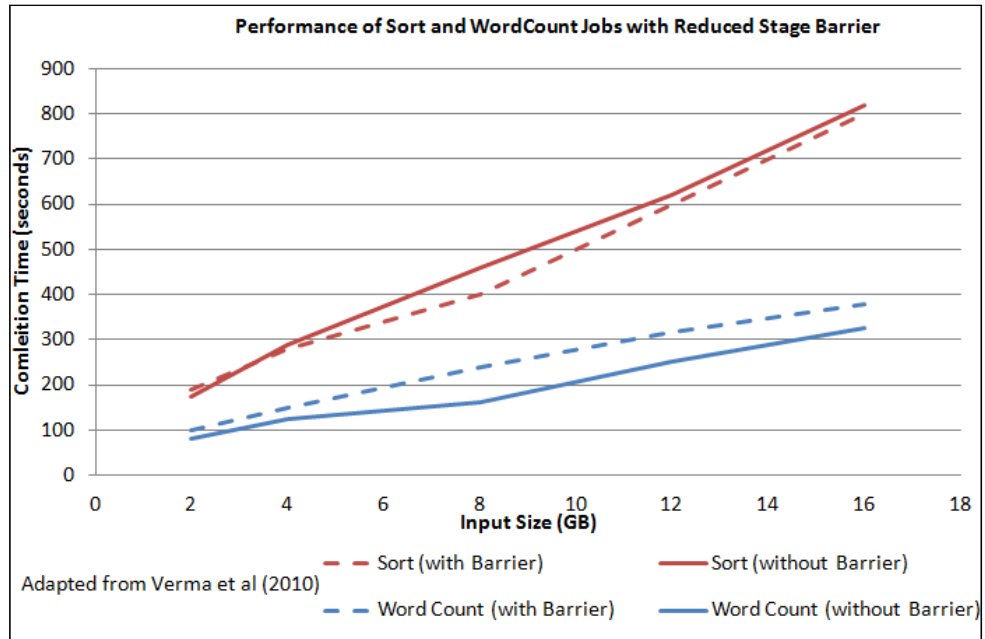
The changes to the functions discussed in the previous section represent the most fundamental change possible to MapReduce. Although the enhancements examined in this final group are not as high-level as model changes, they still have a significant impact on the framework. These enhancements attempt to mitigate the performance constraints introduced by the stage barrier. This is the synchronisation point between



the Map and Reduce phases, where the shuffle operation writes the results from the Map phase to the DFS for the Reduce phase to use as its input data. The Reduce phase can only begin once all of the Map tasks have finished and the results have been written to the DFS. This process contributes to MapReduce's fault tolerance and simplicity, but constrains its performance (see section 2.1.1).

Two approaches to mitigating the impact of stage barrier are pipelining (Verma et al., 2010) and utilising a distributed memory cache (DMC) (Zhang et al., 2009). Pipelining works by drip-feeding data continuously between the stages as it is produced rather than doing it in a single monolithic operation. The provision of a DMC provides a faster mechanism to exchange data between nodes, effectively introducing a secondary DFS. The single monolithic synchronisation point between the Map and Reduce phases still exists but the reliance upon slower file systems is reduced in favour of faster RAM.

Rather than benchmarking these enhancements with a standardised benchmark, such as TeraSort, Verma et al. (2010) and Zhang et al. (2009) use a range of proprietary applications. The results for the sorting and word counting applications for the pipelining approach are shown in Figure 8. Overall, the word counting, graph processing and genetic algorithm applications yielded a performance improvement of up to 87% but the performance of the sort task degraded. Although precise data is not provided for the DMC's performance, both the word count and spatial join applications yielded an improvement. The improvements in many cases were substantial, such as reducing the word count's time from 1000s to 500s in a 2-slave cluster.



**Figure 8: Performance Impact of Stage Barrier Reduction**

A major disadvantage of pipelining is that it burdens the application developer with more work. For example, the Reduce-stage code now has to account for data arriving after the execution has started rather than having all the required data available from the start. As application simplicity is a key aim of MapReduce (Dean and Ghemawat, 2008), this is a significant drawback. Performance enhancements should therefore aim to add as small a burden on the developer's workload as possible.

Unlike with pipelining, the developer is not burdened with any additional complexity when utilising the DMC and there is no need to account for it when writing MapReduce applications. However, since the DMC is non-persistent fault tolerance, a founding principle of MapReduce (Dean and Ghemawat, 2008), is weakened. This means that a DMC is only suitable for smaller clusters where failure is less common. Although Zhang et al. (2009) acknowledge this disadvantage there is

limited investigation into what point using a DMC becomes detrimental, with the cluster size only reaching six slaves.

These two enhancements have illustrated two additional criteria to assess a MapReduce enhancement. First, since MapReduce is intended to simplify distributed programming, the additional effort required when implementing the Map and Reduce functions should be minimal. Second, since MapReduce relies upon being scalable to achieve massive computational power (Shafer et al., 2010) enhancements should not constrain the number of nodes supported in a cluster.

#### **2.3.4 Summary of Qualitative Evaluation Criteria**

The variation in the experimental methodology employed makes direct comparison with other enhancements difficult. The following questions have therefore been derived to offer an additional means of assessing the quality of SCFS as an enhancement:

1. Does SCFS apply to a broad range of tasks or is its scope limited?
2. Does SCFS create the need for additional cluster configuration?
3. Does SCFS introduce backwards incompatibility with existing MapReduce jobs?
4. Does SCFS make MapReduce applications more complex?
5. Does SCFS constrain MapReduce's scalability and fault tolerance?

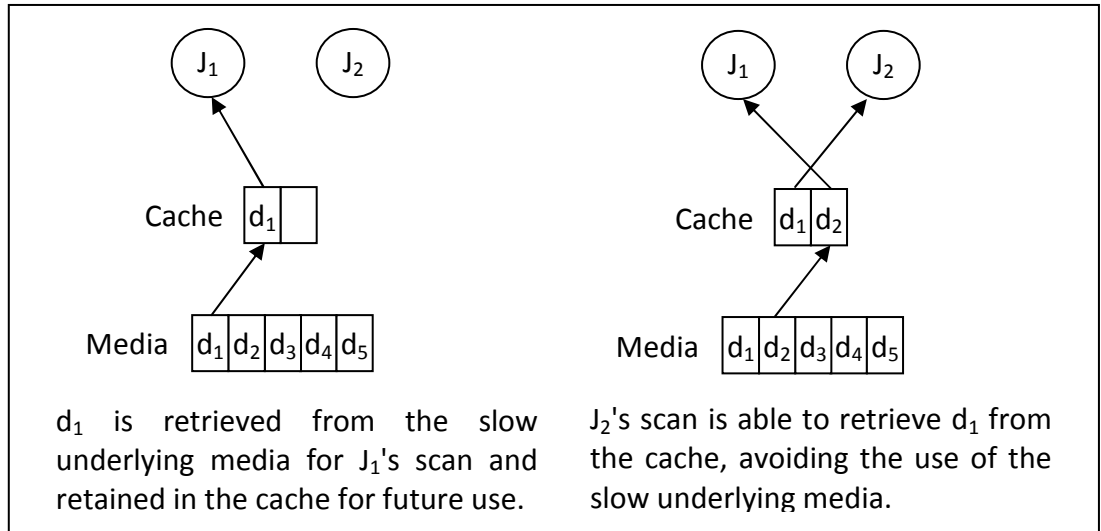
## **2.4 Shared File Scanning**

The previous two sections have examined a range of MapReduce enhancements that do not directly relate to SCFS. The focus now shifts to examining the concept of shared scanning, which underpins SCFS. This section examines other attempts to apply the concept of sharing the overhead of file scanning among multiple consumers. This leads to four hypotheses being formed about how SCFS will impact upon the performance of MapReduce.

Previous attempts to apply shared scanning to MapReduce are examined in Section 2.4.1. Section 2.4.2 examines attempts to apply shared scanning to database systems, which share many similarities with MapReduce, such as having to process large quantities of data. Finally, Section 2.4.3 examines the relatively poor performance of traditional storage media - the key assumption that drives attempts to share scans between multiple consumers.

### **2.4.1 Shared Scanning in MapReduce**

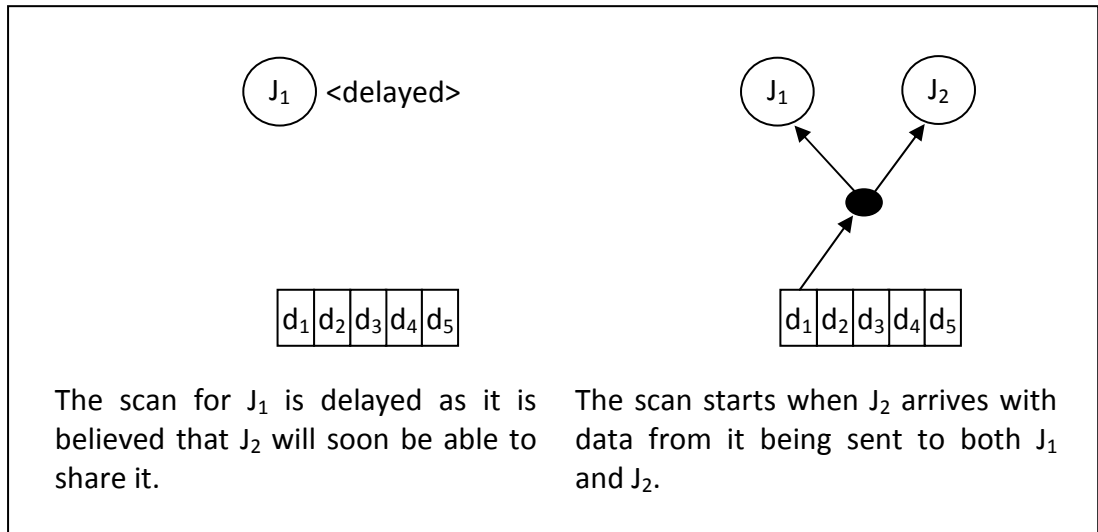
When identifying opportunities to share work amongst jobs across the whole MapReduce job lifecycle, Nykiel et al. (2010) identify sharing the scanning of input data as one potential opportunity. By grouping jobs together that require access to the same data and executing these simultaneously, it is more likely that the required data for one scan will already be cached. This is illustrated in Figure 9, with job  $J_2$  being able to retrieve  $d_1$  from the cache as  $J_1$  had previously accessed it. By making more use of the cache, reliance upon slower media is reduced.



**Figure 9: Scans Sharing a Cache**

The strategy investigated by Nykiel et al. (2010) was based on implicitly sharing a scan. There may still be multiple simultaneous scans of the same file - it is just that some will utilise the cache. An alternative approach adopted by Agrawal et al. (2008) is to use just one scan to directly feed multiple concurrent jobs. This was achieved by amending the MapReduce scheduling algorithm to predict when jobs requiring the same data are likely to be submitted. If it is likely that another job will soon be submitted that could share a scan with a job that is about to be executed, then the current job is delayed. This means the both jobs can start at the same time and share the same scan of the file. This is shown in Figure 10, with  $J_1$  being delayed until  $J_2$  arrives.

The rationale behind these approaches and SCFS is the same - performance will improve if less use is made of slow media. This leads to the first hypothesis. As the number of concurrent jobs utilising the same file increases, the fixed scanning overhead is shared across a larger number of jobs (Nykiel et al., 2010; Agrawal et al. 2008). SCFS will therefore provide greater performance relative to not utilising SCFS when the number of jobs sharing the scans increases.



