IFMapReduce

Interactive analysis of Big Data

Aalborg University
Department of Computer Science
Student report
Selma Lagerlöfs Vej 300
Telefon 99 40 99 40
Fax 99 40 97 98
http://www.cs.aau.dk

Title: IFMapReduce - Interactive analysis of Big Data

Theme: Functional Programming and Big Data

Project period:
SW10, 2/2/2015 - 8/6/2015

Project group:
dpt103f15

Participants:
Christian Møller Jensen
Lars Harald Larsen

Supervisor:
Bent Thomsen

Copies: 4

Pages: 47

No. of appendix pages: 14

Finished: 5/6/2015

Synopsis:

This report investigates the opportunities for interactive analysis of Big Data on the .NET platform through F# Interactive. This investigation includes the design of a programming framework called IFMapReduce which uses F# Code Quotations to facilitate code shipping in cluster computing.
A prototype of IFMapReduce have been implemented and evaluated through benchmarks using Pagerank and Wordcount algorithms. In addition the report present several suggestions to improve IFMapReduce and its prototype.

SUMMARY

This report investigates the opportunities for interactive analysis of Big Data on the .NET platform through the F# Interactive Read Eval Print Loop. The hypothesis is that F# Interactive can be used to ship closures in the form of F# Code Quotations to facilitate interactive Big Data analysis. F# Code Quotations represents F# code as an Abstract Syntax Tree which provides opportunities for manipulating the code programatically.

The report briefly presents some of the tools used to analyze big data such as Hadoop, Spark and Dryad. Furthermore programming languages focused on distributed computation are described.

To investigate the use Code Quotations as a facility for Big Data analysis, the report documents the design and implementation of a programming framework prototype called IFMapReduce that uses Code Quotations to distribute code in a cluster. The IFMapReduce prototype is based on the foundations of the FMapReduce framework and uses a master/slave architecture. IFMapReduce uses Hadoop Distributed File System to handle data storage.

IFMapReduce makes use of a data abstraction called IFDataset to facilitate computations over data, these IFDatasets can be used in F# interactive to specify distributed jobs. IFDatasets uses Code Quotations to manipulate data through a predefined API.

The IFMapReduce prototype is compared to the Spark and FMapReduce frameworks by performing benchmarks of Pagerank and Wordcount algorithms. These benchmarks are performed to compare the running time and work distribution of the frameworks in order to evaluate the use of F# Code Quotations as a code distribution facility.

The report discusses ideas on how to improve the IFMapReduce framework, these ideas are based on the benchmark results and the opportunities enabled by utilizing F# Code Quotations.

PREFACE

This report investigates F# Code Quotations as a distribution facility for interactive analysis of Big Data on the .NET platform. This is done by designing a distributed Big Data analysis framework called IFMapReduce, which through a data abstraction called IFDataset can build distributed jobs in the F# Interactive shell.

A prototype implementation of IFMapReduce is compared to the Spark and FMapReduce frameworks using benchmarks, to determine if Code Quotations are fast enough to facilitate analysis of big data.

Chapter 1 describes the motivation behind this project and gives a short introduction to MapReduce and F# Code Quotations. Work related to this project is presented in chapter 2, while chapter 3 presents the problem definition the project is based upon.

The architecture of IFMapReduce is presented in Chapter 4 along with a description of the IFDataset abstraction. Chapter 5 documents the benchmarks used to evaluate the prototype of IFMapReduce while chapter 6 discusses ideas on how to improve IFMapReduce and its prototype. The report is concluded in chapter 7.

The appended CD contains the source code for the IFMapReduce frameworks and the IFMapReduce scripts used in the benchmarks.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

Big Data is a term used for datasets that have grown to sizes where traditional analysis methods have become inadequate. Big Data sets are becoming more common as the industry and research community become more adept at gathering and utilizing Big Data. In April 2014, facebook estimated that the amount of data stored in their Hive cluster was almost 300 PB and growing at a rate of 600 TB every day[24]. The European Organization for Nuclear Research(CERN) estimates that their data cluster, consisting of 11.000 servers, process roughly 1PB of data each day, this includes about 6000 database changes per second[1], most of CERNs data is gathered from various measurement equipment such as the Large Hadron Collider which measures the collision of approximately 600 million particles per second.

The general approach to analyzing Big Data is to make use of cluster computing, however writing distributed parallel applications is generally considered hard[4, 16]. To alleviate some of the challenges in regards to distributed parallel programming, several methods to analyze Big Data have been proposed, each with their own field of specialization.

A popular approach to Big Data analysis is the MapReduce programming model which imitates the functional programming concepts of map and reduce. The model was first presented by Google[5] and then made widely available by the Apache Hadoop project[9]. Several extensions to the MapReduce model have been proposed, resulting in a multitude of frameworks with MapReduce as the core such as PIG[23] and Hive[29]. Other approaches to Big Data analysis includes Parallel DBMS, graph databases, distributed file systems and Extract Transform Load.

Typically most cluster computing methods have focused on batch jobs which

analyse large amounts of data at a time, these jobs can run for hours or days at a time before results are available. While PIG and HIVE gives the illusion of adhoc queries, each query is translated to a batch of MapReduce jobs, which introduces some overhead as each of these jobs are treated as separate entities that each reads from consistent data storage. To eliminate this overhead some programming frameworks such as Spark[33] focus on interactive analysis where programmers can load and persist a dataset in memory to perform multiple queries on this dataset.

The approaches to analyzing Big Data are diverse and differences in specialization means that choosing the right tools is important for providing the optimal solutions. In recognition of this, several programming frameworks are made to cooperate with each other, such that multiple frameworks can run on the same cluster simultaneously. An example of this is Hadoop's second generation platform YARN[30], which is a resource negotiator that can allocate compute nodes in a request based manner, allowing multiple frameworks to allocate workers on the same cluster.

A correlated problem to analyzing big data is how to store the data itself. When the amount of data increases, so does the pressure on the underlying storage facilities, meaning that storage facilities should be able to grow with the data. However, some of the Big Data analysis methods are dependent on specific storage facilities and/or data structures. Most approaches to storing Big Data is to partition the data across a cluster of nodes, however this introduces a need to ensure data integrity in case of node failure.

Distributed storage introduces some issues with data locality, where transferring data between nodes can result in significant overhead in an already slow process. To account for the locality of data most distributed frameworks attempt to move the computations to the nodes storing the data. Some solutions such as the Hadoop Framework ships compiled applications to the relevant nodes, while other solutions such as Spark ships code closures in the form of Java Objects.

This report investigates the possibilities of allowing interactive analysis of big data through the F# Interactive Read Eval Print Loop. The hypothesis is that F# Interactive can be used to interactively ship closures in the form of F# Code Quotations to a distributed framework, which distributes these closures for execution on a cluster, allowing for interactive analysis on the .NET platform. To test this theory, this report documents the design of a programming framework called IFMapReduce.

1.2 MapReduce

The MapReduce programming model is designed to run across a a cluster of commodity nodes, it was presented by Google[4] and has since been widely adapted with the availability of the Apache Hadoop project. MapReduce is based on the

functional programming concepts of map and reduce functions.

In MapReduce the user specifies a map and a reduce function. The map function takes a key value pair as arguments and produces a list of key value pairs. The output from the map function is then partitioned based on the key and forwarded to the reduce functions. The reduce function takes a key and a list of values associated with that key as input and aggregates the values to produce a potentially smaller set.

Figure 1.1 illustrates the work flow of a MapReduce job, as can be seen in the figure MapReduce requires the input to be split, such that each mapper and each reducer works on its own unique dataset.

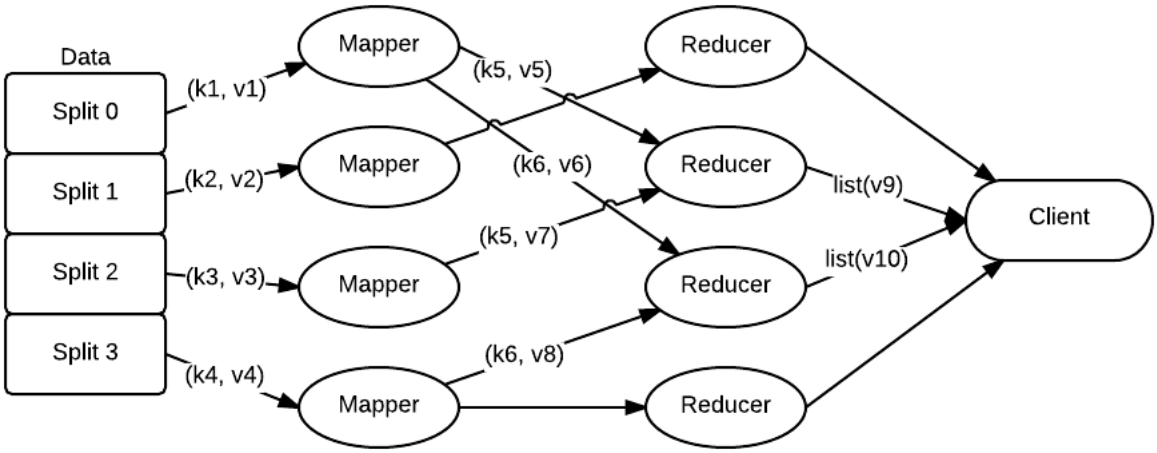

Figure 1.1: The execution flow of a MapReduce job

1.3 Code Quotations

Code Quotations is a feature in F# that generates an Abstract Syntax Tree(AST) based on F# code[20]. Having code in the form of an AST provides opportunities for working with the code programatically, including compiling the AST back into F# Code or other languages.

Code Quotations can be specified using the `<@` and `@>` delimiters for typed quotations and `<@@` and `@@>` for untyped quotations. Figure 1.1 illustrates some basic code quotations, where line 1 is a quotations of a lambda expression to square an integer, while line 3 is an expression that adds two and three.

```
let square : Expr<int> = <@ fun x : int -> x * x @>

let five : Expr = <@@ 3 + 2 @@>
```

Listing 1.1: Code Quotation examples

The splicing operator `%` can be used to combine code quotations, meaning that it is possible to include Code Quotations within a Code Quotation. Listing 1.2 shows how the splicing operator can be used to combine two expressions.

```
let five = <@ 3 + 2 @>

let squarefive = <@ %five * %five @>
```

Listing 1.2: Code Quotation splicing example

Listing 1.3 shows the AST gained by priting `squarefive` from listing 1.2.

```
Call (None, op_Multiply,
  [Call (None, op_Addition, [Value (3), Value (2)]),
   Call (None, op_Addition, [Value (3), Value (2)])])
```

Listing 1.3: AST generated from listing 1.2

Unlike other .NET Code Quotations, the F# Code Quotations are serializable, meaning that it is possible to ship these across nodes, allowing these to be used to facilitate distribution of F# code.

CHAPTER 2

RELATED WORK

This chapter describes work related to distributed computing, this includes programming frameworks and languages focused on writing distributed applications. Code examples are supplied to give a quick introduction on how the frameworks can be used.

2.1 Apache Hadoop

Apache Hadoop[9] is an open source project for cluster computing containing several sub projects: Hadoop Common which is a collection of common utilities to support the other Hadoop modules, the Hadoop Distributed File System (HDFS) which specializes in high-throughput access to application data, Hadoop YARN for job scheduling and cluster resource management and Hadoop MapReduce which is a MapReduce framework built upon the other subprojects.

2.1.1 YARN

YARN[30] (Yet Another Resource Negotiator) is part of the second generation of the Hadoop framework. The intention of Yarn is to separate resource management from the programming platform to allow multiple frameworks to be used on a single cluster. As illustrated in figure 2.1 YARN uses a dedicated Resource Manager to keep track of the global state of the cluster and allocate resources based of this state. Each node in a cluster runs a local Node Manager which monitors local resources and status.

YARN applications are facilitated by a user defined driver process called an Application Master, which is in charge of handling the job related tasks such as starting

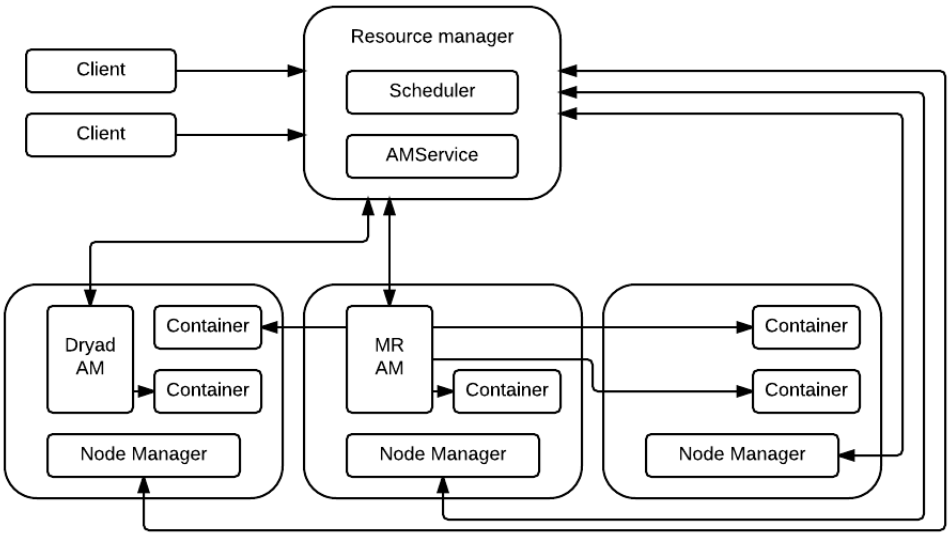

Figure 2.1: The YARN architecture

and monitoring workers. YARN separate resources into bundles called containers, where Application Masters can lease Containers to start worker processes.