**Figure 10: Non-Cyclical Shared Scans**

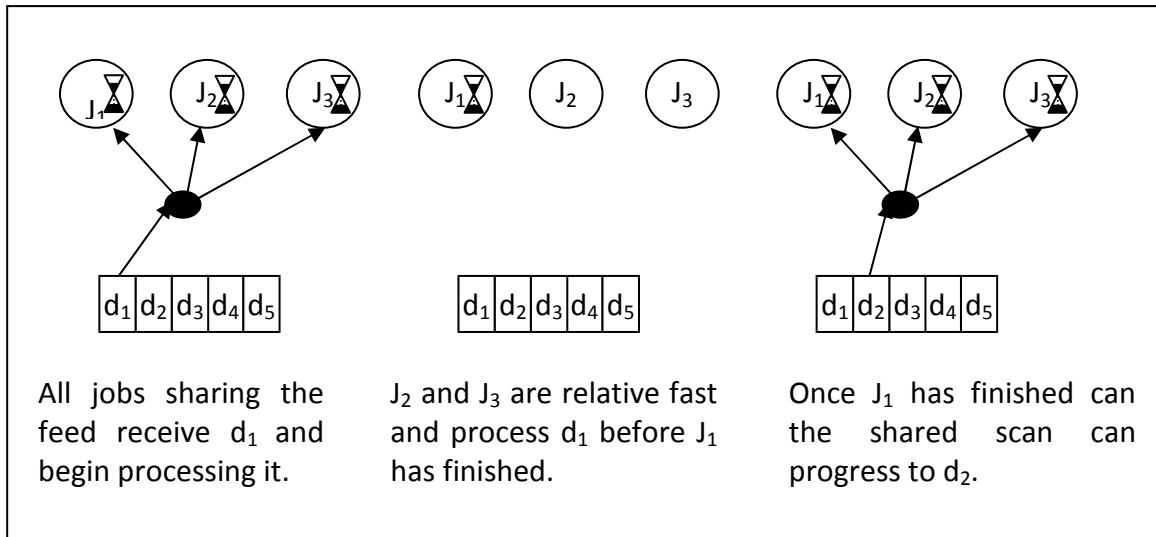
SCFS does have a key advantage over each of these existing approaches. Neither the use of caching or non-cyclical shared scanning actually guarantees there being only one concurrent scan of a file. They merely reduce the likelihood of multiple scans being required. SCFS, however, strictly enforces there being only one scan per file at any time, further reducing the reliance upon the slow media.

However, both of these approaches provide the contents of the file being scanned from start to finish. SCFS may, however, provide some jobs with data part-way through the scan with the remaining part received later. This means that jobs requiring strictly in-order data (e.g. pre-sorted data) are discriminated against. SCFS has therefore already suffered a blow against the first criteria established for assessing MapReduce enhancements as its scope is limited to jobs that can handle data being presented in a different order to which it is stored.

### 2.4.2 Shared Scanning in Database Systems

Numerous database vendors have implemented shared scanning, such as Microsoft's SQL Server (Kotidis et al., 2001). Original attempts to share I/O operations between different concurrent queries relied upon caching, with one query being able to retrieve data from a buffer as a result of another query having recently retrieved that data item from the disk. Queries following the 'convoy' pattern can exploit data already in the cache (Qiao et al., 2008), an idea similar in concept to how MapReduce jobs can share data via a cache (Nykiel et al., 2010). This type of sharing, referred to as a 'normal' sharing policy (Zukowski et al., 2007), suffers the limitation of there being only a small window of opportunity for sharing data. If queries are not sufficiently synchronised, the likelihood of the cache contents being useful is reduced.

A strategy to improve the likelihood of the cache containing the required data is to examine what data is currently being read by existing queries. If there is one query reading data in a range required by another query, it is possible to instruct the other query to start reading its data from this location. Zukowski et al. (2007) refer to this as the 'attach' strategy. One drawback is that it is possible to attach one scan to another that is considerably faster or slower, leading to the queries becoming detached and the buffer becoming less useful.



**Figure 11: The Elevator Scan Strategy**

The 'elevator' policy (Zukowski et al., 2007) addresses this issue. It strictly enforces the order in which data is read for all queries and ensures that queries never detach. This means attached queries and can always share I/O operations. This is the closest match to SCFS. A disadvantage of this is that if there is just one scan per file then some tasks may have to wait for the scan to reach the relevant data before they can start conducting useful work. Also, if a set of jobs accessing a particular file share the same scan then they are constrained by the slowest job. This is illustrated in Figure 11 where Jobs  $J_2$  and  $J_3$  must wait for  $J_1$  to finish before they can progress.

This leads to the second hypothesis. As an implementation of the elevator policy, the overall performance of MapReduce under SCFS will be hindered by slow jobs.

### 2.4.3 The Impact of Storage Media

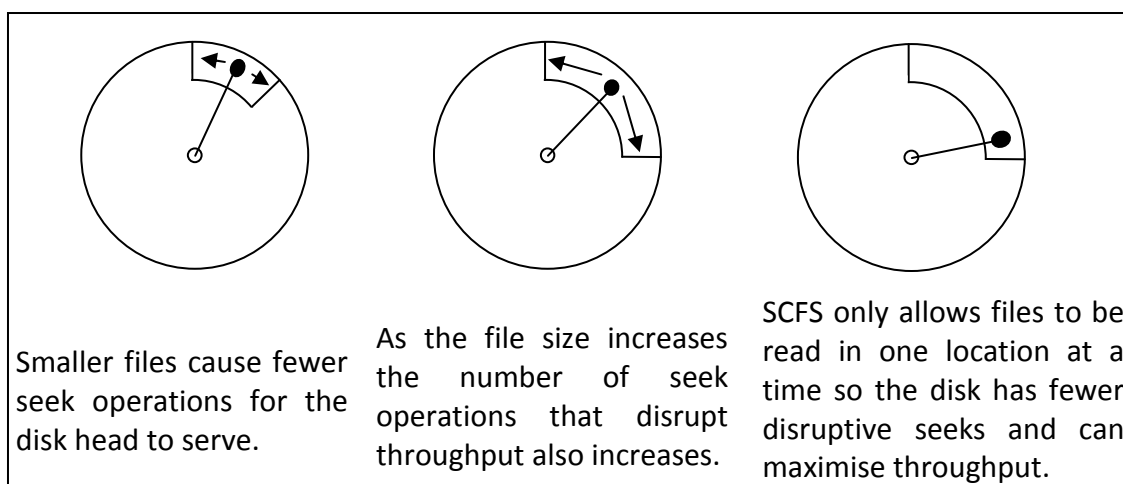
As many MapReduce jobs are I/O bound (Agrawal et al., 2008), the performance of the underlying media will be a large factor that determines whether SCFS improves MapReduce's performance. A key criteria for evaluating the performance of a storage



device is the seek time (Ruemmler and Wilkes, 1994). On traditional magnetic hard disk drives (HDDs), this translates to the time taken for the disk head to move to the part of the disk containing the required data. This is a relatively slow operation, taking in the order of milliseconds (Grochowski and Halem, 2003). If the disk is forced to jump between different locations repeatedly, then seeking will become more significant and reduce the productive throughput of data that the disk can achieve.

The third hypothesis is derived from this behaviour of the HDDs present in the vast majority of commodity hardware in MapReduce clusters. As different jobs request access to different parts of the input file then the disk must service these requests by conducting slow seek operations. If the size of the input file increases the likelihood of different parts of the disk being accessed simultaneously also increases. This is partly because the larger file occupies a larger area of the disk as well as there being more data in the file for jobs to read. As a result, the number of seeks will increase, causing the overall throughput of useful data to decrease if larger files are scanned independently.

However, with SCFS the file can only be accessed at one location at a time, reducing the number of expensive seek operations. This is illustrated, in a much simplified means, in Figure 12. Therefore, the third hypothesis is that as the input size grows, the relative performance of SCFS will improve.



**Figure 12: The Impact of Simultaneous Access to Larger Files**

Many models developed to optimise the performance of shared scanning are derived from assumptions that are true on traditional magnetic HDDs, but not on more modern storage media, particularly solid state drives (SSD) (Xu, C. et al., 2010). For example, the seek time on a typical HDD (Western Digital 7200rpm) is 9.96ms, compared to just 0.03ms on SSD (Intel X-25) - two orders of magnitude smaller. Although currently a relative niche product, solid state media is set to become common place (Kennedy, 2011) - if one believes the marketing hype - potentially invalidating the assumptions behind SCFS.

Models working on these old assumptions will attempt to reduce disk seeks at the expense of not utilising the drive's bandwidth. This would be unsuitable for SSDs due to its reduced seek times being less of a constraining factor and its higher throughput not being utilised. If SCFS yields an improvement because it reduces the amount of high-latency seeks being conducted as fewer locations on the disk are being accessed simultaneously, the benefit may be reduced if the underlying hardware were changed.

This leads to the final hypothesis. If SSDs are used as the underlying storage technology then the benefit of SCFS will be reduced compared to when traditional magnetic HDDs are used.

#### **2.4.4 Shared Scanning Summary**

Shared scanning has been employed in a variety of systems with implementation ranging from implicitly sharing scans via a cache to strict enforcement of a single scan. The following hypotheses predict how SCFS will perform when applied to MapReduce:

**Hypothesis 1:** As the number of concurrent jobs utilising the same file increases, the fixed scanning overhead is shared across a larger number of jobs. Therefore, the performance of SCFS relative to having independent concurrent scans will improve.

**Hypothesis 2:** As an implementation of the elevator policy, the overall performance of MapReduce under SCFS will be hindered by slow running jobs.

**Hypothesis 3:** As the input size grows, the detrimental impact of increased seek times to serve concurrent scans increases when scans are not shared. This means that the relative performance of SCFS will improve.

**Hypothesis 4:** If solid state media is used instead of traditional magnetic media then any potential benefit of SCFS will be reduced.

## **2.5 Literature Review Summary**

Although there are numerous enhancements proposed to MapReduce, including some that rely on the concept of shared scanning (Agrawal et al., 2008; Nykiel et al., 2010), no current implementation has yet employed SCFS. Many MapReduce jobs are I/O intensive and rely upon scanning in whole files rather than utilising an index (An et al., 2010). This means SCFS has potential to improve the performance of MapReduce.

The literature review examined the experimental technique that can be employed to measure an enhancement empirically. This concluded with recommendations regarding the use of virtualisation technologies, the choice of jobs and datasets and the metrics to record. However, the variation in research methodologies makes objective comparison between enhancements difficult. This difficulty has been partly mitigated by the development of qualitative criteria to evaluate a MapReduce enhancement, such as impact on the framework's scalability.

Finally, the concept of absorbing the overhead of scanning data amongst multiple recipient jobs has been utilised in a variety of systems. Examining these applications has led to four hypotheses that predict how different scenarios will impact upon the performance of SCFS in MapReduce.

The remainder of this dissertation seeks to address the research question by investigating the accuracy of these hypotheses and assessing SCFS against the established qualitative criteria for evaluating MapReduce enhancements.

## Chapter 3 Research Methods

This chapter describes the methodology employed when conducting the primary research. Section 3.1 re-examines the research question to determine what information is needed to answer it. Section 3.2 outlines the overall approach and shows that prototyping was the most suitable research technique. Section 3.3 describes in more detail how prototyping was used to investigate the proposed hypotheses. Finally, section 3.4 provides an overview of the prototype's design and operation.

### 3.1 Re-Examining the Research Question

The original research question was:

*"To what extent can shared cyclical file scanning improve the performance of concurrently executing jobs within the MapReduce distributed computing framework?"*

This question requires an answer that gives an indication of the scale of any performance benefit provided by SCFS. This could range from a significant improvement that justifies further research to a performance degradation which shows that there is little purpose continuing. Given the large variation in the experimental methodology across other research shown in sections 2.2 and 2.3, performance metrics will be difficult to compare with those from other enhancements.

The answer to this question should also account for the qualitative criteria summarised in Section 2.3.4. This is necessary to determine whether or not any performance benefit is worth the price in terms of these criteria. One might argue, for example, that a 5% performance improvement which significantly hindered MapReduce's scalability is not worth pursuing.