2.1.2 Hadoop Distributed File System

The Hadoop Distributed file system(HDFS)[27] was implemented to be used as storage facility for Hadoop's MapReduce engine. In HDFS files are separated into blocks, these blocks are distributed throughout the cluster meaning that a single file is stored across multiple nodes, file consistency is ensured by replicating blocks across several nodes.

HDFS uses a dedicated NameNode server which handles file metadata such as block placement, while the actual data is stored on servers called DataNodes.

2.1.3 Hadoop MapReduce

Hadoop MapReduce v2 is implmented as a YARN Application Manager[9] which facilitates the execution of MapReduce jobs. This Application Manager provides status updates on the execution of MapReduce jobs through a web interface. By default Hadoop MapReduce jobs are specified as Java classes, but the Application manager allows for execution of binary files through streaming.

Listing 2.1 illustrates how a Wordcount job can be specified in Hadoop.

```
1  public static class TokenizerMapper extends Mapper<Object, Text,
     Text, IntWritable>{
2    private final static IntWritable one = new IntWritable(1);
3    private Text word = new Text();
4
5    public void map(Object key, Text value, Context context) throws
       IOException, InterruptedException {
6      StringTokenizer itr = new StringTokenizer(value.toString());
7      while (itr.hasMoreTokens()) {
8        word.set(itr.nextToken());
9        context.write(word, one);
10     }
11   }
12 }
13
14 public static class IntSumReducer extends Reducer<Text,IntWritable,
     Text,IntWritable> {
15   private IntWritable result = new IntWritable();
16   public void reduce(Text key, Iterable<IntWritable> values,
       Context context) throws IOException, InterruptedException {
17     int sum = 0;
18     for (IntWritable val : values) {
19       sum += val.get();
20     }
21     result.set(sum);
22     context.write(key, result);
23   }
24 }
```

Listing 2.1: Wordcount in Hadoop[10]

2.2 MarsHadoop

MarsHadoop[8] is a MapReduce framework built to utilize GPU's on a distributed cluster. MarsHadoop hides the complexity of GPU architectures and API's behind a MapReduce interface.

MarsHadoop is an extension of the Mars MapReduce framework, which is a single node MapReduce framework that can utilize GPUs for computation. Mars utilizing GPUs can be up to an order of magnitude faster than Mars using CPU[15]. MarsHadoop uses the core of the Hadoop framework to distribute computations.

GPU arrays cannot be of dynamic size, and must be allocated before data can be stored, therefore Mars makes use of secondary functions to compute the size of intermediate and final datasets. These secondary functions are user defined and executed on the CPU before the data is moved to the GPU.

2.3 Spark

Apache Spark[33] is an open source cluster computing framework that offers a programming model with an execution engine optimized for iterative algorithms that supports in-memory computing. Spark allows interactive analysis from both Scala and Python shells through a modified version of the Scala interpreter allowing for faster prototyping on Big Data sets.

Sparks uses an abstraction of distributed memory called Resilient Distributed Datasets(RDDs)[32] that allows programmers to perform in-memory computations on clusters in a fault tolerant manner. RDDs are read-only, partitioned collections of records which can be generated from operations on other RDDs or data extracted from a stable storage. Spark saves information about how RDDs were created and can use this information to recover lost RDD partitions.

Spark jobs are written as a driver program that can manipulate RRDs by using two types of operations. `Transformations` are an operation which create a new dataset from an already existing RDD, and `Actions` which return a value to the driver program after performing computations on the dataset. All transformations in Spark are lazy, they are only computed when an `action` requires the result to be returned to the driver program.

Spark introduces two types of shared variables. `Broadcast variables` which are read only variables that are broadcasted to all workers, `broadcast variables` allow the programmer to keep a variable cached on each node rather than shipping a copy of it with tasks. The other type of variable is `Accumulators` which are variables that workers can append to. The data in an `Accumulator` can only be retrieved by the driver program.

Listing 2.2 shows an example of a Wordcount job in Spark. The listing shows the RDD `counts` being generated by using `flatMap()` to split the input file into words, then these words are mapped to KeyValue pairs and then aggregated using `reduceByKey()`. The `counts` RDD is then saved as a text file to the distributed file system.

```
1 val file = spark.textFile("hdfs://...")
2 val counts = file.flatMap(line => line.split(" "))
3                  .map(word => (word, 1))
4                  .reduceByKey(_ + _)
5 counts.saveAsTextFile("hdfs://...")
```

Listing 2.2: Wordcount in Spark[11]

2.4 Disco

Disco[21] is a MapReduce framework built using the Erlang language to take advantage of Erlang's concurrency and distribution features. As shown in figure 2.2 Disco's architecture uses a dedicated master to handle resource management and task scheduling while the other nodes in the cluster runs a slave host responsible for starting workers to perform map or reduce tasks as instructed by the master.

In Disco jobs are started by sending job-packets containing metadata about the jobs such as which data to use and a zipped worker directory containing binary executables needed to execute the jobs. To allow multiple workers to use the same binary file, each task have their own directory to store temporary data and result data.

Listing 2.3 shows how a Wordcount job can be specified in Disco using Python.

```
1 def map(line, params):
2   for word in line.split():
3     yield word, 1
4
5 def reduce(iter, params):
6   from disco.util import kvgroup
7   for word, counts in kvgroup(sorted(iter)):
8     yield word, sum(counts)
```

Listing 2.3: Wordcount in Disco[3]

2.4.1 Disco Distributed File System

Disco makes use of the Disco Distributed File System(DDFS)[21] to store data. DDFS is designed for storing and retrieving large immutable files and files pushed to DDFS are separated into chunks. Rather than using file names DDFS uses a tag system to identify and group its data, where sets of data objects are tagged with arbitrary names, and can be retrieved using these tags. Tags can contain chunks, URL links to external data and even other tags. Fault tolerance in DDFS is handled by replication of both data and metadata. By default DDFS compresses chunks to limit the storage space needed.

2.5 FMapReduce

FMapReduce[18] is a MapReduce framework built in F#, it consists of three main components as illustrated in figure 2.3, a dedicated master, a client and a number of workers. FMapReduce is built on top of DDFS, where it schedules tasks based on the file metadata.

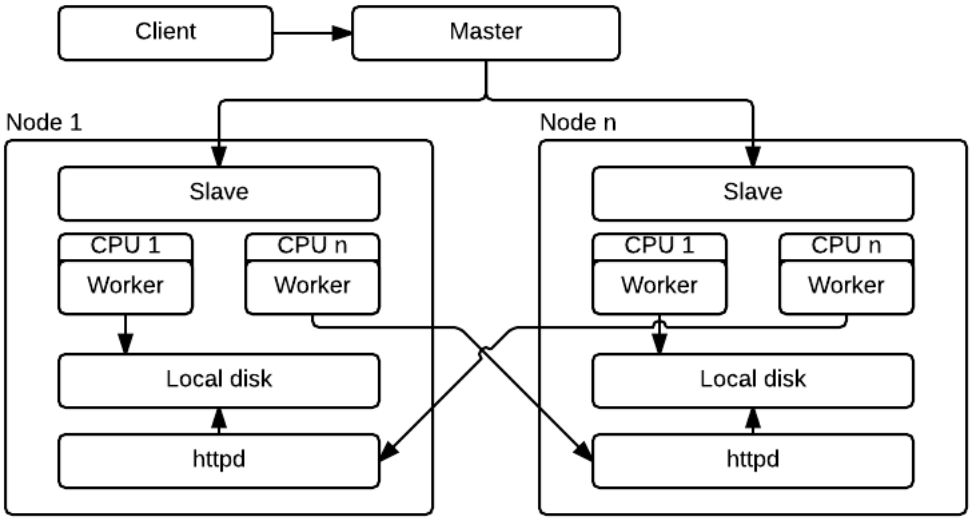

Figure 2.2: The architecture of Disco

The worker nodes runs a daemon that spawns threads to execute tasks as instructed by the master. when this daemon is started it announces its presence to the master and starts heartbeating its status to the master node. The master uses these heartbeats to monitor the status of the cluster, and task instructions are sent as replies to these heartbeats.

In the FMapReduce framework jobs are written as .NET classes which inherit from the FMapReduce Class. Listing 2.4 shows how map and reduce tasks are specified, by overriding the `distributedMap` and `distributedReduce` methods. The `yieldKV` method is used to generate the output by adding the key and value to an output list. The specified MapReduce jobs are compiled to `.dll` assemblies which can be dynamically loaded by the workers at runtime to get the job specifications. To lessen the burden on the master, the workers retrieve the compiled assemblies directly from the client.

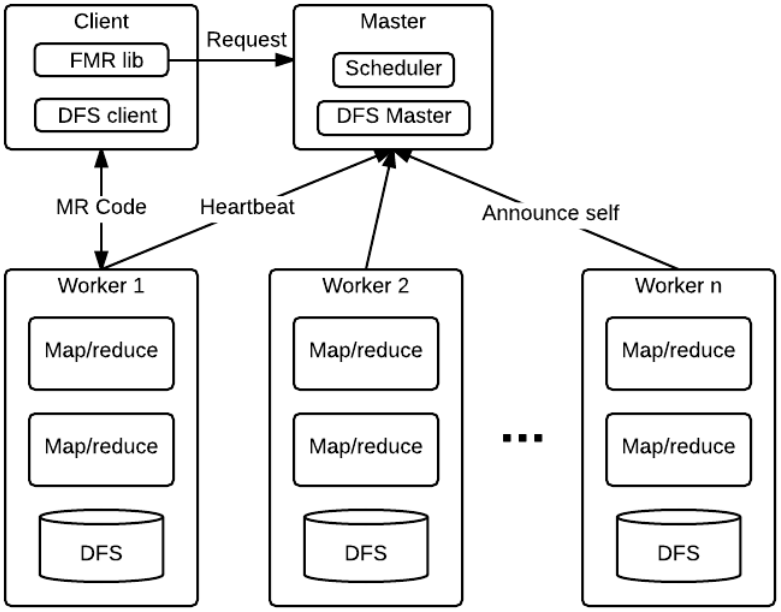

Figure 2.3: The FMapReduce architecture

```
type fmapreducetester() =
  inherit FMapReduce()
    override this.distributedMap (key : string) (value : string) =
      for word in key.Split([|" "; "\n"|], StringSplitOptions.
        RemoveEmptyEntries) do
        this.yieldKV word (string 1)

    override this.distributedReduce (key : string) (value : string
      array) =
      this.yieldKV key (string (value |> Array.map int |> Array.sum
        ))
```

Listing 2.4: Wordcount job in FMapReduce

2.6 MBrace

MBrace[6] is a programming model and execution runtime for distributed computing on the .NET platform. MBrace makes use of an abstraction called Cloud Workflows, which makes us of monads through F#'s Computation Expressions to specify distributed computation. MBrace imitates the programming style of F# Asynchronous

workflows.

The MBrace runtime uses a master/slave architecture where Cloud Workflows are submitted to a scheduler which interprets the structure of the workflow to assign work to slave nodes.

MBrace does not handle data storage, but relies on underlying storage facilities such as FileSystems, SQL and/or Azure storage providers.

Listing 2.5 illustrates how MBrace can be used to describe a distributed workflow. The example defines two Cloud Workflows called `job1` and `job2`, and uses `Cloud.Prallel` to execute the workflows in parallel.

```
cloud {
  let job1 = cloud { return 1 }
  let job2 = cloud { return 2 }
  let! [| result1 ; result2 |] = Cloud.Parallel [| job1 ; job2 |]
  return result1 + result2
}
```

Listing 2.5: Cloud Workflow in MBrace[6]

The MBrace runtime allows for interactive analysis and deployment through F# interactive.

2.7 Dryad

Dryad[16] is a distributed execution engine, that allows execution of parallel applications in a cluster. In Dryad applications are modeled as directed acyclic graphs (DAG). Where vertices defines operations performed on data and edges define channels where the data run through. When executing jobs the vertices in the DAG are executed on available nodes in the cluster.