As MapReduce is deployed in many different environments, the chosen research method will need to test the enhancement's performance in as many circumstances as possible. The hypotheses developed in Section 2.4 provide the basis for the range of circumstances that SCFS will be evaluated against. These are varying the number of clients, varying the size of the input data, varying the speed of the jobs and varying underlying storage technologies.

In summary, there needs to be some indication of the scale of performance improvement in as many of the hypothesis-derived circumstances as possible as well as an evaluation of SCFS against the qualitative criteria.

### **3.2 Research Techniques**

Primary research methods can be classified into four main groups: surveys, field experiments, case studies and laboratory experiments (Sharp et al. 2002). This section shows that surveys, field experiments and case studies would be inappropriate and, that of the laboratory techniques available, prototyping is the best choice to address the research question.

Surveys offered extremely limited potential to address the research question. Consulting with subject experts may have yielded interesting insights, but would not provide tangible evidence of performance improvement/degradation. As surveys will not be able to provide any metrics to satisfy the research question, it is not a viable research technique.

Case studies and field experiments rely upon observing a technique being employed in real-life circumstances. This could involve observing how SCFS performs on an organisation's production cluster, for example. This may provide the best evidence of performance improvement since the environment will be very close to how the technique may be used in practice. However, this may limit the scenarios that can be feasibly examined. This significantly reduces the ability to answer the research question which requires that as many circumstances as possible are investigated.

On a more practical matter, as no MapReduce SCFS implementation currently exists one would have to be developed. Any such product developed within the resource constraints of this project would most likely lack the required testing and maturity to be employed in such an environment. Therefore case studies and field experiments are not viable research techniques.

Laboratory experiments would involve investigating SCFS in a more tightly controlled environment than that offered by a field experiment. The two main types of laboratory experiment to be considered are modelling and prototyping.

Modelling would attempt to predict the performance of SCFS under various conditions in a completely theoretical environment. Agrawal et al. (2008) developed a model to predict how different scheduling algorithms influence the performance of (non-cyclical) shared-file scans. Simulation packages such as MRSim (Hammoud et al., 2010) can also be used to evaluate cluster design decisions.

Modelling offers some significant advantages. It would allow a wide variety of scenarios to be tested rapidly by simply adjusting variables. It would probably be possible to model circumstances relating to all four hypotheses. This can be done without a potentially expensive cluster or having to consider every detail that would need to be accounted for when creating a working implementation of SCFS.

A disadvantage of modelling is that discoveries are purely theoretical and may not be achievable in practice. For example, some of the assumptions required to construct the model may be incorrect or too simplistic. However, this is acceptable given that only an indication of the performance improvement is required.

The most serious disadvantage of modelling in this context is the inability to assess many of the qualitative evaluation criteria. Whilst it may be possible to speculate about some of the issues it will not be possible to address them authoritatively without experiencing them first hand. For instance, the simplified nature of the model would prevent the additional cluster configuration required under SCFS from being fully explored.



The alternative laboratory technique, prototyping, would involve implementing a crude but functioning SCFS system and benchmarking its performance. Much of the research into MapReduce enhancements reviewed utilised prototyping. Zhang et al. (2009), Verma et al. (2010) and others suggest enhancements, implement them and then conduct controlled experiments to benchmark performance.

Prototyping offers the potential to demonstrate that a theoretical concept can work in practice (Open University, 2007). It also provides an opportunity to benchmark the performance of the prototype under more realistic conditions than a modelled environment. Any performance metrics will therefore be actual and not predicted. Furthermore, it will provide practical experience that could inform the qualitative evaluation - such as experiencing the complexity involved with configuring SCFS.

As prototyping can be considered an intermediary stage between the theoretical approaches of modelling/simulation and the practically-focused field testing, some advantages of the alternative techniques are eroded. Although the prototype is a tangible system, the experiments are still executed in controlled laboratory conditions and so are several steps removed from how a MapReduce cluster may be used in practice.

Also, the variety of situations that can be tested are more constrained than a model/simulation, where a 10,000 node cluster can be tested by simply changing a variable. Instead, the research will be constrained by the available equipment meaning

it would be impossible to replicate some extreme configurations (e.g. large clusters) which can provide a better test of a theory (Sharp et al., 2002).

In summary, the choice was between modelling and prototyping. Both techniques would be able to provide an indication of the scale of the performance improvement, but the empirical and non-simplified nature of prototyping provides a more authoritative benchmark. Likewise, both techniques would be able to test a wide variety of circumstances, although more so with modelling as it does not rely upon physical equipment, particularly a large cluster, being available. However, only prototyping could have evaluated SCFS against the qualitative criteria sufficiently well to address the research question. Prototyping was therefore the chosen method.

The prototype required can be best classified as an experimental prototype (Floyd, 1984) as opposed to exploratory or evolutionary. These other types are typically used to help establish user requirements or developing an existing product to meet changing requirements, which are not required in this situation. Unlike "Wizard of Oz" prototypes, which are primarily used as a tool in user interaction design, this prototype needed to be functional but not necessarily fully featured (Dernsen et al., 1994). The prototype system, described in Section 3.3, which extends Hadoop, therefore has a number of short-comings that a more elaborate implementation might resolve.

### 3.3 Research Plan

When conducting controlled experiments Rugg and Petre (2007) specified several over-arching steps that should be followed. These steps, shown in Figure 13 (adapted for non-human participants), were the basis for the methodology employed.

Step 1: Design the experiment and develop the environment  
Step 2: Conduct a small-scale pilot experiment  
Step 3/4: Plan larger scale experiments and conduct them  
Step 5: Analyse the data gathered from the main experiments  
Step 6/7: Write up results and conduct follow up experiments if appropriate

*Adapted from Rugg and Petre (2007)*

**Figure 13: Steps for Conducting Controlled Experiments**

This section concentrates on Step 1, the design of the experiments and the development of the baseline and prototype environments. Step 2 is not formally documented since there was no single pilot experiment but rather smaller iterative experiments forming part of the testing stage. Each iteration revealed minor software bugs or methods to improve the process that were fixed in the main experiments.

The details of the main experimental execution of steps 3-4 are documented in Chapter 4. Steps 5-7 are conducted in Chapters 5 and 6, which present the experimental results and project conclusions.

#### 3.3.1 Experiment Design

Each experiment is designed to simulate a scenario that would provide some insight into whether the hypotheses in Section 2.4 are accurate. Hypothesis 1 will be tested by

varying the number of simultaneous jobs, hypothesis 2 by introducing jobs with different speeds and hypothesis 3 by varying the block size used within the cluster.

Unfortunately it was not possible to test hypothesis 4 since the equipment available did not include solid state media. This demonstrates a disadvantage of prototyping discussed previously. The circumstances that can be examined are limited compared to the modelling approach because of the resource constraints of the project. This hypothesis will need to be examined in the future if it is still considered of importance.

The precise details of the cluster configuration and job schedule for each experiment are provided in Chapter 4. The cluster consisted of three machines with no use of virtualisation, consistent with recommendation 1 from section 2.2.

When conducting experiments it is necessary to decide what jobs to execute and what data to parse. A standardised benchmark, such as the TeraSort, would allow better comparison between the improvements (recommendation 2). Unfortunately the existing TeraSort application is not compatible with SCFS, since it relies upon the use of line breaks to separate records. This is not supported by the initial prototype (see the implementation details in Section 3.3.2). Furthermore, it would not have been possible to conduct the TeraSort as the total DFS space available is around 150GB.

Another available benchmark is MRBench (Kim et al., 2008), the MapReduce adaptation of TPC-H. Unfortunately, it was not feasible to use this benchmark due to its complexity. As it consists of 22 different queries, many of which are complex, it

would have taken too much time to amend these MapReduce jobs so that they are compatible with SCFS. The relatively immature prototype would need to be more thoroughly trialled before it is capable of executing MRBench.

The third recommendation is that an enhancement should be tested with different sets of jobs, including some that are I/O intensive and some that are CPU intensive. This is to ensure that any performance improvement quoted is likely to be experienced in practice, as well as in the laboratory. Unfortunately, executing multiple sets of different jobs would simply not have been possible in the time available.

For these reasons a single bespoke MapReduce application, the 'ProjectSummariser', was written to conduct experiments with SCFS. The ProjectSummariser can process large volumes of data and conduct a moderate degree of processing and is presented in detail in Chapter 4. Therefore these experiments will not include a standard benchmark job or a diverse set of I/O and CPU-bound jobs and so do not adhere fully to the recommended experimental technique.

This application analyses the number of page-views from data supplied by 'Wikistats' (Wikimedia, 2012), which logs accesses made to Wikipedia. An example of this data is included in Figure 14. During this particular hour the "Modern Language Association" article from the "en" (English) project was viewed 20 times, transferring a total of 363KB. The ProjectSummariser parses this data and totals the number of downloads and data transferred for each project. The source code is included in Appendix B.

en Modern_Kitchen	1	529
en Modern_Korean	1	9156
en Modern_Language_Association	20	363479
en Modern_Language_Bible	1	7961

**Figure 14: Sample Input Data**

It is also necessary to decide what metrics will be recorded. Recommendation 1 was that performance should be measured by recording how long a cluster takes to perform a defined set of jobs. This was therefore the key measurement for these experiments.

Several ways of recording this were identified in Section 2.2.1, including 'synthetic micro-benchmarks' (Moise et al., 2010) with a customised DFS client. This approach could not be justified as considerable time would be required to develop a custom DFS client and Hadoop's existing capabilities already support the collection of the chosen metric. One could also question the reliability of a customised client compared to Hadoop, which has (theoretically, at least) been subject to wider scrutiny.

It is also important to consider the sources of error or bias and possible mitigations. The start/finish times recorded by Hadoop would be very accurate information with little bias or error. One potential source of error was that the prototype and baseline systems committed different volumes of logging information to the disk. As the performance of the underlying storage device is critical to the experiments, any significant disk activity would interfere with the performance of the system being measured. The prototype and baseline systems therefore have almost identical volumes of logging to ensure one system did not have an inherent advantage over the other.

Some potential for random error was difficult to remove. For example, the Java Virtual Machine may garbage-collect whilst the prototype is running but not during the equivalent baseline experiment. There could also be considerable variation in a job's performance between subsequent executions due to the non-deterministic and concurrent nature of distributed processing. This was mitigated by taking several measurements and using the average for analysis (Sharp et al., 2002). This approach was vindicated by the variability of some results between different runs. For instance, the largest Reduce duration in one experiment (Experiment 1, 5x simultaneous jobs for the baseline system) varied from 41 seconds in one repetition to 63 seconds in another 'identical' repetition.

One major bias was caused by utilising a cluster that is unrepresentative of those used in practice. The test cluster consisted of two slaves and one master. A real cluster would typically contain hundreds or thousands of slaves. Also, the types of job and nature of the input data may also be unrepresentative of the real-life use of a MapReduce cluster. However, it was infeasible to address these concerns with the limited resources available, so one must be wary of inferring universal applicability of any conclusions based solely on this research. This research only claims to benchmark SCFS for the limited set of jobs, input data and the low-specification cluster used.