As illustrated in figure 2.4 the architecture of Dryad contains a `Job Manager(JM)` that maintains the current DAG and schedules work across the available nodes in the cluster. The `Name Server(NS)` maintains a list of available resources and makes these resources available to the Job Manager. Each worker node in a dryad cluster runs a `Daemon(D)` where the Job Manager can starts processes. Dryad uses a distributed file system to distribute the data throughout the system.

The first time a vertex is executed, the binary is sent by the Job Manager to the Daemon which then keeps it in memory. Dryad's DAG based architecture allows vertices to have an arbitrary number of inputs and outputs, as opposed to the MapReduce model where only single input single output is allowed.

To make the development process for dryad jobs simpler DryadLINQ[17] was introduced because it allows programmers to use LINQ when specifying DAGs. DryadLINQ provides additional LINQ operators to accommodate distributed com-

puting, such as operations for partitioning datasets and iterating over partitioned datasets.

Listing 2.6 shows an example of a Wordcount job specified in DryadLINQ. `DryadLinq.GetTable` loads data from a distributed file system into a table `docs`. The table is then manipulated as though the data was locally available. `.ToDryadTable()` saves the results to the distributed file system.

```
1 var docs = DryadLinq.GetTable<Doc>("file://docs.txt");
2 var words = docs.SelectMany(doc => doc.words);
3 var groups = words.GroupBy(word => word);
4 var counts = groups.Select(g => new WordCount(g.Key, g.Count()));
5
6 counts.ToDryadTable("counts.txt");
```

Listing 2.6: Wordcount in DryadLINQ[31]

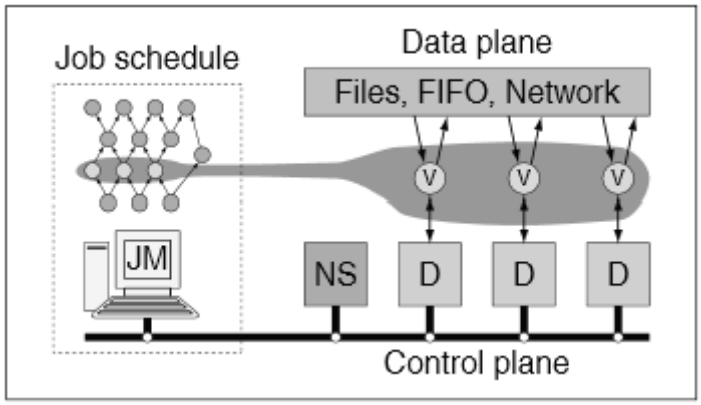

Figure 2.4: The architecture of Dryad[16]

2.8 Cloud Haskell

Cloud Haskell[7] is an extension to the Haskell Language, it is a domain-specific language targeted at distributed computing environments, that is inspired by the Erlang language to makes use of Message passing between `Processes` as the main communication method.

Where Erlang only provides untyped messages, Could Haskell supports two types of messages: untyped messages similar to messages in Erlang and typed messages which take advantage of Haskell's strong type system, both types of messages are asynchronous, reliable and buffered. To transmit data in Cloud Haskell the data

must implemented the `Serializable` class, ensuring that the type can be converted to and from binary.

To serialize closures Cloud Haskell introduces the `MkClosure` function which takes a static function and an environment as arguments, where the environment can be used to capture the free variables needed by the function. To solve the problem of deserialization Cloud Haskell proposes that serialization and deserialization happens at closure-construction time rather than closure-serialization time.

Fault tolerance in Cloud Haskell is based on ideas from Erlang, where a process terminates instead of attempting recovery. To makes this possible processes can monitor other processes, if a process terminates the monitoring process is notified. Recovery of failed processes is left to a higher level framework or application.

2.9 JoCaml

JoCaml[12] is an extension of Objective Caml and is designed for Concurrent Distributed and Mobile Programming in a functional setting based on the join calculus. It is disigned to provide a simple and well defined model for distributed applications. Programs written in JoCaml can be executed on a single machine or in a distributed manner on several machines.

JoCaml relies on asynchronous message passing and provides two ways of passing message content which are by copying or by referencing.

JoCaml allows functions to be sent to remote machines by sending a copy of the functions code and the values for its local variables, any call of that function will then be executed on the remote machine. JoCaml can invoke functions defined on another machine by providing function name and routing information.

CHAPTER 3

PROBLEM STATEMENT

This report investigates the hypothesis that F# Code Quotations can be used to distribute code for interactive analysis of Big Data using the F# Interactive shell on the .NET platform. This investigation includes:

- Designing a programming framework to facilitate distribution and execution of Code Quotations on a cluster.
- Implementing a prototype of the designed framework.
- Evaluating the prototype with other big data analysis frameworks through benchmarking.

CHAPTER 4

IFMAPREDUCE

This chapter documents the design of a programming framework called IFMapReduce. Section 4.1 elaborates on the architecture of IFMapReduce while section 4.2 elaborates on the data abstraction used to specify jobs in IFMapReduce.

4.1 Architecture

This section describes the architecture of IFMapReduce shown in figure 4.1 and elaborates on the responsibilities of the major components. IFMapReduce is built upon the foundations of the FMapReduce framework[18] described in section 2.5. The IFMapReduce prototype is built on top of HDFS, which is used to handle both input and output, furthermore the metadata provided by HDFS is used by the master when scheduling tasks. IFMapReduce refers to HDFS blocks as chunks.

IFMapReduce uses a Master/slave architecture, where a dedicated master schedules tasks for the workers. The workers send heartbeat messages to the master which is used to track worker status, and the master sends task instructions as replies to these heartbeat messages.

IFMapReduce jobs are specified using `IFDatasets` (see section 4.2), which are abstractions over data and operations needed to generate the dataset. These `IFDatasets` can initiate jobs by forwarding metadata describing the input along with the list of operations. The master uses the supplied metadata to schedule tasks which it forwards to the workers with the list of operations.

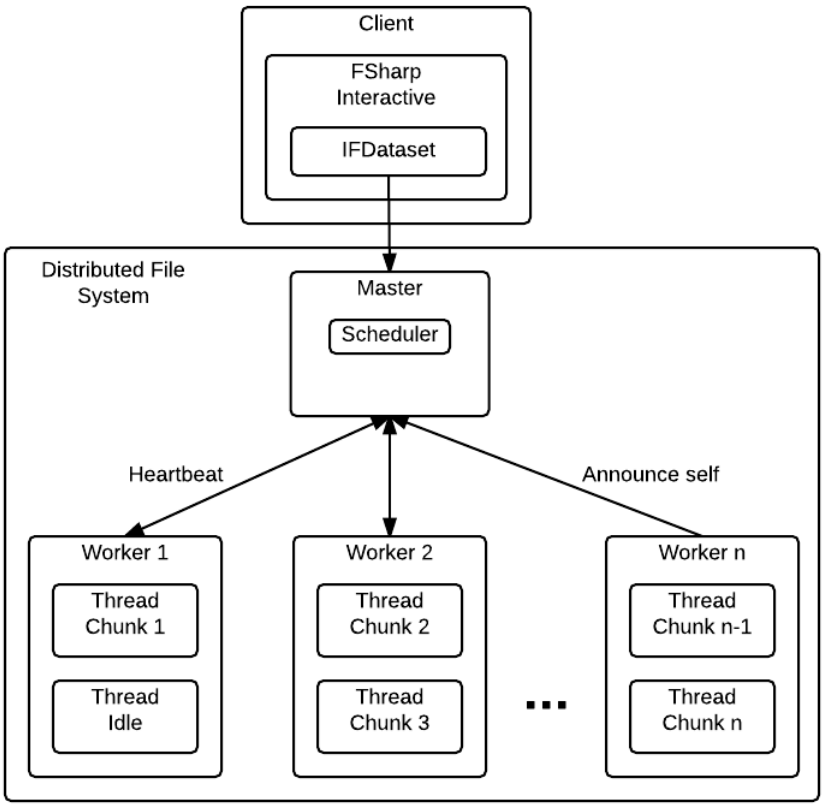

Figure 4.1: The architecture of IFMapReduce

4.1.1 Master

The master is responsible for scheduling jobs and tasks to the workers. The master monitors the status of the cluster through heartbeats from the workers, task instructions are sent as replies to these heartbeat messages.

The master assumes the role of NameNode in the HDFS cluster and tasks are scheduled based on the file metadata provided by HDFS. HDFS partitions files into a number of chunks and the IFMapReduce master schedules a task for each file chunk. Furthermore the master also uses the file metadata to account for locality when scheduling tasks.

As HDFS does not allow simultaneous writes to a file, IFMapReduce uses a leasing system, where a worker must lease a file from the master before it is allowed to write to that file. This ensures consistency at the cost of overhead.

4.1.2 Worker

When a worker starts up, it announces its presence to the master, this announcement includes a description of the resources available for computation. Following this announcement the worker begins to heartbeat its status to the master.

The workers are responsible for executing the tasks assigned by the master. When a worker receives task instructions from the master it spawns a thread to handle that task, instead of binding threads to a specific processor IFMapReduce assumes that the underlying operating system is fair when handling thread execution.

If input is locally available, the worker reads the chunk directly from HDD, however if the input is not locally available it establishes an ssh connection to a worker with the chunk available and read the chunk through this connection, this requires that the cluster is setup to support passwordless ssh between slaves.

4.1.3 Client

IFMapReduce provides a library containing the `IFDataset` type which is used to specify jobs in the F# Interactive Read Evaluate Print Loop.

Using `IFDatasets` the client builds a list of operations that must be applied to the dataset, the client can use the `IFDataset` to initiate requests by sending the file metadata and the list of operations to the master.

4.2 IFDataset

To specify manipulation of data, IFMapReduce makes use of a data abstraction called `IFDataset`, these can be generated from HDFS files or other `IFDatasets`.

An `IFDataset` consists of a reference to a file and a set of operations needed to generate that `IFDataset`.

`IFDatasets` provides a set of predefined operations that describes how quoted expressions should be applied to the data. The operations supported by the IFMapReduce prototype are described in section 4.2.2, some of these operations are handled in a lazy manner where computation is only initiated when necessary.

Listings 4.1 shows an examples of how to specify a Wordcount job in IFMapReduce.

```
let iDataset = new FInteractiveClient.Dataset("/user/slave/bigfile.
  txt", "Words", "localhost", "7040")
let mappedValues =
  iDataset.Map <@ fun (x : string) -> (x, 1) @>
                  "StringInt"

let reduceIntegers =
  <@ fun (x : string) (y : int list) -> (x, List.sum y) @>

let combinedValues =
  mappedValues.Combine reduceIntegers
                       "StringInt"
let reducedValues =
  combinedValues.Reduce reduceIntegers
                        "StringInt"
                        "/user/slave/testoutput.txt"
                        1
```

Listing 4.1: IFMapReduce Wordcount using Map operation

The IFMapReduce prototype makes use of a union type to handle the input and output data for the operations, as a consequence of this, the IFMapReduce prototype only support intermediate and result data of the types `string`, `string*string`, `string*int`, `int` and `int*int` lists, where `String*Int` is a `Tuple` containing a string as the first value and an integer as the second value.

4.2.1 Quoted expressions

The `IFDataset` operations are specified using F#'s quoted expressions, which allows programmers to use the F# syntax to write the operations. However the prototype of IFMapReduce imposes some limitations on how these expression can be formed. All of the expressions must be formed as lambda expressions taking an input suitable for the operation.
IFMapReduce uses FsPickler[22] to serialize expressions, to correctly deserialize the expressions IFMapReduce requires that the data type of the output is specified, as illustrated in listing 4.1 this is done by supplying a string describing the out-

put type as the secondary parameter to the operation. Besides the output types, IFMapReduce requires that the quoted expressions themselves have their input type specified.

The IFMapReduce prototype does not support the use of the Quotation splicing operator which can be used to nest a quoted expression within another quoted expression, it does however support nested functions as long as these are specified within the expression. Furthermore, all values used in the expression must be specified within the expression, meaning that external values cannot be used.

4.2.2 Operations

This section provides an overview of the operations supported by the `IFDatasets`. When the operation states that it is applied lazily this means that the operation is applied locally without any awareness of the state of the other chunks. If multiple consecutive operations are lazy, these operations are chained to reduce IO.

All operations (with the exception of `Store`) returns an new `IFDataset` corresponding to the data with the specified changes.

Factory methods

An `IFDataset` is in its basis form a file, therefore the `IFDatasets` provides a list of factory methods used to specify how to parse the file. The IFMapReduce prototype supports three factory methods: `ReadAll` which reads the entire chunk as a single string, `Lines` which splits the chunk into one item per line and `Words` which splits the chunk into words.

All of the methods returns a list of strings which must be manipulated through `IFDataset` operations to change the datatype.

Besides the factory methods it is possible to use the standard constructor, however this requires that the splitting method is provided as a string to correctly split the file as shown in listing 4.1.

Apply

The `Apply` operation is applied lazily, it takes the entire local dataset as input in the form of a list, and produces a new list. This allows the programmers to take a more imperative approach than the `Map` and `Reduce` operations.

Listing 4.2 shows an example of how to use the `Apply` operation to filter a list.

```
let filteredData =
  ifDataset.Apply
    <@ fun (x : (string * int) list) ->
      x |> List.filter (fun y -> (fst y).Contains("a"))
    @>
    "StringInt"
```

Listing 4.2: Apply operation to filter list

Map

The `Map` operation applies the given expression to each element in the chunk and is applied lazily. The map operation will always produce the same number of output elements as the number of input elements.

Listing 4.3 shows an example of how to apply a `Map` operation.

```
let mappedData =
  ifDataset.Map
    <@ fun (x : (string * int)) -> (fst x, ((snd x) % 7))
    @>
    "StringInt"
```

Listing 4.3: Map operation to manipulate values

Reduce

Unlike the other operations the `Reduce` operation is not applied lazily, but instead requires that the all chunks have been through the prior operations. This is because the results of the previous operations must be partitioned and merged by key before the reduce operation can commence.

Before the `Reduce` operation is applied, the IFMapReduce worker separates values to be reduced by key, where key refers to the first entry of a tuple and value is the second entry. Listing 4.4 illustrates a reduce operation that reduces integers by calculating their sum.

The `Reduce` operation saves its output to HDFS, therefore the operation must be supplied with a file path specifying where to place the output. One of the reasons to save the output to HDFS is for the output to be repartitioned into new chunks, to ensure a reasonable data size. Besides a file path the user also needs to specify the amount of reducers, such that the intermediate data can be partitioned accordingly.

`Reduce` operations return a new `IFDataset` with an empty operation list, this is because the new dataset is based on the file generated by the reduce operation.

```
let reducedValues =
  mappedValues.Reduce
    <@ fun( x : string, y : int list ) -> (x, (List.sum y)) @>
    "StringInt"
    "/output/file/path.postfix"
    1
```

Listing 4.4: Reduce operation to add integers

Combine

While the `Reduce` operations is applied to the entire dataset, the `Combine` operation is a reduce operation that is applied lazily on the local chunk, meaning that it can be chained with the other local operations. Like the `Reduce` operation the worker separates values by key such that the expression should only focus on reducing values.

Performing a combine operation before the actual reduce operation can improve runtime by reducing the IO needed to prepare data for the reduce operation[18]. Unlike the `Reduce` operation `Combine` does not output its results to HDFS.

```
let combinedValues =
  mappedValues.Combine
    <@ fun( x : string ) ( y : string list ) ->
      let combinedString = y
        |> List.reduce(fun (a : string) (b : string) -> a + " " + b
          )
      (x, combinedString) @>
    "StringString"
```

Listing 4.5: combine operation to concatenate strings

Cache

The `Cache` operation instructs the worker to store the current local chunk in memory such that it can be accessed without having to read the data from HDFS. To allow this, the master keeps track of which workers cache what data, as cached data is not replicated throughout the cluster.

Like the `Reduce` operation the `Cache` operation initiates computation and returns a dataset with an empty operations list.

Store

The `Store` operation is used to specify that the current dataset should be saved to permanent storage. It is the only operation that does not return a new `IFDataset`,

but instead initiates computation and saves the results to HDFS.

The operation does not facilitate merging or sorting data, meaning that data is appended to the specified file directly from each task in arbitrary order.

4.3 Broadcast Functions

IFMapReduce allows users to broadcast lambda functions to all workers in the cluster, these functions are kept in a global session that is independent of the `IFDatasets`.

Listing 4.6 shows how to add and apply a broadcast function, any IFMapReduce operation can make use of a `Broadcast function` once added to the session.

To correctly deserialize Broadcast functions, these impose the same datatype limitations as the `IFDataset` operations. Furthermore, the function must be applied using the get method corresponding to the input and output type of the function. In the listing the `getIntToIntExpression` method is used because the `square` function is of the type `int -> int`.

```
let squareIntegers = <@ fun (x : int) -> x * x @>
BExpression.addFunction squareIntegers "square" "localhost" 7040
let reducedValues =
  mappedValues.Map
    <@ fun (x : string) (y : int) ->
      (x, ((BExpression.getIntToIntExpression "square") y))
      @>
    "StringInt"
    "/user/slave/BroadcastOutput.txt"
    1
```

Listing 4.6: example of using a broadcast function

Making use of Broadcast functions breaks some of the principles of parallel computing, as each task can no longer be considered a single entity, but is dependent on the broadcast function. Furthermore, broadcast functions may be replaced at runtime, meaning that that it is possible for the behavior of the application to be changed at runtime. Currently it is possible for broadcast expression to be overwritten from within an `IFDataset` operation, meaning that the broadcast expressions can be replaced from the worker side.

Broadcast functions are subject to race conditions, as the IFMapReduce prototype does not provide any facilities to detect or handle these race condition.

Broadcast functions can be used to upload standard functions onto the cluster, this could be a check to determine if strings have a specific structure, if multiple analysis jobs have to perform such check, this only needs to be specified once, meaning that changes to the check can be done in a centralized manner.

CHAPTER 5

EXPERIMENTS

This chapter documents how benchmarks have been used to compare the Spark, IFMapReduce and FMapReduce frameworks. Section 5.1 describes the cluster setup used in the benchmarks. Section 5.2 compares IFMapReduce and Spark using a Pagerank algorithm, while section 5.3 compares Spark, IFMapReduce and FMapReduce using Wordcount algorithms. General observations gained during the benchmarks are summarized in section 5.4.

5.1 Cluster setup

The benchmarks were performed on a cluster containing three nodes, one acting as a dedicated master while the remaining two as dedicated worker nodes. The OS used on the machines in the cluster were Ubuntu 14.10 64-bit with Mono Jit compiler version 4.0.1. Table 5.1 provides a summary of the resources available to the nodes in the cluster.

	WorkNode 1	**WorkNode2**	**Master**
CPU	Intel core i7 950, 3.07 GHz	Intel core i7-2670QM, 2.2 GHz	Intel Core i5-3340M, 2.7 GHz
RAM	12.212 MB	12.212 MB	3.826 MB
Cores	4 (8 HyperThreads)	4 (8 HyperThreads)	2 (4 HyperThreads)

Table 5.1: Cluster specifications

5.2 Pagerank

A Pagerank algorithm is used to compare Spark and IFMapReduce because it is an iterative algorithm. Both IFMapReduce and Spark uses HDFS as the underlying storage facility.

Spark is optimized for iterative algorithms, as the RDD abstraction keeps intermediate results in memory for faster access. Furthermore Spark supports in-memory shuffle, which means that it does not rely on the underlying file system when redistributing data, whereas IFMapReduce is reliant on HDFS when shuffling data.

5.2.1 Pagerank setup

The Pagerank algorithms were implemented in similar fashion for both frameworks to minimize the implementations effect on the benchmark results. Listings 5.1(Spark) and 5.2(IFMapReduce) shows parts of the Pagerank algorithms used during the benchmarks, for the full algorithms see appendix B. The applications were run from the F# interactive shell and the Spark interactive Scala shell.

The frameworks are compared by running an iterative Pagerank algorithm with ten iterations. To gain stable results, each job was executed five times and the best and the worst performing benchmarks were discarded.

The dataset consists of four dumps of Wikipedia pages[1] appended to each other to form a 7.3gb file. The IFMapReduce job from appendix A was used to extract page titles and outgoing links for each Wikipedia page contained in the file. This collection of titles and links form a 470MB file which was used for both the IFMapReduce and Spark Benchmarks. HDFS was set to use a block size of 64MB resulting in HDFS splitting the 470MB data file into seven chunks.

```
1  val rawFile = sc.textFile(inputFile)
2  var linksAndRanks = rawFile.map(line => {
3    if (line.length > 0) {
4      val splitValues = line.split("\t")
5      if (splitValues.length > 1) {
6        (splitValues(0), splitValues(1).replaceAll(",", ";;"))}
7      else {
8        (line, "1.0")
9      }
10   }
11   else {
12     ("None", "1.0")
```

[1] url: *https://dumps.wikimedia.org/enwiki/20150304/* file1: *enwiki-20150304-pages-meta-current10.xml-p000925001p001325000.bz2* file2: *enwiki-20150304-pages-meta-current13.xml-p002425001p003124998.bz2* file3: *enwiki-20150304-pages-meta-current13.xml-p002425001p003124998.bz2* file4: *enwiki-20150304-pages-meta-current15.xml-p003925001p004825000.bz2*

```
  }
})

for (i <- 1 to iterations) {
  val linksMapped = linksAndRanks.flatMap(
    keyValues => {
      val rankAndLinks = (keyValues._2).split(" ")
      var results = List[(String, String)]()

      if (rankAndLinks.length > 1) {
        val rank : String = rankAndLinks(0)
        val links = rankAndLinks(1).split(";;")
        val length : String = " " + (links.length).toString
        links.foreach{link => {results =  (link : String, rank +
          length) :: results}}
        results = (keyValues._1, rankAndLinks(1)) :: results
      }
      results
    }
  )

  linksAndRanks = linksMapped.groupByKey()
    .map(tuple => {
      val links = tuple._2
      val outLinks = links.filter(entry => entry.contains(";;"))
      var oLinks : String = ""
      outLinks.foreach{outlink : String => oLinks = oLinks +
        outlink}
      val inLinks = links.filter(entry => entry.contains(" "))
      val combinedPages = inLinks.map(linkValues => {
        val splitLink = linkValues.split(" ")
        (splitLink(0).toDouble / splitLink(1).toDouble) }
      ).sum
      val newRank = combinedPages * dampening + (1.0 - dampening)
      (tuple._1, newRank.toString + " " + oLinks)
    }
  )

linksAndRanks.saveAsTextFile("/usr/bench/sparkOutput")
```

Listing 5.1: Pagerank in Spark

```
let dataSet = FInteractiveClient.Dataset.Lines("/usr/bench/
  parsedData", masterIp, (string masterPort))

let mutable dataTuple =
    dataSet.Map
        <@ fun (x : string) ->
            let splitString = x.Split([|"\t"|], System.
```

```
                StringSplitOptions.None)
            if splitString.Length > 1 then
                (splitString.[0], splitString.[1])
            else
                (x, "1.0")
            @>
        "StringString"

let pageMap =
    <@ fun (x : (string * string) list) ->
        let mutable results : (string * string) list = []
        for kvPair in x do
            if (snd kvPair).Length > 4 then
                let rankAndLinks = ((snd kvPair).Split([|" "|],
                  System.StringSplitOptions.RemoveEmptyEntries))
                if rankAndLinks.Length > 1 then
                    let rank = rankAndLinks.[0]
                    let links : string array = ((rankAndLinks.[1]).
                      Split([|","|], System.StringSplitOptions.
                      RemoveEmptyEntries))
                    let length = (float links.Length)
                    for link : string in links do
                        results <-
                            (
                                (link.Replace("_", " ")),
                                (rank + " " + string length + ".0")
                            ) :: results
                    results <- ((fst kvPair), (rankAndLinks.[1]))
                      :: results
        results
        @>

let pageReduce =
    <@ fun (key : string, values : string list) ->
        let dampening = 0.85
        let noOfLinks = values.Length
        let mutable combinedPages : float = 0.0
        let mutable outLinks = ""
        for value : string in values do
            let splitValues = value.Split([|" "|], System.
              StringSplitOptions.RemoveEmptyEntries)
            if splitValues.Length > 1 then
                combinedPages <- (((float splitValues.[0]) / (float
                  splitValues.[1])) + combinedPages)
            else
                outLinks <- value
                combinedPages <- combinedPages + 0.0
        let rank = (1.0 - dampening) + dampening * combinedPages
        (key, ((string rank) + " " + outLinks))
```

```
49          @>
50
51  for i = 0 to iterations do
52      let mapIteration =
53          dataTuple.Apply
54              pageMap
55              "StringString"
56      dataTuple <-
57          mapIteration.Reduce
58              pageReduce
59              "StringString"
60              ("/usr/bench/testOutput" + (string i))
61              8
```

Listing 5.2: Pagerank in IFMapReduce

5.2.2 Pagerank Results

The results of the Pagerank benchmarks shows that spark is roughly six times faster than IFMapReduce, as summarized in table 5.2.

	IFMapReduce	**Spark**
Avg. Running time	3572s	542s
Avg. Iteration time	357s	50s

Table 5.2: Overview of Pagerank benchmark results

One of the reasons for this difference in execution time is because Spark is optimized for iterative algorithms like the Pagerank algorithm, and is built to avoid IO operations where possible. In contrast IFMapReduce is reliant on HDFS when shuffling data, resulting in an IO overhead.

To better understand where IFMapReduce can be optimized, it logs how it uses its time during each iteration. Each iteration can be viewed as having a map stage and a reduce stage and figure 5.1 and figure 5.2 illustrates the workload for these stages respectively.