### **3.3.2 Prototype Development**

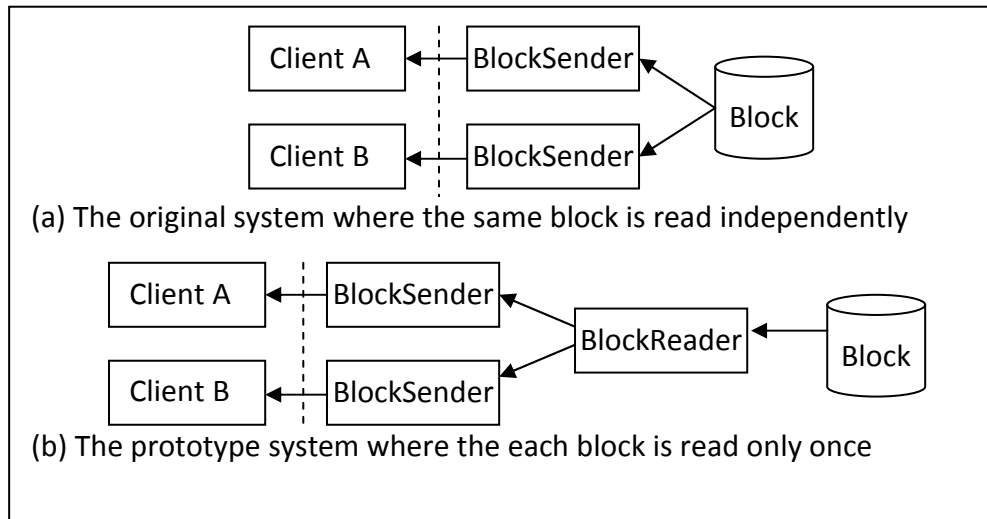
As well as designing the experiments, Step 1 also involved developing two environments, the baseline and prototype. The baseline consists of an unmodified

MapReduce framework used to generate control data. This was necessary because although baseline performance measurements were available through other researchers, the circumstances under which they were made would not have been feasible to reproduce. This variation in experimental conditions of MapReduce research was a key theme of the literature review in section 2.2.

The prototype is a MapReduce framework supporting SCFS. As it would be infeasible to have created a new MapReduce implementation, an existing one was modified. Hadoop was chosen because it is a mature product distributed under the Apache License, allowing modifications. There is also considerable documentation (e.g. Lam, 2011) and it utilises a mainstream platform (Java/Linux). No other MapReduce implementation, including those listed in Section 2.1, satisfied these criteria.

The prototype was developed by modifying Hadoop's 'BlockSender' class, which is responsible for accessing local files and presenting them to DFS clients. Since HDFS is block-centric (Shvachko et al., 2010), the system is configured on a per-block basis. Upon start-up the prototype reads a file which specifies a list of blocks and the access method to be used (baseline or SCFS). The baseline system, illustrated in Figure 15(a), allows BlockSenders to read a block independently, regardless of whether other BlockSenders are accessing it. SCFS was implemented by inserting a new component (BlockReader) between the BlockSender and the local file system, as in Figure 15(b). If a new BlockSender needs to read a block using SCFS it registers with an existing BlockReader or instantiates a new one should one not already exist. After registering, the BlockSender begins to receive data from the BlockReader to pass to its client.





**Figure 15: The SCFS MapReduce Prototype**

The version of Hadoop/HDFS that was modified (0.20.2) supports two modes of transferring data, streaming and non-streaming. Streaming pipes data from the file directly to the client's socket connection. This optimisation sacrifices CRC checks on data. The non-streaming approach scans the file in stages, checks the relevant CRC values from a separate metadata file and then sends the data to the output stream. The comparison being made is with the non-streamed approach. This means that both systems conduct CRC checking and most other underlying tasks in a similar manner.

It is worth noting some limitations of the prototype. It can only handle files that are exact multiples of 512 bytes. This was to simplify the BlockSender, which tracks how much of a block the client has consumed so far. Also, there is no capability to recognise logical breaks in files such as XML element termination. Instead, data is simply presented to new clients from the point in the file that the scan has reached when the new client registers. Neither limitation is insurmountable but they were both necessary to deliver the prototype within the available time.

### **3.4 Summary**

Prototyping has been chosen because it is able to provide an indication of the performance improvement that SCFS can provide across a range of circumstances. Whilst this is possible with modelling, the practical experience will help evaluate SCFS against many of the qualitative criteria established.

The SCFS prototype was employed in different experiments which map to three of the four hypotheses drawn from the literature review. This includes comparing the performance of the baseline and prototype systems whilst varying the number of concurrent jobs, the speed of jobs and the size of the input data. Unfortunately it was not possible to investigate hypotheses 4 (impact of solid state media) with the resources available.

## **Chapter 4 Data Collection**

Data was collected by submitting jobs to a cluster under various conditions and recording completion times. This chapter documents the procedures and configuration used so that the results can be understood and that the experiments are repeatable.

Section 4.1 documents the hardware and configuration employed across all experiments. Section 4.2 documents specific conditions for each experiment.

### **4.1 Cluster Configuration**

A cluster of 3 machines was utilised. Each node was a Dell Optiplex GX520 (3GHz CPU, 1GB RAM, 80GB HDD), running Fedora 14 and an ext4 local file system. One node was the master (running a NameNode and JobTracker) and the rest slaves (each running a DataNode and TaskTracker). No SecondaryNameNode was configured as little resilience benefit would be achieved.

The replication factor, the degree to which blocks are replicated, was set to 1. The default for HDFS is 3, but in such a small cluster would result in no network-based transfer and reduce the number of simultaneous scans/SCFS clients as each node would possess a local copy of the data.

## **4.2 Experimental Configuration**

Three experiments, each explained in more detail in the following sub-sections, were conducted. Each trial consisted of three runs, with the average completion time being recorded.

Where possible, prototype experiments were kept identical to the equivalent baseline experiments. The same data blocks were read from the same nodes. When experiments used different files (necessary when varying the file size), this was achieved by uploading the new files multiple times until they were hosted on the desired node. Care was taken not to disturb the execution of the experiments - for example, by frequently refreshing the status web page. The only exception was in extremely long running experiments where system information, particularly memory usage, was queried to determine why jobs were taking so long.

### **4.2.1 Experiment 1: Varying Degrees of Concurrency**

This experiment investigates hypothesis 1 - that increasing the number of concurrent clients accessing the same file will improve the relative performance of SCFS.

A 64MB file was uploaded to the DFS and was stored on Slave 1. For each trial, a number of simultaneous ProjectSummariser jobs that read this file were executed. The number of jobs submitted ranged from 1 to 12. The number of simultaneous Mappers and Reducers allowed on each node was increased at each stage. These values were

set at half of the number of simultaneous jobs for each of the two nodes ensuring that all jobs are executed simultaneously with the processing being fairly distributed.

#### **4.2.2 Experiment 2: Slow Running Jobs**

This experiment is designed to investigate hypothesis 2 - that slow running jobs can degrade the performance of SCFS.

The file from Experiment 1 was also used here. Two different sets of jobs were executed. Set A consisted entirely of ProjectSummariser jobs whilst Set B consisted of ProjectSummariser jobs with one SlowProjectSummariser. A SlowProjectSummariser conducts exactly the same work as the ProjectSummariser but contains a small pause between each record to make it progress more slowly.

In all cases the total number of simultaneous jobs submitted was eight, chosen because the prototype out-performed the baseline system when running this number of concurrent jobs. This would clearly show the impact of a slow running job if the baseline system were to out-perform the prototype. All jobs required access to the same file/scan and were submitted simultaneously.

#### **4.2.3 Experiment 3: Varying input file sizes**

This experiment is designed to test hypothesis 3 - that as the input size increases the performance of SCFS will increase relative to the baseline system.

In this experiment, the number of simultaneous jobs was kept the same and only one type of job (the ProjectSummariser) was used. However, the size of the file accessed by the jobs increased in increments of 64MB up to 448MB. This was achieved by increasing the block size, reformatting the DFS and uploading a single file of that size. The number of concurrent jobs was fixed at 2. To attempt to force the disk to access multiple locations simultaneously, the second job was submitted halfway through the Map phase of the first job. The delay to use was determined by executing one instance of the job and inspecting how long it took for the Map phase to complete and halving this time.

Unfortunately, the timed offset of submitting the second job was decided arbitrarily to be half the time of a single execution as there is no theoretical model to guide these decisions. This illustrates another disadvantage of prototyping - a theoretical model may need to be in place to guide decisions when designing and conducting experiments.

### **4.3 Summary**

This section outlined the three experiments used to test three of the proposed hypotheses. Experiment 1 varied the number of concurrent scans, experiment 2 introduced a slow running scan and experiment 3 varied the size of the input file.

## **Chapter 5 Results**

This chapter presents the results from the experiments with sections 5.1 to 5.3 analysing data from each experiment. Section 5.4 compares the results to the performance improvements achieved by the other enhancements and assesses SCFS against the qualitative criteria established in the literature review. Section 5.5 examines to what extent the hypotheses and research question have been addressed and section 5.6 evaluates the primary data collection process.

### **5.1 Impact of Increasing Client Numbers**

The average completion times for Experiment 1 are shown in Table 1. Completion times for the Map and Reduce stages are shown separately, along with the total time. The difference between the sum of the Map and Reduce times and the total is due to the framework conducting tasks such as shuffling data between phases.

As more concurrent jobs are executing, average completion time increases for both systems. The increases in duration remain steady until 11 concurrent jobs, at which point a very substantial increase is observed. For example, the Reduce time for the baseline system ranged from 20 to 22 seconds between 6 and 10 concurrent jobs but increased by 16x when an 11th job was introduced. It was observed that all available RAM was utilised at this point causing the nodes to progress tasks very slowly. This illustrates a limitation of prototyping in that any conclusions drawn from these experiments will be limited to 10 concurrent scans.

Clients	SCFS Prototype Times (s)			Baseline Times (s)		
	Map	Reduce	Total	Map	Reduce	Total
1	19	12	44	18	12	43
2	23	12	46	19	12	44
3	43	18	74	34	15	62
4	51	20	87	37	19	74
5	101	22	163	78	19	138
6	128	22	170	100	20	139
7	160	22	220	175	21	203
8	185	23	237	222	21	304
9	233	26	303	257	22	351
10	377	31	450	337	22	439
11	2203	512	3012	1836	368	2501
12	4722	1101	6501	3984	702	5375

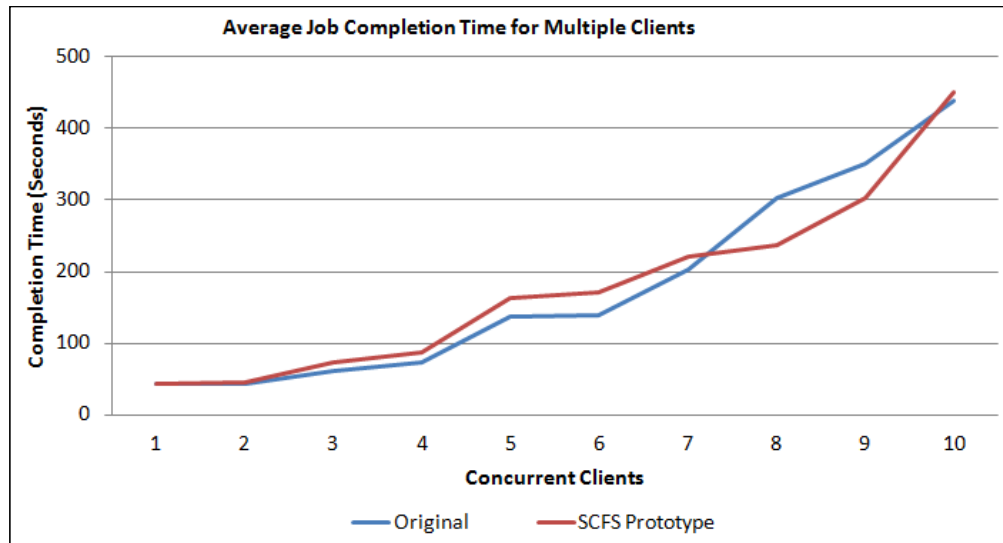
**Table 1: Experiment 1 Results (Varying Concurrent Clients)**

Figure 16 shows a graph for the average completion times, excluding the outlying results for 11/12 concurrent jobs. When there is only one job, the performance of each system was similar, with the prototype being just 1s slower (2%). This is expected as both systems behave similarly when serving one client. The performance difference as the number of jobs increases is more pronounced with the prototype being 31s (18%) slower when there are 6 concurrent jobs.

Beyond 6 concurrent jobs the performance of the prototype relative to the baseline improves. When there are 7 jobs the prototype is only 17s (8%) slower than the baseline and actually outperforms it for 8 and 9 concurrent jobs, being 67s (22%) and 48s (14%) faster. However, for 10+ jobs the baseline outperforms the prototype. It is also interesting to note the Reduce stage consistently takes longer for the prototype. This can be explained by SCFS regulating the speed of the Map tasks so that they finish



closer together. This causes more resource contention during the Reduce stage than in the baseline system where some jobs raced ahead and finished earlier.