As the figures illustrate, the map and reduce stages spend more than half their time interacting with HDFS, introducing significant overhead compared to Spark's in-memory shuffle.

Besides overhead when interacting with HDFS, IFMapReduce has some scheduling overhead. As the cluster was able to execute all tasks in parallel the average iteration time of 357s can be compared with the average task execution times of 123s for map and 121s for reduce, showing that IFMapReduce uses 113s on job preparation and other scheduling tasks.

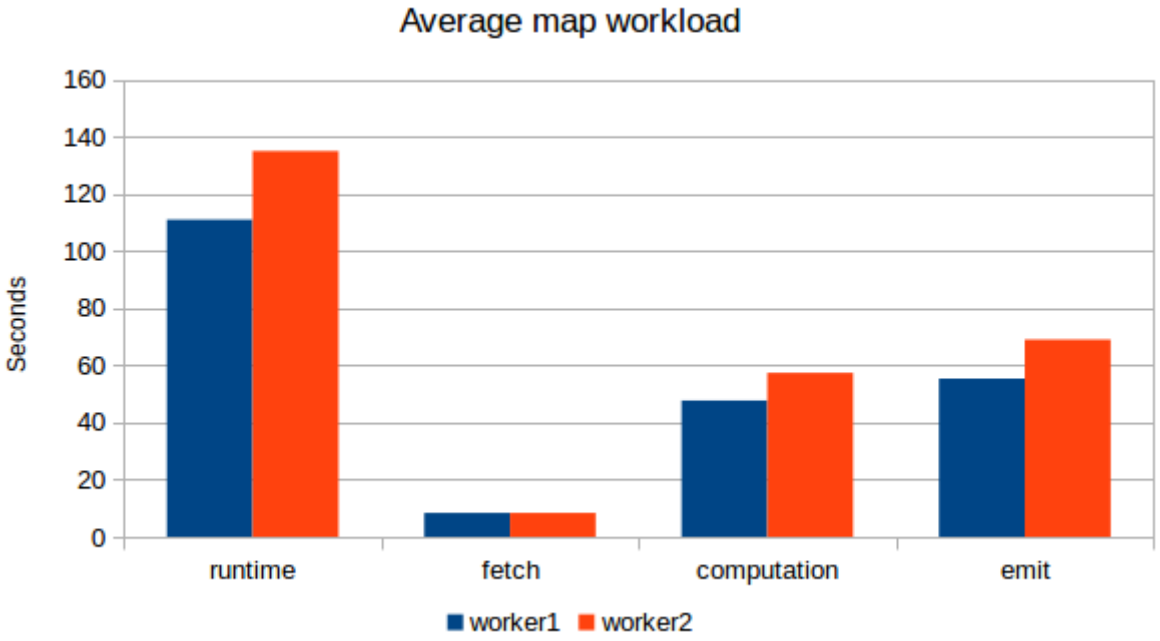

Figure 5.1: workload during map stage

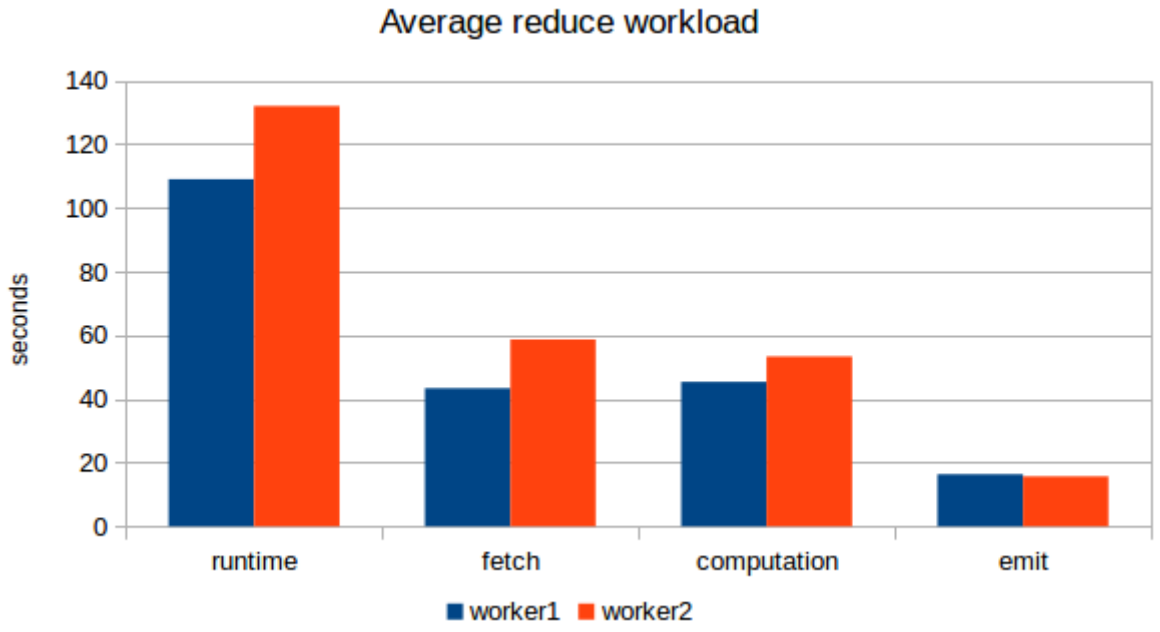

Figure 5.2: workload during reduce stage

The largest scheduling overhead observed was the time taken to initiate a request, where the master must interact with HDFS to gather file metadata before it can generate the tasks for the request.

Compared to spark who schedules all the tasks for the iterations at the beginning of the job, IFMapReduce start a job each time the `Reduce` operation is applied. This means that instead of scheduling one job, IFMapReduce schedules one job per iteration, resulting in IFMapReduce repeating the scheduling overhead ten times.

Figure 5.3 illustrates the workload distribution for the average iteration in percentages. This chart shows that IFMapReduce only spent 30% of each iteration actually computing the pagerank.

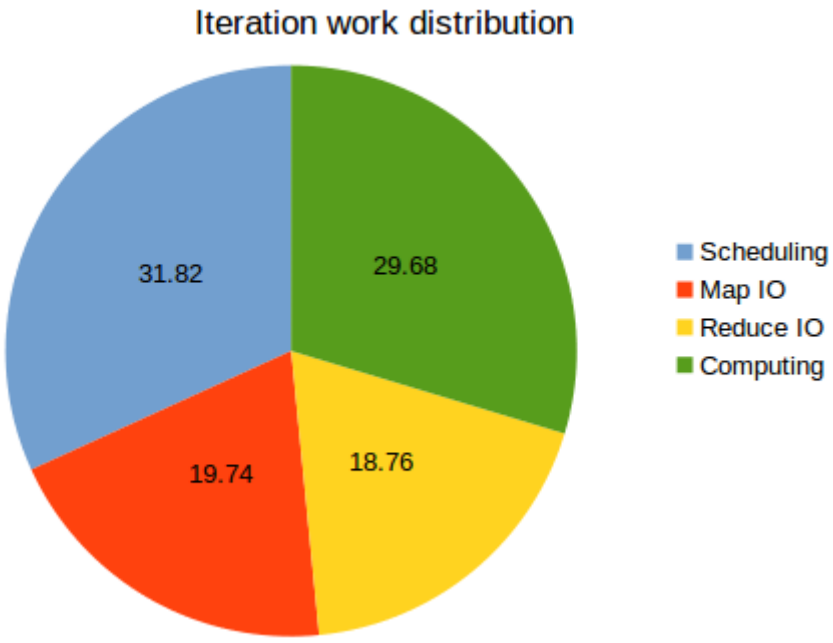

Figure 5.3: Iteration workloads (percentages)

The general observations gained from performing these benchmarks are that optimizations must be made to the IFMapReduce prototype, to lower the amount of IO operations and lessening the scheduling overhead.

5.3 Wordcount

This section uses benchmarks of Wordcount jobs to compare the FMapReduce, Spark and IFMapReduce frameworks. As opposed to the Pagerank algorithm, the Wordcount algorithm is a single pass algorithm.

These benchmarks are mainly performed to compare IFMapReduce with its predecessor FMapReduce, to see how the changes have affected the work distribution and running time.

5.3.1 Wordcount Setup

Four benchmarks were performed, one for Spark using the algorithm shown in listing 5.3, one for FMapReduce using the algorithm in listing 5.4 and two in IFMapReduce. To better compare the performance of the IFMapReduce framework with the FMapReduce framework, two benchmarks were performed using the algorithm in listing 5.5, one with the combine operation and one without.

```
val file = sc.textFile ( "/usr/bench/data1" )
val counts = file.flatMap(line => line.split(" ")).map(word => (
  word ,1)).reduceByKey(_ + _)
counts.saveAsTextFile("/usr/bench/sparkWCOut")
```

Listing 5.3: Wordcount in Spark

```
type fmapreducetester() =
  inherit FMapReduce()
    override this.distributedMap (key : string) (value : string) =
      for line in key.Split('\n') do
        if line.Length > 0 then
          for word in line.Split(' ') do
            if word.Length > 0 then
              this.yieldKV word (string 1)

    override this.distributedReduce (key : string) (value : string
      array) =
      this.yieldKV key ((value.Length).ToString())
```

Listing 5.4: Wordcount in FMapReduce

```
let iDataset = FInteractiveClient.Dataset.Words("/usr/bench/data1",
   "192.168.2.4", "7040")
let mappedValues = iDataset.Map
                                  <@ fun (x : string) -> (x, 1) @>
                                  "StringInt"
let reduceFunc = <@ fun (x : string, y : int list) ->
                                       (x, (List.sum y ))
                                       @>
let combinedValues = mappedValues.Combine reduceFunc "StringInt"
let wordReduce = combinedValues.Reduce
                                  reduceFunc
                                  "StringInt"
                                  "/usr/bench/WordcountOut"
```

```
13 16
```

Listing 5.5: Wordcount in IFMapReduce with combine operation

The benchmarks were performed on a 1GB file containing Wikipedia pages[2]. For the implementations using HDFS(Spark and IFMapReduce) the block size was set to 64MB resulting in 15 chunks. For DDFS the chunk size was set to 32MB resulting in 12 chunks because of the way DDFS compresses data, meaning that the uncompressed chunks were averagely 25% bigger for DDFS.

5.3.2 Wordcount results

This section presents and discuss the results of the Wordcount benchmarks.

The average runtime of the Wordcount jobs are presented in table 5.3 and illustrated in figure 5.4. The figures show that Spark had the shortest runtime completing the Wordcount job in 44 seconds. The results also show that use of the combine operation almost halved the running time of IFMapReduce. Comparing IFMapReduce with its predecessor FMapReduce a slight increase in running time was observed, but the options to add combine stages makes the IFMapReduce framework superior.

IFMapReduce	**IFMapReduce + Combine**	**FMapReduce**	**Spark**
722s	368s	499s	44s

Table 5.3: Average runtime in seconds

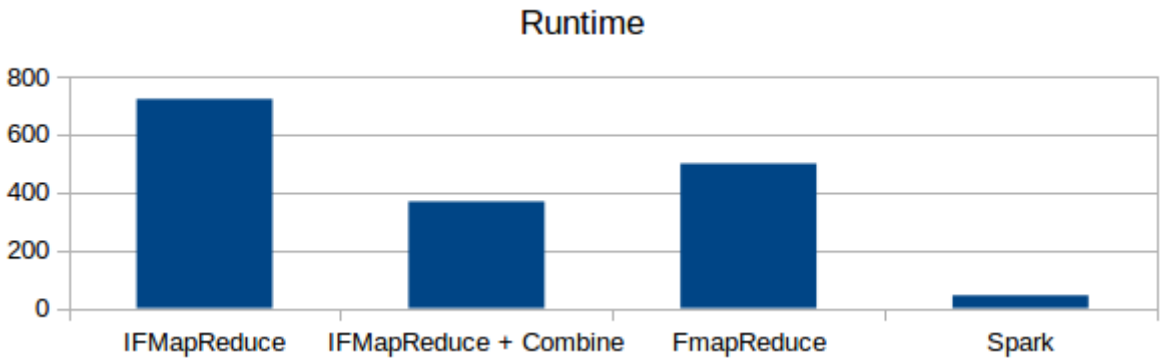

Figure 5.4: Average runtime

[2]url: *https://dumps.wikimedia.org/enwiki/20150304/ file: enwiki-20150304-pages-meta-current9.xml-p000665001p000925000.bz2*

Figure 5.5 illustrates the difference in time spent during the map and reduce stages for the IFMapReduce and FMapReduce frameworks. As the figure shows, IFMapReduce with the combine operation improved the running time of the reduce stage to be several times faster than both FMapReduce and IFMapReduce without the combine operation.

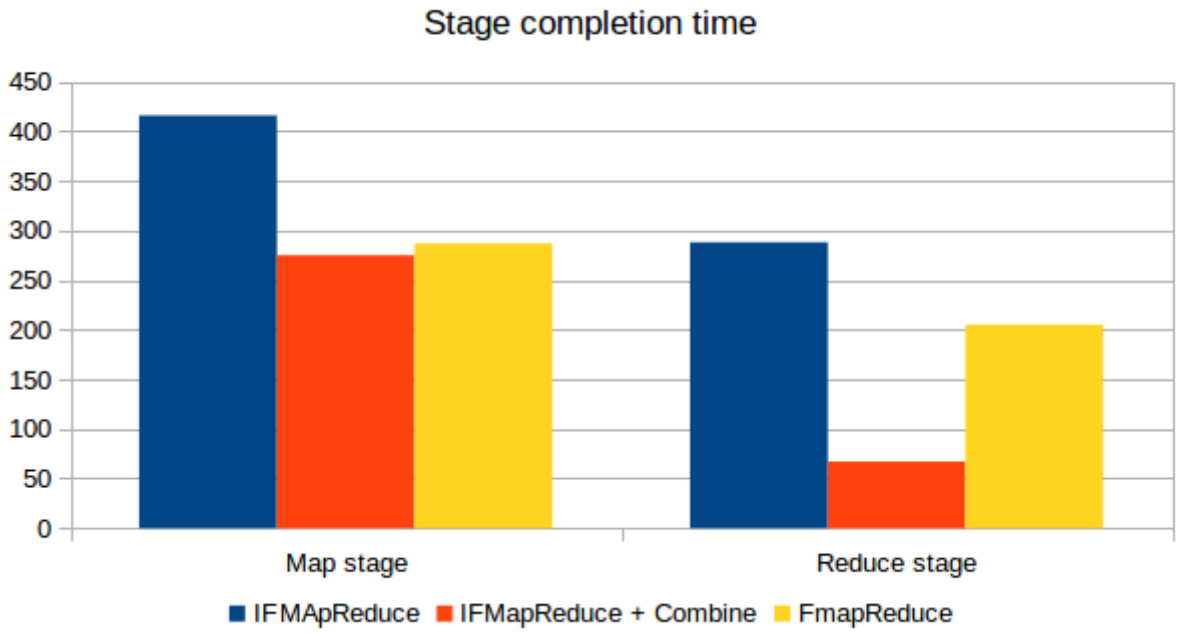

Figure 5.5: Stage completion time

To better understand the observed difference, the IFMapReduce and FMapReduce framework measured the work distribution of each stage. Figure 5.6 illustrates the work distribution of the map (and combine) stage. The figure shows that IFMapReduce used more time interacting with HDFS than FMapReduce did when interacting with DDFS, this difference was even bigger considering that the average chunk size was bigger.
Before outputting data to the underlying file systems, both IFMapReduce and FMapReduce converts the data to strings. FMapReduce does this conversion during the map stage while IFMapReduce does it in a separate stage, showing that the IFMapReduce prototype has introduced some overhead during this stage compared to its predecessor. However, the IFMapReduce framework makes up for this by being faster when partitioning the data.

Figure 5.7 illustrates the work distribution for the reduce stage, this figure can be misleading as each of the distributions illustrated made use of different amounts

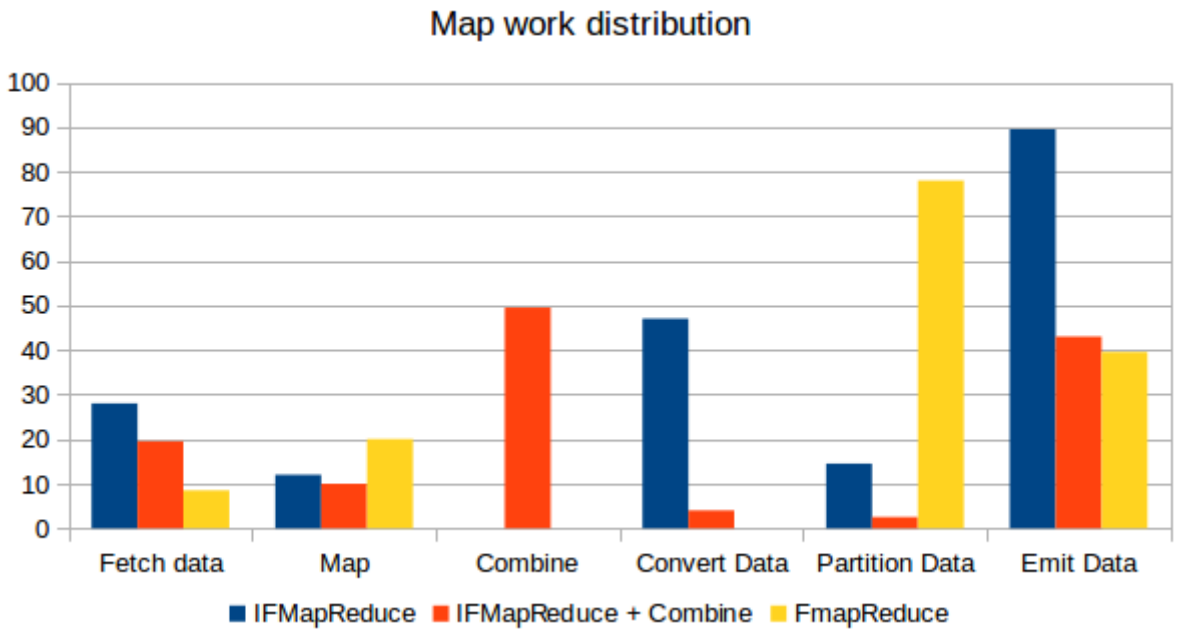

Figure 5.6: Work distribution of the map (and combine) stage

of reduce tasks. IFMapReduce without the combiner used 16 reduce tasks, while IFMapReduce with the combiner used eight and FMapReduce used 12 reduce tasks.

This discrepancy of reduce tasks meant that the running time for IFMapReduce without the combine should be doubled when comparing it to IFMapReduce with the combine operation as the cluster can only perform eight tasks simultaneously.

Like the map stage the results show that FMapReduce were the faster option when interacting with the underlying file system, when considering that IFMapReduce with the combiner operated on less input data.

Considering both figures 5.6 and 5.7 FMapReduce proved to be faster when performing reduce operations. While IFMapReduce with the combiner had the shortest reduce step, the time spent during the combine step was slower than the reduce step of FMapReduce, however the reduction in IO more than made up for this delay. Changes to the way IFMapReduce aggregates values by key is suspected to be the main cause for the differences in running time, as an example IFMapReduce sorts the data as part of the reduce action, whereas FMapReduce sorts the data during the Fetch data action.

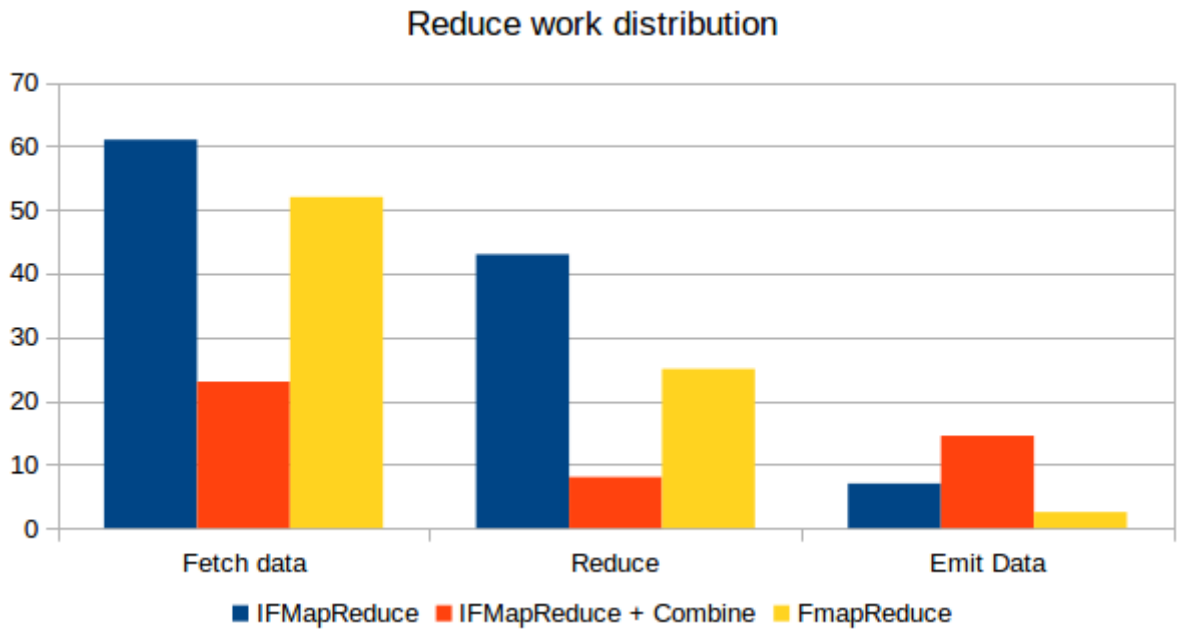

Figure 5.7: Work distribution of the reduce stage

5.4 General observations

This section discusses the results gained from performing the experiments. During the Pagerank experiments we observed a large scheduling overhead, this lead to several optimizations before performing the Wordcount experiments. These optimizations resulted in an average reduction in scheduling time of 87.2 seconds.

As the IFMapReduce Pagerank algorithm schedules as ten separate jobs, this reduction in overhead would be equal to 872 seconds, or nearly 15 minutes overall runtime reduction, which is roughly a fourth of the measured running time.

The benchmark results show that using Code Quotations is a viable option for distributed computing. Observing the map time illustrated in figure 5.6, it shows that the time taken to map using Code Quotations is comparable to mapping using a loaded precompiled assembly.

Based on these observations, it is the IFMapReduce frameworks itself that causes it to be slower than the FMapReduce framework, and not because IFMapReduce makes use of Code Quotations.

CHAPTER 6

DISCUSSION

This chapter discuss how to compare distributed programming frameworks in section 6.1, while section 6.2 presents ideas on how to improve upon IFMapReduce and its prototype.

6.1 Comparison metrics

The diversity of methods used to perform Big Data analysis makes it difficult to compare these methods objectively, many tools have specific use cases where they excel but fall short on other use cases. Some solutions focus on optimizing running time, while other focus on usability and making it easier to specify distributed jobs.

While runtime is a measurable metric, the diversity of methods makes it hard to find a fair way to compare these using running time. Some methods may prove to have shorter running times when doing iterative computations while others may be faster for scanning jobs. This diversity means that care must be taken when deciding how and which methods to compare using running time.

The task of writing distributed applications are generally considered hard[4, 16] and usability is one of the focus points when creating tools to help programmers write distributed applications. Amongst these solutions are programming languages focused on distributed computing, extensions to existing languages, programming frameworks, etc. Usability is however a subjective metric as it depends on the past experience of the programmer. Some tools try to make the approach to distributed computation more declarative, while other tools try to mimic existing paradigms to draw on the programmers experience.

The MapReduce approach attempts to facilitate distributed computations by providing the programmers with a well defined API. While this may limit the flexi-

bility of the approach, it provides usability by introducing a "correct" way of specifying distributed applications.

IFMapReduce attempts to perform Big Data analysis using the F# syntax while providing a well defined API for distributed programming.

6.2 Future work

This section describes ideas that could improve the usability and performance of the IFMapReduce framework.

6.2.1 Improved type handling

The IFMapReduce prototype imposes several limitations on the data types that expressions can interact with and data types must be explicitly declared for both input and output.

Having to specify data types can prove to be a distraction for the user, especially as F# determines data types implicitly. Having IFMapReduce handle types implicitly would make it easier to write applications for use with IFMapReduce, as operations could be specified by only providing the expression as parameter.

The problems with explicit typing is most clear when using broadcast functions, where the function must be retrieved using a specific get method. The IFMapReduce prototypes uses a list of union types to store broadcast functions, this approach imposes limitations on how the expressions can be retrieved. Instead of using a union type IFMapReduce should use a wrapper class and make use of a class hierarchy to make retrieval of the expressions uniform.

Besides making IFMapReduce able to use data types implicitly, the utility of the framework should be increased by allowing clients to send type metadata, such that custom data types can be supported.

6.2.2 Shipping expressions

To ship expressions the IFMapReduce prototype makes use of FsPickler[22], this is because the standard F# libraries does not supply sufficient support for shipping quoted expressions.

One of the main issues related to shipping expressions is to correctly deserialize the expression. F# provides a wildcard pattern that tries to match any input, while FsPickler can use the wildcard pattern when deserializing simple expression types such as `Int -> Int` expressions, this option does not allow more complex datatypes such as `Tuples`. Meaning that using the wildcard pattern in conjunction with FsPickler is insufficient for most use cases of IFMapReduce.