**Figure 16: Average Job Completion Time (Experiment 1)**

This experiment has shown a critical point where SCFS is beneficial - between 8 and 9 jobs for this particular cluster and set of jobs. SCFS requires additional computation for tracking scan progress and distributing data which incurs an additional performance cost. Below this critical point, this additional cost outweighs the overhead incurred from conducting multiple concurrent scans of the same underlying media. The limited hardware resources prevented investigation of how SCFS performed beyond 10 jobs.

## 5.2 Impact of Slow Running Jobs

The average completion times for Experiment 2 are shown in Table 2. Each trial consisted of eight concurrent jobs, with Set A consisting entirely of fast jobs and Set B consisting of seven fast jobs and one slow job.

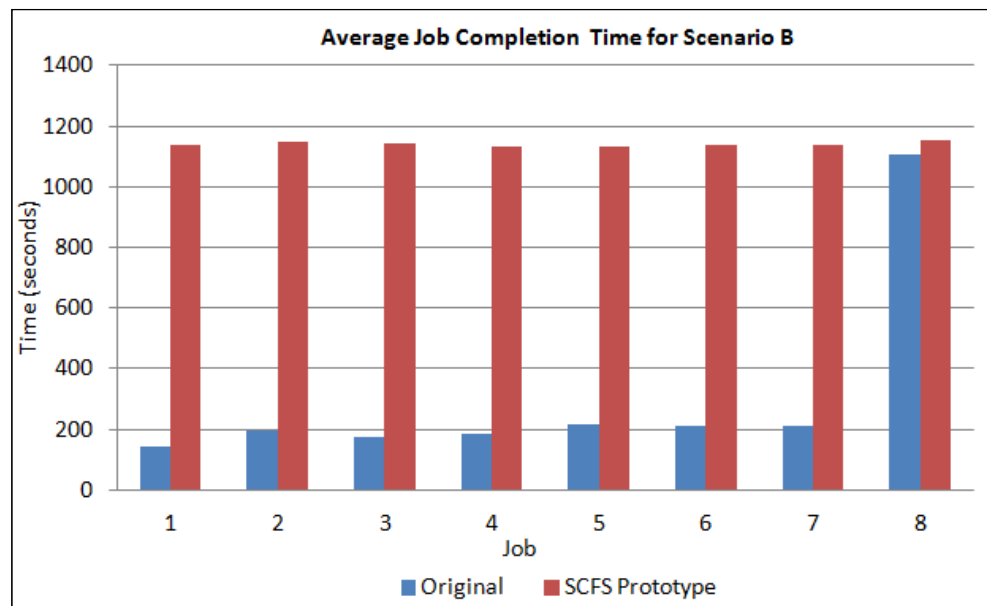
		SCFS Prototype Times (s)			Baseline Times (s)		
Job Set	Stage	Slow Job	Fast Jobs	All Jobs	Slow Job	Fast Jobs	All Jobs
A	Map	n/a	185	185	n/a	222	222
	Reduce	n/a	23	23	n/a	21	21
	Total	n/a	237	237	n/a	304	304
B	Map	1090	1085	1086	1075	136	254
	Reduce	32	27	28	12	27	25
	Total	1153	1138	1140	1106	233	306

**Table 2: Experiment 2 Results (Introducing a Slow Job)**

These figures show that the prototype out-performed the baseline system when eight identical fast jobs were submitted, with the prototype taking an average of 237s to complete each job and the baseline system 304s. However, this was not the case when a slow-running job was present. The average completion time for the baseline system increased marginally (304s to 306s), whilst average completion time for the prototype increased by nearly 5x (237s to 1140s).

Why the observed times in the baseline system were not larger may be explained by how the slow job was made. Inserting a sleep into the processing meant that there was one less thread competing for resources during this time. This allows the other jobs to progress more quickly, thus limiting the increase in the average completion time.

Figure 11 shows a graph that breaks down the overall completion time for each job, averaged across the jobs in Set B, where Job 8 is the slow job. Under the prototype system, all jobs took approximately 1100s whilst for the baseline system jobs there was much greater variation with the slow job taking approximately 1100 seconds and the fast jobs 230s to complete.



**Figure 17: Average Job Completion Time (Experiment 2)**

This shows that slow running jobs can have a significant negative impact on SCFS. In the prototype system, fast jobs were constrained by the slow job that they shared the scan with.

### 5.3 Impact of Increasing File Sizes

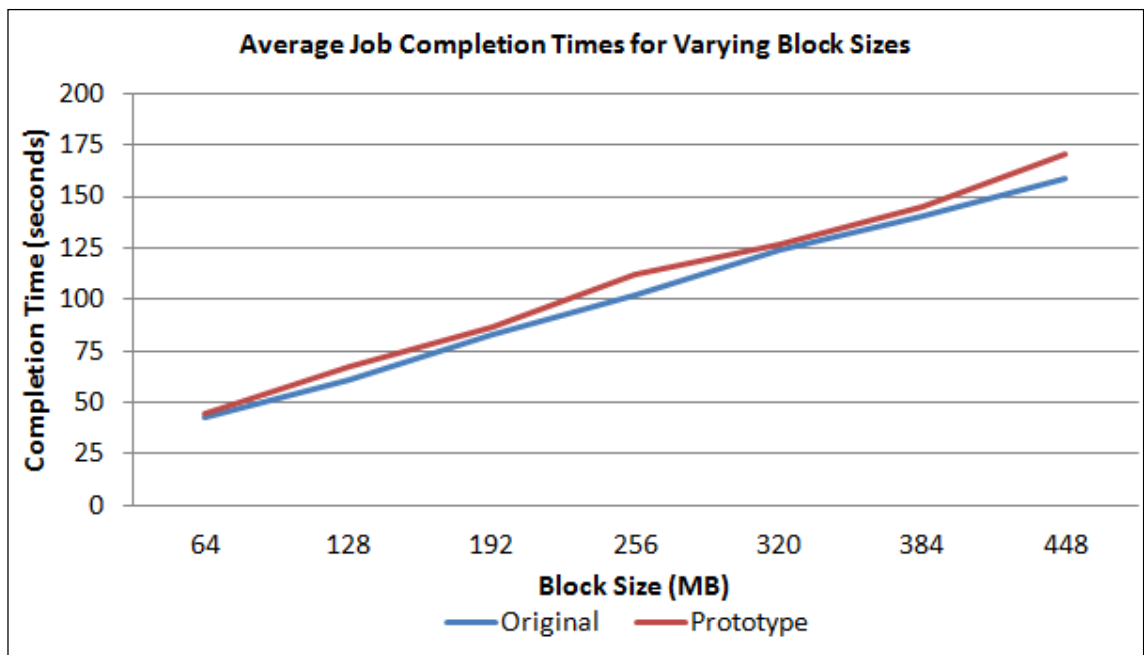
The average completion times for Experiment 3 are shown in Table 3. They show, unsurprisingly, that the time taken increases as the input file size increases, ranging from 43s for an input size of 64MB to approximately 160s for 448MB.

	SCFS Prototype Times (s)			Baseline Times (s)		
Input Size (MB)	Map	Reduce	Total	Map	Reduce	Total
64	19	12	44	18	12	43
128	39	15	67	35	15	62
192	61	15	87	54	16	83
256	82	18	111	72	18	102
320	98	19	127	92	19	124
384	121	20	145	110	19	141
448	143	20	171	128	20	159

**Table 3: Experiment 3 Results (Varying Input Size)**

The vast majority of the increase comes from an increased Map duration, which incurs an increase roughly proportional to the input size. For each 64MB increment, the prototype takes an additional 20s, whilst the baseline system takes an additional 18s. Figure 18 shows a graph of the overall completion time. When there are only two concurrent jobs, the performance of the prototype system never exceeds the baseline system for any size of input.

This experiment has shown that whilst average job completion time increases with large input sets, with only two concurrent jobs requesting different parts of the same file, the prototype performs consistently worse. Like experiment 1, this outcome may be dependent upon other factors, such as cluster specification so it is not possible to reach generalised conclusions from this data.



**Figure 18: Average Job Completion Time (Experiment 3)**

#### 5.4 Evaluation of SCFS

The best-case improvement was a 22% reduction in job completion time with most circumstances yielding worse performance. This does not compare favourably to the other enhancements examined. The best-case improvements for other enhancements were more significant and degradation rare. For example, the MapJoinReduce model amendments typically halved completion times for its jobs (Jiang et al., 2010) and the distributed memory cache also achieved this degree of improvement in its best-case scenario (Zhang et al., 2009).

However, as shown in section 2.2, it is difficult to make any meaningful quantitative comparison. Section 2.3 therefore established several qualitative criteria as a means of evaluating the quality of a MapReduce enhancement. It is now possible to assess SCFS against these criteria:

1. Configuration: Unfortunately, SCFS introduces additional configuration as it is necessary to specify what blocks should be read with SCFS. This is necessary because some files must be read strictly linearly. However, this is a relatively straightforward binary option rather than a numerical value where gauging the optimal setting is difficult, such as 'io.sort.percent', highlighted by Babu (2010).
2. Compatibility: Although SCFS does not prevent any existing MapReduce applications from running, jobs requiring strict ordering of data will not be compatible (as discussed in section 2.4.1).
3. Scalability in MapReduce is achieved by the ability to easily expand the number of nodes. Whilst there is no intuitive reason why SCFS would limit this, none of the experiments demonstrated this. As indicated in Section 3.2, a modelled approach may have been better placed to investigate this criterion given the limited equipment available.
4. Simplicity: There is additional work for the developer, who must now account for line records being truncated depending on when the job joins the scan. However, this is a relatively simple amendment requiring only a few lines of Java. It is significantly less complicated than the stage barrier reduction enhancement (Verma et al., 2010). However, unlike SCFS, many enhancements were introduced transparently without burdening the developer with any additional complexity, particularly the scheduling enhancements like SAMR (Chen et al., 2010).

5. Scope: Since many MapReduce applications are I/O-bound (Agrawal et al, 2010), many jobs potentially stand to benefit from SCFS. Although it is believed that CPU-intensive jobs would not benefit from SCFS, the limited time available restricted the types of jobs that were examined.

Overall, SCFS does not compare favourably against these criteria. The major failure is compatibility with existing jobs since some applications requiring data to be ordered is not supported by SCFS with minor flaws identified against configuration and complexity.

### **5.5 Addressing the Hypotheses and Research Question**

Hypothesis 1 predicted that as the number of concurrent jobs utilising the same file increases, the performance of SCFS relative to the baseline system will improve. This was the case with the performance of the prototype overtaking the baseline with 8 concurrent jobs. However, as the number of concurrent jobs increased they exceeded the hardware's ability to service them. If the limitations of equipment, specifically the amount of RAM, had not been reached then it is possible the performance benefit could have continued beyond the 9 concurrent jobs observed.

Hypothesis 2 predicted that the overall performance of MapReduce under SCFS will be hindered by slow running jobs that share a scan. This proved to be the case with the baseline completing the fast-running jobs more quickly than the prototype and the slow-running job in around the same time. This is consistent with the disadvantages of the elevator policy (Zukowski et al., 2009) identified in section 2.4.2.

Hypothesis 3 predicted that as the input volume grows, the detrimental impact of increased seek times to different parts of the disk would cause the relative performance of SCFS to increase. This was not shown to be the case whilst there were only two concurrent scans. Insufficient data was collected to answer whether a higher number of concurrent scans, and therefore more seek operations, would increase the sensitivity to file size. This could be investigated if more time were available.

Unfortunately, it was not possible to evaluate hypothesis 4 with the available resources.