One of the reasons that the IFMapReduce prototype only supports a limited amount of expression types is that .NET types are not serializable, thus the expression type cannot be shipped with the expression. IFMapReduce would benefit from a method to ship the quoted expressions with associated type metadata, not only because this could allow for greater freedom when writing IFMapReduce applications. but also because several type checks and conversions can be avoided by the IFMapReduce worker.

In general, the base support in F# for shipping expressions and type metadata is found lacking, and IFMapReduce would need some extended abstraction for quoted expressions like the MBrace CloudWorkflows described in section 2.6. This abstraction should be able to contain the metadata needed to simplify the shipping of expressions.

6.2.3 Fault tolerance

Fault tolerance is an important aspect of distributed applications as there are more points of failure compared to single node applications. The IFMapReduce prototype does not provide any fault tolerance, meaning that if a single tasks fails at any worker, the master will wait for that task to finish and not reschedule said task.

While the master is able to detect failed workers and have the metadata required to restart tasks, it does not make use of this information. Besides not handling failures, IFMapReduce does not inform the client about failures, meaning that the user will not be able to determine the cause of a failure.

6.2.4 Better debugging facilities

While users of IFMapReduce can make use of the debugging facilities available in F# editors, this only ensures a correct syntax. Currently IFMapReduce provides no facilities for forwarding errors from expressions executed on a worker to the client, meaning that the client is never informed if errors happen. Furthermore, while some code quotations may be run locally, FsPickler may not be able to deserialize the code because it imposes stricter semantics.

One way for IFMapReduce to help programmers could be to introduce .NET Attributes like the `ReflectedDefinition` attribute. This would allow IFMapReduce to produce warnings or errors when the programmer attempts break the conventions specified in the attribute.

6.2.5 In-memory shuffle

The IFMapReduce prototype is heavily Dependant on HDFS to handle intermediate data, however as the results of the benchmarks (sections 5.2 and 5.3) show, this

introduces overhead to the point where more time is spent interacting with the file system than doing computation.

To avoid this overhead, IFMapReduce should be able to move data directly between workers, meaning that the master must predetermine where to place data before initiating the requests. This process includes determining the workload and size of each data partition, and should also account for locality to minimize communication in the cluster.

Becoming less dependent on HDFS would allow the Reduce operation to be lazy, meaning that it could be chained with the other operations. This would allow the Pagerank algorithm described in section 5.2.1 to be scheduled as a single job rather than ten, removing the overhead associated with scheduling multiple jobs.

6.2.6 GPU utilization

The processing power of GPUs have been increasing drastically over the last few years and even the less expensive GPUs have become powerful highly parallel processing units. The benefit of GPUs over CPU are that the GPUs have access to magnitudes more cores compared to the standard CPU's and as Big Data analysis focus on parallel computation, these jobs should fit well with GPU architectures.

MarsHadoop[8] has proven how utilizing GPUs can be used to increase the performance of MapReduce jobs, and while IFMapReduce is not targeted at batch jobs, some operations might still benefit from utilizing GPUs to improve runtime performance.

F# provides several libraries for GPU programing, amongst these are Alea.CUDA[25], FSCL[2] and Brahma.FSharp[14] who all makes use of F#'s Quoted Expressions to generate GPU code. As IFMapReduce is based on shipping Quoted Expressions, these libraries provides ample examples that IFMapReduce could be extended to utilize GPUs. However, the libraries requires that the Quoted expressions is formed to fit specific APIs, meaning that expressions written for the GPU cannot be executed on the CPU.

Both Alea.CUDA and FSCL requires that the quotations are marked with the `[<ReflectedDefinition>]` attribute, which means that the quotation cannot contain the splicing operator, thus preventing merging general purpose expressions into predefined GPU expressions. While Brahma.FSharp supports the splicing operator it sets several limitations to the supported data types, most notable in the context of IFMapReduce Brahma.FSharp does not support `Tuples`.

6.2.7 Runtime Estimation

Rather than relying on HDFS and the user to determine the amount of tasks to schedule, IFMapReduce should be able to choose an optimal amount of tasks based

on workload estimation. Choosing the number of workers based on workload would complement the idea of in-memory shuffle described in section 6.2.5, as this would further remove the reliance on the underlying file system.

Besides choosing how many workers to use, runtime estimation could also compliment the GPU utilization idea described in section 6.2.6. While GPUs are faster at certain tasks, some overhead is introduced when moving the data to GPU Memory[28]. Therefore SPRAT[28] (Stream programming with runtime auto-tuning) proposes a solution that chooses which processing unit to use based on the estimated workloads.

SPRAT uses the linear model shown in equation 6.1 to estimate kernel execution time. In the equation p refers to the processor and k_i refers to a kernel i where a kernel is a processor running a specified kernel function. The equation states that the execution time is equal to the startup time required by the processor to launch the kernel plus the number of output elements divided by the throughput.

$$T_{p,k_i(Execution\ time)} = \frac{D_{k_i(Output\ stream\ elements)}}{B_{p,k_i(Effective\ throughput)}} + S_{p,k_i(Startup\ time)} \tag{6.1}$$

While SPRAT gives a model for when to move computations between processors, it does not provide information on how to estimate the runtime parameters needed in the equation.

There are several methods and tools to perform runtime analysis, most approaches includes modeling the application and using said model to determine the running time. One of the more simple methods is the Implicit Path Enumeration Technique(IPET)[19] that models code as Control Flow Graphs as illustrated in figure 6.1

```
1  k = 13
2  i = 0
3  while i < k do
4    if(q)
5      do_something()
6    else
7      do_other()
8      i++
9    i++
10 print_result()
```

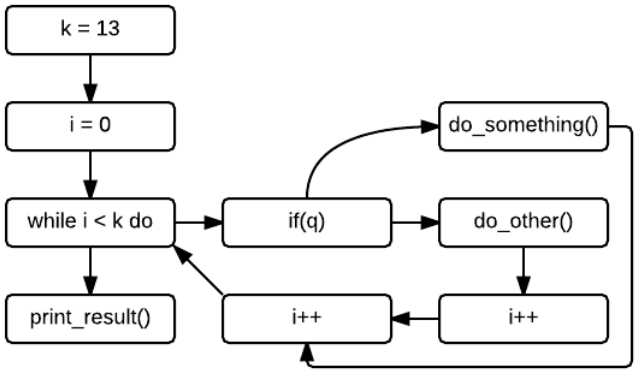

Figure 6.1: Example of Control Flow

While the IPET method is relatively simple, it is also imprecise, and depending on the use case more precise estimation is needed.

Some methods attempt to gain precise runtime estimations by attempting to model the runtime environment, one such tool is TetaJ(Tool for Execution Time Analysis of Java bytecode)[13]. TetaJ not only models the application, but also the JVM and the Hardware of the runtime environment. The hardware model simulates how JVM instructions are executed on the hardware, while the JVM model simulates how the JVM handles bytecode instructions. The application is modeled as a control flow model where each location corresponds to a bytecode instruction. TetaJ uses the UPPAAL model checker to simulate the execution of the program and calculate the execution time.

Looking at the context for IFMapReduce or cluster computing in general, then the clusters may be heterogeneous, making approaches like TetaJ hard to apply, as multiple Hardware models would be required. Besides being heterogeneous, cluster components may fail resulting in unpredictable delays.

IFMapReduce jobs are written in F# which is higher-order language, meaning that functional arguments must be augmented to account for the cost of their application. Furthermore F# provides features that can make evaluation lazy, meaning that not only must functions be augmented with application cost, but also with a description of the context as to determine if lazy expressions will be evaluated.[26].

In the context of determining how many tasks to start or whether to move computations to a GPU determining the amount of computations should be sufficient, therefore an IPET approach could be suitable.

In F# a Code Quotations is at its core an Abstract Syntax Tree(AST), this means that generating a Control flow graph from a code quotation is relatively simple and estimation can be made by creating a parser for the AST. F# provides the `Quotations.Patterns` module which can be used to build a parser for the AST by using pattern matching.

A prototype of a runtime estimation using `Quotations.Patterns` can be found in appendix D, this prototype is unfinished in that it only recognizes a subset of the AST nodes likely to appear, but it can be used for simple expressions.

Runtime estimation of the IFMapReduce operations should be performed client side, such that each expression is only evaluated once and to avoid having the master deserialize the expressions.

CHAPTER 7

CONCLUSION

This report has investigated the use of F# Code Quotations as the mechanism for shipping code from the F# Interactive shell to perform interactive analysis of Big Data on the .NET platform.

To evaluate the viability of using Code Quotations as distribution mechanism we have:

- Implemented a prototype of a programming framework called IFMapReduce which facilitates the distribution and execution of Code Quotations in a cluster.
- Used benchmarks to compare the IFMapReduce prototype with the Spark and FMapReduce frameworks.

There are several methods for analyzing Big Data, each with their own focus and specialization to fit certain use cases. Historically frameworks such as MapReduce frameworks have focused on batch jobs, while recently frameworks such as Spark has made it possible to perform interactive analysis for a more flexible approach to Big Data analysis.

IFMapReduce is a programming framework that facilitates interactive analysis of Big Data through the F# Interactive Shell. IFMapReduce is an extension of FMapReduce, and inherits its Master/Slave architecture.

IFMapReduce jobs are specified using IFDatasets and F# Code Quotations through the interactive shell. An IFDataset is an abstraction over a file and a list of operations that should be applied to that file. IFDataset operations are evaluated lazily when possible, meaning that computation is not initiated until required, thus most IFDataset operations return a new IFDataset allowing the lazy operations to be chained together. IFDatasets initiate jobs by forwarding a request to the master

containing the list of operations specified in the IFDataset and metadata describing the input. Based on the supplied metadata the master schedules a series of tasks, which are distributed to the workers for execution.

The IFMapReduce prototype has been compared to the Spark framework and the FMapReduce prototype using Pagerank and Wordcount benchmarks. The results show that the IFMapReduce prototype has a running time comparable to that of the FMapReduce prototype, but that the IFMapReduce prototype is slower than Spark. By measuring the work distribution of the IFMapReduce prototype we have determined that IO is the main time consumer, this means that IFMapReduce would benefit from moving data directly between nodes without interacting with the underlying storage facilities, in a similar fashion to Sparks in-memory shuffle. The Benchmarks also show that the use of Code Quotations are viable as a facility for code distribution as these had little effect on the running time.

The report finds that F# Code Quotations are viable for interactive analysis of big data, and proposes ideas to utilize the Code Quotations in regards to Runtime estimation and GPU utilization.

BIBLIOGRAPHY

[1] CERN. Computing. `http://home.web.cern.ch/about/computing`. Visisted: 27/5/2015.

[2] Gabriele Cocco. Fscl.compiler. `http://fscl.github.io/FSCL.Compiler/`. visisted: 24/5/2015.

[3] Nokia Corporation. Tutorial. `http://disco.readthedocs.org/en/latest/start/tutorial.html`. visisted: 8/1/2014.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[6] Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. Mbrace: cloud computing with monads. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, page 7. ACM, 2013.

[7] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM, 2011.

[8] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K Govindaraju. Mars: Accelerating mapreduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):608–620, 2011.

[9] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`. visited: 20/5/2015.

[10] The Apache Software Foundation. Mapreduce tutorial. http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0. visisted: 21/5/2015.

[11] The Apache Software Foundation. Spark examples. https://spark.apache.org/examples.html. visisted: 8/1/2014.

[12] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, pages 129–158. Springer, 2003.

[13] Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow, and Bent Thomsen. Wcet analysis of java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 30–39. ACM, 2011.

[14] Semyon Grigorev. Brahma.fsharp. https://sites.google.com/site/semathsrprojects/home/brahma-fsharp/. visisted: 24/5/2015.

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[16] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[17] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 987–994. ACM, 2009.

[18] Christian M Jensen and Lars H Larsen. Fmapreduce: A survey of big data analysis methods, 2015.

[19] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, pages 88–98. ACM, 1995.

[20] Microsoft. Code quotations (f#). https://msdn.microsoft.com/en-us/library/dd233212.aspx. Visited: 2/6/2015.

[21] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. Disco: a computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 84–89. ACM, 2011.

[22] Nessos. Fspickler : A fast .net object serializer. `http://nessos.github.io/FsPickler/index.html`. visisted: 17/5/2015.

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

[24] Kevin Wilfong Pamela Vagata. Scaling the facebook data warehouse to 300 pb. `https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/`. Visited: 25/11/2014.

[25] QuantAlea. Alea gpu. `http://quantalea.com/`. visisted: 24/5/2015.

[26] David Sands. Complexity analysis for a lazy higher-order language. In *Functional Programming*, pages 56–79. Springer, 1990.

[27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[28] Hiroyuki Takizawa, Katsuto Sato, and Hiroaki Kobayashi. Sprat: Runtime processor selection for energy-aware computing. In *Cluster Computing, 2008 IEEE International Conference on*, pages 386–393. IEEE, 2008.