This original question that motivated this research was:

*To what extent can shared cyclical file scanning improve the performance of concurrently executing jobs within the MapReduce distributed computing framework?*

There are some situations where it is possible for SCFS to improve the performance but improvement is far from universal. As the number of concurrent scans of the same file increases, SCFS becomes more beneficial and eventually outperforms the non-SCFS system. However, the point where this happens is likely to be cluster-specific as well as depend on the type of job and input data.

The vast majority of experiments showed that SCFS actually performed worse than the non-SCFS system. A single slow-running job destroyed any benefit provided by SCFS causing its use to be detrimental to performance. Furthermore, this enhancement



requires additional cluster configuration, addition complexity in the Map code and is incompatible with existing applications that rely upon the ordering of input data.

## **5.6 Research Evaluation**

The prototype implementation was satisfactory and contained sufficient functionality to generate the required metrics. Changes to address the short comings identified in section 3.3 could have ensured that the functionality being compared was exactly like-for-like. This includes input files having to be a multiple of 512 bytes and lines being split if the scan is joined whilst it is part-way through reading it. The systems being compared do not currently exhibit identical behaviour, making comparisons slightly less valid.

Despite these limitations the experiments themselves progressed largely as planned. Reliance exclusively upon job completion times may have led to unnecessary speculation into the underlying causes of the observed behaviour. This could have been improved by recording cumulative progress of each job, enabling a more comprehensive analysis of which jobs are actively progressing and revealing bottlenecks that SCFS contributes to or relieves. A richer explanation of the internal workings of SCFS could be provided by employing the synthetic benchmark technique, described in section 2.2.1. A customised client could record more precise behaviour than the unmodified Hadoop client.

Whilst some of this additional information can be recorded by Hadoop, it can lack sufficient granularity until the logging verbosity is increased. However, a principle

established when justifying the method in section 3.2 was that the system being measured should not be interfered with. This would happen if verbose measurements were recorded as this data would be written to the device being used to read data, causing it to have a higher workload.

With hindsight, a better balance could have been achieved, such as generating selective output of system status with a periodic 'snapshot'. However, this would have required additional development time which was not available.

Another problem with the experimentation was that two of the four experimental recommendations in section 2.2, relating to the choice of jobs and employing standard benchmarks, were not adhered to. Although this may limit the ability to generalise results, it was still possible to address the hypotheses with the caveat that the conclusions are limited to the specific cluster and job types used.

Experiments 1 and 2 led to specific conclusions for their respective hypotheses, albeit not generalised due to the experimental constraints. However, this was lacking with Experiment 3. The decision to introduce the second job halfway through the first job's Map phase was arbitrary and a better understanding of an underlying theoretical model could have produced a better approach or at least justified the chosen one. Also, Experiment 3 may have led to a more interesting conclusion if it was repeated for different numbers of concurrent jobs. More experimentation could resolve this.

## 5.7 Summary

SCFS can improve the performance of MapReduce in some situations, specifically as the number of concurrent jobs increases, but a single slow-running job sharing a scan will significantly degrade performance. Unfortunately, most scenarios saw a performance degradation and SCFS had several flaws when compared to the qualitative criteria.

Whilst there were no major problems with the prototype and experiments, several improvements were identified that could improve the richness of the data collected and the comparability of the baseline and prototype systems.

## Chapter 6 Conclusions

The circumstances where SCFS demonstrated a performance improvement were limited. Performance was only improved relative to the baseline system when there were a significant number of concurrent jobs accessing the same file. The best-case 22% reduction in job completion time from SCFS was significantly less than the best cases from other enhancements reviewed in Chapter 2. Performance degradations in benchmarks published for other enhancements were relatively rare. However, for SCFS degradation was encountered more often than improvement. For example, the presence of slow jobs significantly degrades the performance of SCFS.

Furthermore, SCFS introduces some backwards incapability and burdens the cluster administrator and application developer with additional work. This is contrary to the founding principles of MapReduce, which aims to simplify and generalise distributed computing (Yang et al., 2007).

However, it is premature to dismiss SCFS as a means of improving MapReduce. Even though it may be inappropriate for mainstream use, there may be some users who could benefit. For this reason, the prototype has been made available separately to the main Hadoop release (see appendix B), allowing users to utilise it and develop it further.

## **6.1 Project Review**

The original aim of this project was to evaluate SCFS in the context of the wider body of knowledge about MapReduce and related topics and to demonstrate whether, and in what circumstances, SCFS can yield a performance improvement. This section looks back over the project to determine what went well and what could be improved.

Unfortunately, resource constraints prevented a full taxonomy of factors being investigated. Only a small range of circumstances were investigated. Factors not investigated include the use of larger or higher specification clusters to investigate the impact on scalability and the use of solid state media. Conclusions are therefore limited to the specific cluster, jobs and input data used. Further research would be required to determine the impact of these factors.

The project itself progressed relatively smoothly. When conducting the literature review, it became clear that it would be very difficult to objectively compare SCFS's performance to existing enhancements. There was a huge diversity in jobs, input data, cluster size and specification as well as framework configuration. The secondary research therefore concentrated on establishing qualitative criteria by which an enhancement can be assessed. These criteria became just as important as the numerical measurement of the performance improvement and can be re-used to assess future MapReduce enhancements.

Prototyping proved to be the most appropriate technique, producing tangible evidence demonstrating that SCFS has potential to improve the performance of MapReduce and

allowing practical experience to aid the assessment of the qualitative criteria. The main disadvantage of this approach was the limited the range of scenarios that could be replicated due to the resource constraints of the cluster. Even slightly higher specification nodes, particularly more powerful CPUs and more RAM, would have enabled a wider range of circumstances to be investigated.

The time allocated to prototype development was exceeded by a month. This was partly because of Hadoop's code base was being somewhat cryptic with unexplained variables and unpopulated JavaDoc descriptions. This was despite Hadoop being chosen due to the support and documentation available. Nevertheless, the prototype was of higher quality than envisaged, vindicating the highly iterative development methodology used.

The practical experimentation was guided solely by the secondary evidence, with some added intuition. A better approach may have been to blend theoretical and prototype work together by using modelling to better guide the experiment design, consistent with the approach by Dhok et al. (2010) and others. This may have, for example, removed the need for the arbitrary decisions that guided the design of Experiment 3.

Despite the limitations of this research, the key objective of contributing to the body of knowledge about MapReduce and shared scanning has been achieved. Furthermore, the SCFS prototype and the criteria to assess MapReduce enhancements have been made available for other researchers to utilise in the future.

## 6.2 Future Research

Work that continues this research encompasses four main areas - development of a model, expanding the practical experimentation, addressing the poor performance of SCFS and further developing criteria to assess MapReduce enhancements in general.

The project review indicated that the experimentation could be improved if a theoretical model had been developed to predict which situations will improve/degrade performance. The prototype could be used to verify these predictions empirically. This would be a non-trivial endeavour but would produce an invaluable guide to hone future practical experiments. It could also be used as a resource in its own right to rapidly test more extreme circumstances, such as large clusters, that are infeasible with just a prototype (Sharp, 2002). A starting point could be to amend the non-cyclical shared scanning model developed by Agrawal et al. (2008). Having a model in place would allow more generalised conclusions rather than being specific to the cluster used.

Even without a model, the experimentation could be extended as many variables, or combinations of variables, were not explored fully or even at all. The lack of conclusions regarding Hypothesis 3 could be addressed by varying both the number of simultaneous clients (as per experiment 1) and the size of the input data (as per experiment 3). Hypothesis 4 could be investigated by executing the same experiments but with SSDs installed in the nodes responsible for scanning files.

None of the experiments assessed the impact of SCFS on scalability, despite this being a key criterion established. This could be investigated by growing the number of nodes for each system and attempting to determine if any pattern can be established. When investigating the distributed memory cache Zhang et al. (2009) varied the number of slave nodes from 1 to 6 and observed the behaviour. This demonstrates that investigation into scalability, albeit limited, is possible without a large cluster. The presence of a model would also help this investigation.

Another problem with the experimentation is the lack of comparability with other enhancements. Methodology recommendation 2 in section 2.2, concerning the use of standard benchmarks, would have helped this but was not followed. This could be resolved by conducting further development so that the prototype can run the TeraSort (Nyberg et al., 2012) and MRShare (Kim et al., 2008) applications to benchmark its performance. The performance impact of SCFS can then be more objectively compared to enhancements that published benchmarks against these benchmarks, such as Babu (2010) and Tian et al. (2009) who utilised the TeraSort.

Recommendation 3 was also not followed as only one type of job was run rather than a mixture of I/O and CPU-bound jobs. This means that the improvement figures quoted for SCFS may not be representative for the range of jobs that will be encountered in practice. This could be rectified relatively easily with the existing prototype (which is publically available) by simply implementing some additional MapReduce jobs. This would give a more robust answer to the research question.



The third area for future work attempts to improve SCFS's relatively poor performance. General performance could be improved by utilising Hadoop's streaming capabilities or reducing the state required to manage the shared scans. These improvements could increase the range of circumstances where the prototype outperforms the baseline system.

This research also illustrated a specific circumstance that led to SCFS exhibiting particularly poor performance - the presence of slow running jobs. A possible solution is to categorise jobs according to their demands, similar to the approach of Tian et al. (2009). By assigning jobs to a 'fast' group or 'slow' group, for example, and having one shared scan per group, the impact of the slow jobs would be reduced.

The approach taken by Dhok et al. (2010) of rejecting or deferring jobs that would hurt the performance of other jobs could also be utilised. Performance could be improved in some circumstances by deferring slow jobs when faster jobs are already executing. Both of these approaches would involve investigating how the speed of a job could be determined whilst it is executing or even predicted before it starts.

The final area for future work does not specifically concern SCFS evaluating MapReduce enhancements more generally. At present, the means of assessing the qualitative aspects of an enhancement is simply a list of criteria and lacks a formal methodology. Just as there are standard benchmarks available for measuring performance, such as TeraSort (Nyberg et al., 2012) and MRBench (Kim et al., 2008), this list could be the basis for standardising the qualitative evaluation of MapReduce

enhancements. This could involve developing a means of scoring enhancements, for example weighting violations of scalability more seriously than other flaws.

This would be an improvement on the currently disjointed approach to assessing the quality of MapReduce enhancements. Zhang et al. (2009) assessed the distributed memory cache enhancement against scalability and Verma et al. (2010) referred to the additional complexity introduced when the stage barrier was dismantled. However, few, if any, researchers comprehensively consider and document the wider range of criteria that constitute a high quality enhancement.

Performance metrics alone will rarely be truly comparable and do not reflect the wider impact that an enhancement has upon the MapReduce framework. A standardised mechanism of evaluating enhancements qualitatively will be a valuable resource to the wider MapReduce community.

## References

- Aboulnaga, A., Wang, Z. and Zhang, Z.Y. (2009) 'Packing the Most onto Your Cloud', *Proceeding of the First International Workshop on Cloud Data Management*, Hong Kong, China, New York, NY, USA, ACM, pp. 25-28.
- Agrawal, P., Kifer, D. and Olston, C. (2008) 'Scheduling Shared Scans of Large Data Files', *Proc.VLDB Endow.*, vol. 1, no. 1, pp. 958-969.
- An, M., Wang, Y. and Wang, W. (2010) 'Using Index in the MapReduce Framework', *Web Conference (APWEB), 2010 12th International Asia-Pacific*, pp. 52-58.
- Babu, S. (2010) 'Towards Automatic Optimization of MapReduce Programs', *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, New York, NY, USA, ACM, pp. 137-142.
- Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W. and Ousterhout, J.K. (1991) 'Measurements of a Distributed File System', *SIGOPS Oper.Syst.Rev.*, vol. 25, no. 5, pp. 198-212.
- Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003) 'Xen and the art of virtualization', *SIGOPS Oper.Syst.Rev.*, vol. 37, no. 5, pp. 164-177.
- Callaghan, B. Pawlowski, B. Staubach, P (1995) 'RFC 1813: NFS Version 3 Protocol Specification', Internet Engineering Task Force RFCs.
- Chen, Q., Zhang, D., Guo, M., Deng, Q. and Guo, S. (2010) 'SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment', *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 2736-2743.
- Dadan, Z., Zhebing, C., Jianpu, W., Minqi, Z. and Aoying, Z. (2009) 'Different File Systems Data Access Support on MapReduce', *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pp. 1-4.
- Dean, J. and Ghemawat, S. (2008) 'Mapreduce: Simplified Data Processing on Large Clusters', *Communications of the ACM*, vol. 51, no. 1, pp. 107-113.

Dernsen, N., Dybkjaer, H. and Dybkaer, L. (1994) 'Wizard of Oz Prototyping: When and How?', *CCI Working Papers in Cognitive Science and HCI, WPCS-94-1*.

Dhok, J., Maheshwari, N. and Varma, V. (2010) 'Learning Based Opportunistic Admission Control Algorithm for MapReduce as a Service', *Proceedings of the 3rd India software engineering conference*, Mysore, India, New York, NY, USA, ACM, pp. 153-160.

Floyd, C. (1984) 'A Systematic Look at Prototyping', *Approaches to Prototyping*, Berlin: Springer-Verlag.

Ghemawat, S., Gobioff, H. and Leung, S. (2003) 'The Google File System', *SIGOPS Oper.Syst.Rev.*, vol. 37, no. 5, pp. 29-43.

Grochowski, E. and Halem, R.D. (2003) 'Technological Impact of Magnetic Hard Disk Drives on Storage Systems', *IBM Systems Journal*, vol 42, no. 2, pp. 338-346.

Gupta, R., Gupta, H., Nambiar, U. and Mohania, M. (2010) 'Efficiently Querying Archived Data using Hadoop', *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, ACM, pp. 1301-1304.

Hammoud, S., Maozhen Li, Yang Liu, Alham, N.K. and Zelong Liu (2010) 'MRSim: A discrete event based MapReduce simulator', *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, pp. 2993-2997.

Hive (2011) 'Apache Hive', *Apache Software Foundation* [online], <https://cwiki.apache.org/confluence/display/Hive/> (Accessed 25th February 2012).

Jiang, M. and Wang, Y. (2010) 'Optimization of Multi-join Query Processing within MapReduce', *Universal Communication Symposium (IUCS), 2010 4th International*, pp. 77-83.

Jiang, D., Tung, A. and Chen, G. (2010) 'MAP-JOIN-REDUCE: Towards Scalable and Efficient Data Analysis on Large Clusters', *Knowledge and Data Engineering, IEEE Transactions on*, vol. December, 2010,

Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., Manzanares, A. and Qin, X. (2010) 'Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters', *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1-9.

Johnson, R., Harizopoulos, S., Hardavellas, N., Sabirli, K., Pandis, I., Ailamaki, A., Mancheril, N.G. and Falsafi, B. (2007) 'To Share or Not to Share?', *Proceedings of the 33rd International Conference on Very Large Databases*, Vienna, Austria, VLDB Endowment, pp. 351-362.

Kennedy, D. (2011) 'Solid State Drives Can Bring Magic to Your Computer', *ABA Journal*, [online] [http://www.abajournal.com/magazine/article/solid\\_state\\_drives\\_can\\_bring\\_magic\\_to\\_your\\_computer/](http://www.abajournal.com/magazine/article/solid_state_drives_can_bring_magic_to_your_computer/) (Accessed 10th February 2012)

Kim, K., Jeon, K., Han, H., Kim, S., Jung, H. and Yeom, H.Y. (2008) 'MRBench: A Benchmark for MapReduce Framework', *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, pp. 11-18.

Kotidis, Y., Sismanis, Y. and Roussopoulos, N., (2001) 'Shared Index Scans for Data Warehouses', in Kambayashi, Y., Winiwarter, W. and Arikawa, M. (eds) *Data Warehousing and Knowledge Discovery*, Springer Berlin / Heidelberg.

Lam, C. (2011) *Hadoop in Action*, (1st edn) Stamford, CT, USA, Manning Publications.

Liu, G., Zhang, M. and Yan, F. (2010) 'Large-Scale Social Network Analysis Based on MapReduce', *Computational Aspects of Social Networks (CASoN), 2010 International Conference on*, pp. 487-490.

Michael, M., Moreira, J.E., Shiloach, D. and Wisniewski, R.W. (2007) 'Scale-up x Scale-out: A Case Study using Nutch/Lucene', *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1-8.

Moise, D., Antoniu, G. and Bouge, L. (2010) 'Large-scale Distributed Storage for Highly Concurrent Mapreduce Applications', *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1-4.

Monash (2008) 'Three Different Implementations of MapReduce', *Three different implementations of MapReduce* [online], <http://www.dbms2.com/2008/09/05/three-different-implementations-of-mapreduce/> (Accessed 10th February 2012).

Nyberg, C., Shah, M. and Govindaraju, N. (2012) 'Sort Benchmark Home Page', *Sort Benchmark Home Page* [online], <http://sortbenchmark.org/> (Accessed 10th February 2012).

Nykiel, T., Potamias, M., Mishra, C., Kollios, G. and Koudas, N. (2010) 'MRShare: Sharing Across Multiple Queries in MapReduce', *Proc.VLDB Endow.*, vol. 3, no. 1-2, pp. 494-505.

Pike, R., Dorward, S., Griesemer, R. and Quinlan, S. (2005) 'Interpreting the Data: Parallel Analysis with Sawzall', *Sci.Program.*, vol. 13, no. 4, pp. 277-298.

Qiao, L., Raman, V., Reiss, F., Haas, P.J. and Lohman, G.M. (2008) 'Main-memory Scan Sharing for Multi-core CPUs', *Proc.VLDB Endow.*, vol. 1, no. 1, pp. 610-621.

Ruemmler, C. and Wilkes, J. (1994) 'An Introduction to Disk Drive Modeling', *Computer*, vol. 27, no. 3, pp. 17-28.

Rugg, G. and Petre, M. (2007) *A Gentle Guide to Research Methods*, (1st edn) Berkshire, United Kingdom, Open University Press.

Sarje, A. and Aluru, S. (2010) 'A MapReduce Style Framework for Computations on Trees', *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 343-352.

Schwan, P. (2003) 'Lustre: Building a File System for 1000-node Clusters', *Proceedings of the 2003 Linux Symposium*.

Shafer, J., Rixner, S. and Cox, A.L. (2010) 'The Hadoop Distributed Filesystem: Balancing Portability and Performance', *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 122-133.

Sharma, S. (2010) 'Advanced Hadoop Tuning and Optimisation', *Powerpoint on SlideBoom* [online], <http://www.slideboom.com/presentations/116540/> (Accessed 10th February 2012).

Sharp, J., Peters, J. and Howard, K. (2002) *The Management of a Student Research Project*, (3rd edn) Burlington, USA, Gower Publishing.

Shvachko, K., Hairong Kuang, Radia, S. and Chansler, R. (2010) 'The Hadoop Distributed File System', *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1-10.

Thanh, T.D., Mohan, S., Choi, E., SangBum Kim and Pilsung Kim (2008) 'A Taxonomy and Survey on Distributed File Systems', *Networked Computing and Advanced*

*Information Management, 2008. NCM '08. Fourth International Conference on*, pp. 144-149.

The Open University (2007) *Research Project and Dissertation: Study Guide*, Milton Keynes, UK, The Open University.

Tian, C., Zhou, H., He, Y. and Zha, L. (2009) 'A Dynamic MapReduce Scheduler for Heterogeneous Workloads', *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pp. 218-224.

Verma, A., Zea, N., Cho, B., Gupta, I. and Campbell, R.H. (2010) 'Breaking the MapReduce Stage Barrier', *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, IEEE Computer Society, pp. 235-244.

Wang, X., Olston, C., Sarma, A.D. and Burns, R. (2011) 'CoScan: Cooperative Scan Sharing in the Cloud', *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Cascais, Portugal, New York, NY, USA, ACM, pp. 11:1-11:12.

White, T. (2009) *Hadoop: The Definitive Guide*, (1st edn) Sebastopol, CA, USA, O'Reilly Media.

Wikimedia (2012) 'Page View Statistics for Wikimedia projects', *Wikimedia* [online], <http://dammit.lt/wikistats/> (Accessed 10th February 2012).

Xu, C., Shou, L., Chen, G., Hu, W., Hu, T. and Chen, K. (2010) 'Towards Efficient Concurrent Scans on Flash Disks', *Proceedings of the 21st international conference on Database and expert systems applications: Part I*, Bilbao, Spain, Berlin, Heidelberg, Springer-Verlag, pp. 198-212.

Xu, Y., Kostamaa, P. and Gao, L. (2010) 'Integrating Hadoop and Parallel DBMs', *Proceedings of the 2010 international conference on Management of data*, Indianapolis, Indiana, USA, New York, NY, USA, ACM, pp. 969-974.

Yang, H., Dasdan, A., Hsiao, R. and Parker, D.S. (2007) 'MapReduceMerge: Simplified Relational Data Processing on Large Clusters', *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, Beijing, China and New York, NY, USA, ACM, pp. 1029-1040.

Zhang, S., Han, J., Liu, Z., Wang, K. and Feng, S. (2009) 'Accelerating MapReduce with Distributed Memory Cache', *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, IEEE Computer Society, pp. 472-478.

Zukowski, M., Héman, S., Nes, N. and Boncz, P. (2007) 'Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS', *Proceedings of the 33rd international conference on Very large data bases*, Vienna, Austria, VLDB Endowment, pp. 723-734.



## **Bibliography**

Lin, J. and Dyer, C. (2010) *Data-Intensive Text Processing with MapReduce*, (1st edn) Morgan Claypool.

Magoulés, F., Pan, J., Tan, K. and Kumar, A. (2009) *Introduction to Grid Computing*, (1st edn) Florida, United States, CRC Press.

Rauber, T. and Rünger, G. (2010) *Parallel Programming for Multicore and Cluster Systems*, (2nd edn) Berlin, Springer-Verlag.

Venner, J. (2009) *Pro Hadoop*, (1st edn) 2855 Telegraph Avn., Berkeley, California, Apress.

## Index

- Asymmetric join, 19
- Attach (sharing policy), 30
- Benchmarking, 12, 17, 22, 24, 41, 46
- Case studies, 37
- Cluster, 1, 27, 21, 25, 40, 45, 59, 65, 71
- Complexity, 21, 25, 27, 40, 61, 64, 73
- Database (SQL), 7, 13, 27, 30
- Distributed File System (DFS), 5, 8, 11, 19, 43, 45, 47, 51
- Distributed Memory Cache (DMC), 24, 60, 71, 73
- Elevator (sharing policy), 31, 34, 62
- Error (inc. bias), 45
- Fault tolerance, 1, 5, 8, 24, 46
- Field experiments, 38
- Functions, 1, 6, 19, 22
- GenerateCombine, 23
- Google File System (GFS), 8
- GraySort, 12
- Hadoop, 2, 7, 20, 41, 48, 64, 69, 72
- Hadoop Distributed File System (HDFS), 8, 47, 53
- Hive, 21
- (In)compatibility, 26, 61
- Lustre, 8
- MapJoinReduce, 16, 22, 60
- Mappers, 6, 21, 51
- MapReduce, 1, 5
- MapReduceMerge, 22, 23
- Modelling, 38, 40, 43, 49, 69
- MRBench, 13, 43, 73
- Network File System (NFS), 8
- Normal (sharing policy), 30
- Pipelining, 24
- Prototyping, 36, 40, 42, 49, 53, 69
- Reducers, 6, 20, 51
- Scale-up/scale out, 1
- Scalability, 5, 15, 26, 35, 37, 61, 68, 71
- Shared Cyclical File Scanning (SCFS), 3
- Scheduling, 1, 10, 22, 28, 43, 61
- Scope, 22, 26, 29, 62
- Seek times, 8, 32, 34, 63
- Self-Adaptive MapReduce Scheduling Algorithm (SAMR), 16, 61
- Shuffle, 6, 24
- Solid state media, 33, 43, 49, 68
- Stage barrier, 23, 61, 73
- Surveys, 38
- Synthetic micro-benchmarks, 11, 45, 64
- TeraSort, 12, 17, 20, 24, 43, 71, 73
- Transaction Processing Performance Council (TPC), 13, 22, 43
- Tuning, 18, 20
- Virtualisation, 16, 17, 35, 43

## **Appendix A - Extended Abstract**

# **Extended Abstract**

# **An Investigation into whether Shared Cyclical File Scanning Improves the Performance of Concurrent Processing Jobs within the MapReduce Distributed Computing Framework**

**Russell Martin**

**Extended Abstract of Open University MSc Dissertation Submitted 4th March 2012**

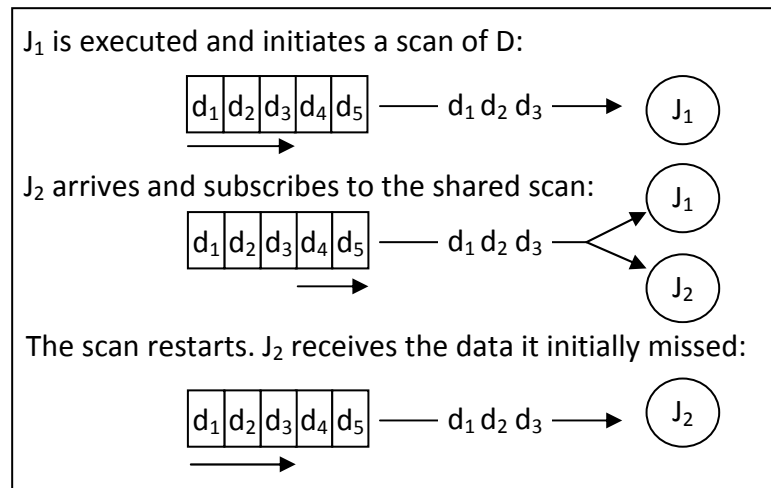
## **Introduction**

Modern organisations collect increasingly large volumes of data. An effective way of processing it is to harness the power of a cluster of computers. Unfortunately, writing distributed applications that can utilise a cluster is not trivial with many issues, such as fault tolerance and result aggregation to consider. MapReduce handles many distribution issues, allowing developers to concentrate on application logic. However, its performance has been criticised with database systems able to out-perform it for many analytical tasks.

A potential performance enhancement for MapReduce is to share the overhead of reading input data amongst multiple jobs. At present, if multiple processing jobs require access to the same file, then that file is read multiple times concurrently. This research investigates whether Shared Cyclical File Scanning (SCFS) can improve MapReduce's performance. Jobs can hook into existing file scans instead of initiating separate ones with the scan restarting from the beginning of the file after it has

finished. This ensures that each job receives all the data in the file, albeit in a different order to which it is stored.

SCFS is illustrated in the diagram below. In this case, there are two jobs,  $J_1$  and  $J_2$  requiring access to the same data file,  $D$ . This file consists of data items  $d_1$  to  $d_5$ . As  $J_1$  arrived first, it has initiated a scan of  $D$  and has reached  $d_3$  when  $J_2$  is submitted to the cluster. Rather than initiating a separate scan of the same file,  $J_2$  subscribes to the same scan, receiving the same data as  $J_1$ . When the scan ends, it restarts allowing  $J_2$  receive the data it missed initially. Any number of jobs can share a scan by subscribing to it at the point at which they start executing.



## Method

A literature review examines other attempts to improve the performance of MapReduce. Unfortunately, variation in experimental technique means that it is not possible to meaningfully compare the performance increases quantitatively. However, several qualitative criteria are established to assess SCFS:

1. Configuration: The additional effort required to configure a cluster.
2. Compatibility: Whether existing MapReduce applications can still execute.
3. Scalability/Fault Tolerance: To what extent these properties are diminished.
4. Application Simplicity: The additional workload for the application developer.
5. Scope: Whether the enhancement applies to a broad range of job types.

The literature review also investigated the concept of shared scanning in other contexts, such as database systems. This led to the development of four hypotheses when SCFS is applied to MapReduce:

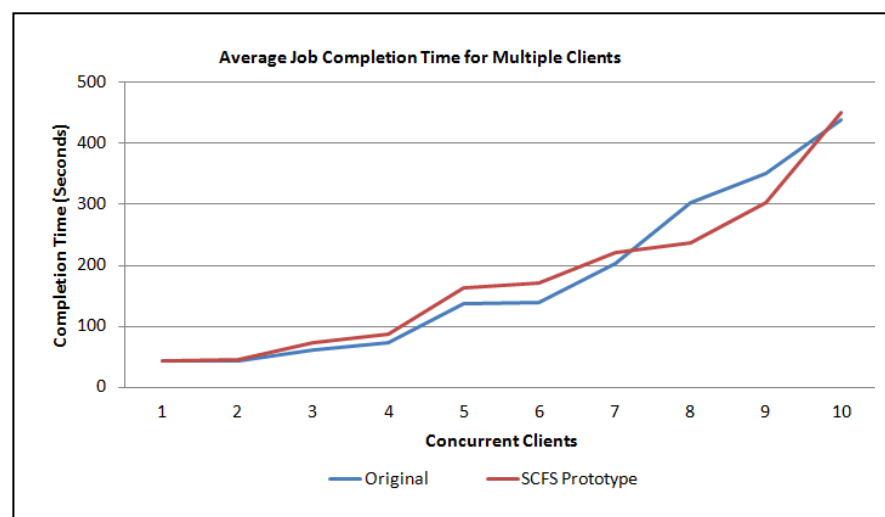
1. As the number of concurrent jobs utilising the same file increases, the performance of SCFS relative to a non-SCFS system will improve.
2. The performance of SCFS will be hindered by slow running jobs.
3. As the size of the input data grows the performance of SCFS relative to a non-SCFS will improve.
4. The use of solid state media (as opposed to traditional magnetic media) will reduce any performance benefit resulting from SCFS.

An SCFS/MapReduce prototype was developed to test these hypotheses. Experiments were conducted on a test cluster by comparing the performance of the SCFS prototype against an unmodified Hadoop cluster. The test cluster consisted of two slave nodes and one master node.

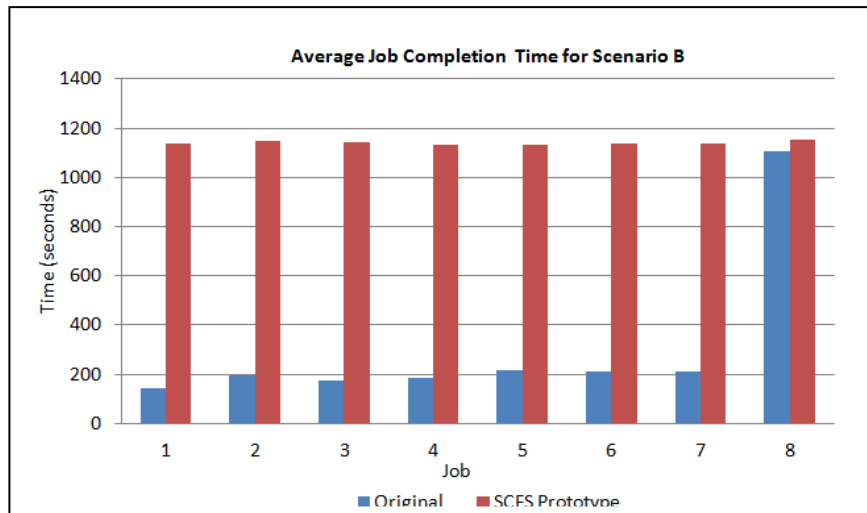
Hypotheses 1, 2 and 3 were tested with experiments that varied the number of concurrent scans, the speed of jobs and the size of the input data. Hypothesis 4 could not be examined due to the resource constraints of the project as no solid state media was available.

## Results

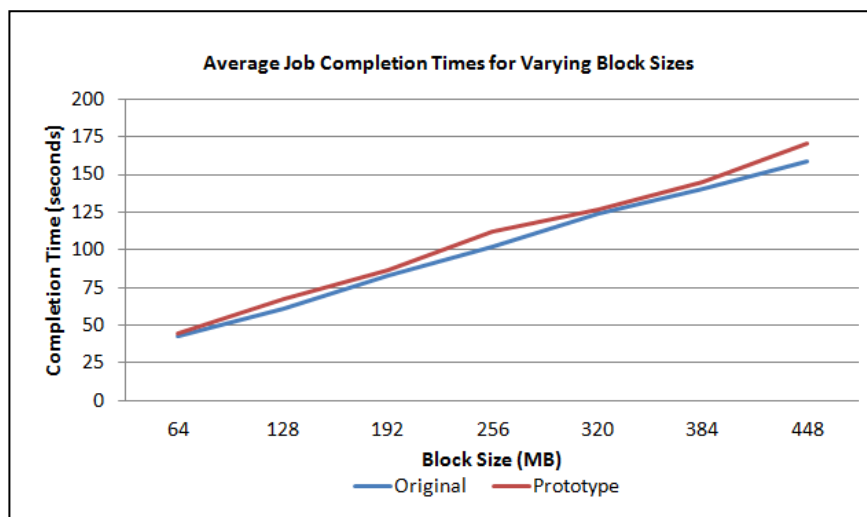
Experiment 1 revealed that SCFS performed worse for low numbers of concurrent jobs. However, as illustrated below, it out-performed the original system when there was a higher number (8 or 9) of concurrent jobs sharing the same scan.



Experiment 2 showed that the SCFS's performance was significantly hindered by slow-running jobs. The next graph shows the average completion time for each of the 8 simultaneous jobs submitted to the cluster, where job 8 was a slow running job. With SCFS, all the fast jobs are constrained by the single slow job while the original system experienced little impact with the fast jobs able to progress unhindered. This demonstrates a significant disadvantage of SCFS.



Experiment 3, which varied the size of the input for two concurrent jobs, showed that the relative performance the systems remained the same, as illustrated in the graph below.



## Analysis

Hypothesis 1 and 2 were shown to be true under the circumstances tested. Experiment 1 demonstrated that there are critical points between which the use of SCFS provides a performance benefit. Experiment 2 showed that SCFS suffers a serious limitation - slower jobs will hinder progress of faster jobs. Insufficient data was collected to fully investigate Hypothesis 3. However, Experiment 3 showed that for small numbers of concurrent jobs, the input file size had little impact on the relative performance of SCFS and the original system.

In terms of the qualitative criteria, the major disadvantage of SCFS was its compatibility with jobs that require strict ordering of data. MapReduce applications that require the data in a particular order will never be able to be executed with SCFS. SCFS also requires additional configuration and adds some complexity to the Map code supplied by the developer, but these are relatively minor disadvantages. Unfortunately

it was not possible to examine SCFS's impact on a MapReduce's fault tolerance or scalability.

## **Discussion**

There are circumstances where SCFS improves MapReduce's performance. However most experiments resulted in worse performance and those circumstances where using SCFS is beneficial may not be encountered frequently in practice. This research highlighted a serious disadvantage of SCFS - that a slow running job can significantly degrade the performance of other jobs that share the scan.

Further work needs to be done to investigate whether SCFS hinders MapReduce's scalability and whether solid state media has the potential to eradicate any performance improvement that results from SCFS. There should also be investigation into whether the impact slow jobs can be mitigated. The final area for future work is to address the current difficulty in comparing different MapReduce enhancements. A suggested way forward is to develop the list of criteria presented in this research into a formal methodology that could be employed alongside standard performance benchmarking methods that currently exist.



## **Appendix B - Source Code**

The source code and compiled libraries are available on the

CD-ROM attached to the rear cover of this document.

The prototype system is also hosted as a Google Code project:

<https://code.google.com/p/mapreduce-scfs/>