[29] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[30] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[31] Dennis Fetterly Mihai Budiu Úlfar Erlingsson Pradeep Kumar Gunda Jon Currey Yuan Yu, Michael Isard. Dryadlinq a system for general-purpose distribution data-parallel computing. `research.microsoft.com/en-us/projects/dryadlinq/dryadlinq-osdi.pptx`. visisted: 8/1/2014.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[33] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

APPENDIX A

WIKI-XML PARSER

```
#I "/home/chr/Documents/git/mapdeduce/FInteractiveTest/
  FInteractiveTest/bin/Debug/"
#I "/home/chr/Documents/git/mapdeduce/FInteractiveLib/
  FInteractiveLib/bin/Release"
#I "/home/chr/Documents/git/mapdeduce/Expulsion/Expulsion/bin/Debug
  /"

#r "System"
#r "System.Net"
#r "FsPickler.dll"
#r "FExpression.dll"
#r "FInteractiveLib.dll"
#r "Expulsion.dll"
open System
open System.IO
open Nessos.FsPickler
open FInteractive.client
open Microsoft.FSharp.Quotations
open System.Net.Sockets
open FInteractive.Broadcast

let masterIp = "192.168.2.4"
let masterPort = 7040

let fileWriter : StreamWriter = File.CreateText("/home/chr/Desktop/
  RealBench1.csv")
fileWriter.WriteLine("Start bench," + DateTime.Now.ToLongTimeString
  ())
let rawFile = FInteractiveClient.Dataset.Lines("/usr/bench/
  bigdatafil", masterIp, (string masterPort))
```

```
25 let iterations = 9
26
27 let extractTitle =
28     <@ fun (x : string) ->
29         let startIndex = (x.IndexOf("<title>") + "<title>".Length)
30         x.Substring(startIndex, (abs (x.IndexOf("</title>") -
           startIndex)))
31         @>
32 BExpression.addFunction extractTitle "extractTitle" masterIp
    masterPort
33
34 let isNotLink =
35     <@ fun (x : string) ->
36         let start = if x.StartsWith("[[") then 2 else 1
37         let length = x.Length
38         if length < start + 2 || length > 100 then
39             1
40         else
41             if x.Contains(":") then
42                 1
43             else
44                 if x.Contains(",") then
45                     1
46                 else
47                     if x.Contains("&") then
48                         1
49                     else
50                         let firstChar = char (x.Substring(start, 1)
                            )
51                         match firstChar with
52                             | '#' -> 1
53                             | ',' -> 1
54                             | '.' -> 1
55                             | '&' -> 1
56                             | '\\' -> 1
57                             | '-' -> 1
58                             | '{' -> 1
59                             | _ -> 0
60         @>
61 BExpression.addFunction isNotLink "isNotLink" masterIp masterPort
62
63 let extractLinks =
64     <@ fun (x : string) ->
65         if x.Contains("[") && x.Contains("]") then
66             let mutable links : string list = []
67             let mutable startIndex = 0
68             let mutable index = 0
69             let mutable stage = 0
70             let charArray = x.ToCharArray()
```

```
71              for character in charArray do
72                  if character = '[' && stage = 0 then
73                      startIndex <- index
74                      stage <- 1
75                  if stage = 2 then
76                      if character = ']' then
77                          let link : string = (new System.String(
                              charArray.[startIndex .. index]))
78                          links <- link :: links
79                      else
80                          let link : string = (new System.String(
                              charArray.[startIndex .. (index - 1)]))
81                          links <- link :: links
82                      stage <- 0
83                  if character = ']' && stage = 1 then
84                      stage <- 2
85                  index <- index + 1
86              let mutable extractedLinks = []
87              for link in links do
88                  if ((BExpression.getStringToIntExpression "
                      isNotLink") link) = 0 then
89                      let start = if link.StartsWith("[[") then 2
                          else 1
90                      let mutable endPos = link.IndexOf("]")
91                      let pipePos = link.IndexOf("|")
92                      if pipePos > 0 then endPos <- pipePos
93                      let part = link.IndexOf("#")
94                      if part > 0 then endPos <- part
95                      extractedLinks <-
96                          (link
97                              .Substring(start, endPos - start)
98                              .Replace(" ", "_")
99                              .Replace("\t", "_")
100                             .Replace(",", "")
101                             .Replace("&amp;", "&")) ::
                                extractedLinks
102             match extractedLinks.Length with
103                 | 0 -> ""
104                 | 1 -> extractedLinks.Head
105                 | _ -> extractedLinks |> List.reduce(fun x y -> x +
                        "," + y)
106         else
107             ""
108         @>
109 BExpression.addFunction extractLinks "extractLinks" masterIp
      masterPort
110
111 let titleAndText =
112     rawFile.Apply
```

```
113         <@ fun (x : string list) ->
114             let mutable resultList : (string * string) list = []
115             let mutable currentLine = 0
116             let mutable title : string = ""
117             let mutable text : string = ""
118             let mutable append : bool = false
119             for line in x do
120                 if line.Contains("<title>") then
121                     title <- ((BExpression.
                              getStringToStringExpression "extractTitle")
                              line).Replace("\t", " ")
122                 if line.Contains("<text>") || line.Contains("<text
                      xml:space=\"preserve\">") then
123                     append <- true
124                 if append then
125                     if line.Contains("[") then
126                         text <- (text + line)
127                 if line.Contains("</text>") then
128                     append <- false
129                     resultList <- (title, text) :: resultList
130                     title <- ""
131                     text <- ""
132                 currentLine <- currentLine + 1
133             resultList
134             @>
135             "StringString"
136
137 let mutable titleAndLinks =
138     titleAndText.Map
139         <@ fun (x : (string * string)) ->
140             (fst x, "1.0 " + ((BExpression.
                  getStringToStringExpression "extractLinks") (snd x)))
                  @>
141         "StringString"
142
143 let res = titleAndLinks.Store "/usr/bench/parsedData"
```

Listing A.1: IFMapReduce script used to extract links from wikipedia XML

APPENDIX B

PAGERANK ALGORITHMS

B.1 IFMapReduce

```
#I "/home/chr/Documents/git/mapdeduce/FInteractiveTest/
  FInteractiveTest/bin/Debug/"
#I "/home/chr/Documents/git/mapdeduce/FInteractiveLib/
  FInteractiveLib/bin/Release"
#I "/home/chr/Documents/git/mapdeduce/Expulsion/Expulsion/bin/Debug
  /"

#r "System"
#r "System.Net"
#r "FsPickler.dll"
#r "FExpression.dll"
#r "FInteractiveLib.dll"
#r "Expulsion.dll"
open System
open System.IO
open Nessos.FsPickler
open FInteractive.client
open Microsoft.FSharp.Quotations
open System.Net.Sockets
open FInteractive.Broadcast

let masterIp = "192.168.2.4"
let masterPort = 7040

let fileWriter : StreamWriter = File.CreateText("/home/chr/Desktop/
  FSBench1.csv")
fileWriter.WriteLine("Start bench," + DateTime.Now.ToLongTimeString
  ())
```

```
let iterations = 9

let dataSet = FInteractiveClient.Dataset.Lines("/usr/bench/parsedData", masterIp, (string masterPort))
let mutable dataTuple =
    dataSet.Map
        <@ fun (x : string) ->
            let splitString = x.Split([|"\t"|], System.StringSplitOptions.None)
            if splitString.Length > 1 then
                (splitString.[0], splitString.[1])
            else
                (x, "1.0")
            @>
        "StringString"

let pageMap =
    <@ fun (x : (string * string) list) ->
        let mutable results : (string * string) list = []
        for kvPair in x do
            if (snd kvPair).Length > 4 then
                let rankAndLinks = ((snd kvPair).Split([|" "|], System.StringSplitOptions.RemoveEmptyEntries))
                if rankAndLinks.Length > 1 then
                    let rank = rankAndLinks.[0]
                    let links : string array = ((rankAndLinks.[1]).Split([|","|], System.StringSplitOptions.RemoveEmptyEntries))
                    let length = (float links.Length)
                    for link : string in links do
                        results <-
                            (
                                (link.Replace("_", " ")),
                                (rank + " " + string length + ".0")
                            ) :: results
                    results <- ((fst kvPair), (rankAndLinks.[1])) :: results
        results
        @>

let pageReduce =
    <@ fun (key : string, values : string list) ->
        let dampening = 0.85
        let noOfLinks = values.Length
        let mutable combinedPages : float = 0.0
        let mutable outLinks = ""
        for value : string in values do
            let splitValues = value.Split([|" "|], System.StringSplitOptions.RemoveEmptyEntries)
```

```
66              if splitValues.Length > 1 then
67                  combinedPages <- (((float splitValues.[0]) / (float
                        splitValues.[1])) + combinedPages)
68              else
69                  outLinks <- value
70                  combinedPages <- combinedPages + 0.0
71          let rank = (1.0 - dampening) + dampening * combinedPages
72          (key, ((string rank) + " " + outLinks))
73          @>
74
75  for i = 0 to iterations do
76      fileWriter.WriteLine("Iter" + i.ToString() + " start," +
          DateTime.Now.ToLongTimeString())
77      let mapIteration =
78          dataTuple.Apply
79              pageMap
80              "StringString"
81      Console.WriteLine("iteration: " + string i)
82      dataTuple <-
83          mapIteration.Reduce
84              pageReduce
85              "StringString"
86              ("/usr/bench/testOutput" + (string i))
87              8
88      fileWriter.WriteLine("Iter" + i.ToString() + " done reducing,"
          + DateTime.Now.ToLongTimeString())
89
90
91  fileWriter.WriteLine("finish all ," + DateTime.Now.ToLongTimeString
      ())
92  fileWriter.Close()
```

Listing B.1: IFMapReduce Pagerank algorithm

B.2 Spark

```
1  import java.io.{InputStream, OutputStream, DataInputStream,
     DataOutputStream}
2  import java.nio.ByteBuffer
3
4  import scala.collection.mutable.ArrayBuffer
5  import scala.xml.{XML, NodeSeq}
6
7  import org.apache.spark._
8  import org.apache.spark.serializer.{DeserializationStream,
     SerializationStream, SerializerInstance}
9  import org.apache.spark.SparkContext._
```

```
10 import org.apache.spark.rdd.RDD
11
12 import scala.reflect.ClassTag
13
14 val inputFile = "/usr/bench/parsedData"
15 val iterations = 10
16 val dampening = 0.85
17
18 val startTime = System.currentTimeMillis
19
20 val rawFile = sc.textFile(inputFile)
21 var linksAndRanks = rawFile.map(line => {
22   if (line.length > 0) {
23     val splitValues = line.split("\t")
24     if (splitValues.length > 1) {
25       (splitValues(0), splitValues(1).replaceAll(",", ";;"))
26     } else {
27       (line, "1.0")
28           }
29   } else {
30     ("None", "1.0")
31   }
32 }
33 )
34
35 for (i <- 1 to iterations) {
36   System.out.println("start iter: "+ i + " " + System.
       currentTimeMillis + ":")
37   val linksMapped = linksAndRanks.flatMap(keyValues => {
38     val rankAndLinks = (keyValues._2).split(" ") //Not 0 indexed?
39     var results = List[(String, String)]()
40     if (rankAndLinks.length > 1) {
41       val rank : String = rankAndLinks(0)
42       val links = rankAndLinks(1).split(";;")
43       val length : String = " " + (links.length).toString
44       links.foreach{link => {results =  (link : String, rank +
           length) :: results}}
45       results = (keyValues._1, rankAndLinks(1)) :: results
46     }
47     results
48   }
49   )
50   linksAndRanks = linksMapped.groupByKey()
51                              .map(tuple => {
52                                val links = tuple._2
53                                val outLinks = links.filter(entry
                                    => entry.contains(";;"))
54                                var oLinks : String = ""
```

```
55                                  outLinks.foreach{outlink : String
                                      => oLinks = oLinks + outlink}
56                                  val inLinks = links.filter(entry =>
                                       entry.contains(" "))
57                                  val combinedPages = inLinks.map(
                                      linkValues => {
58                                      val splitLink = linkValues.
                                          split(" ")
59                                      (splitLink(0).toDouble /
                                          splitLink(1).toDouble) }
60                                  ).sum
61                                  val newRank = combinedPages *
                                      dampening + (1.0 - dampening)
62                                  (tuple._1, newRank.toString + " " +
                                       oLinks)
63          }
64                              )
65  System.out.println("end iter: " + i + " " + System.
      currentTimeMillis)
66  }
67
68  linksAndRanks.saveAsTextFile("/usr/bench/sparkOutput")
69  val time = (System.currentTimeMillis - startTime) / 1000.0
70  println("Completed %d iterations in %f seconds: %f seconds per
      iteration"
71      .format(iterations, time, time / iterations))
```

Listing B.2: Spark Pagerank algorithm

APPENDIX C

WORDCOUNT ALGORITHMS

C.1 IFMapReduce

To remove the combiner line 31 can be commented and line 32 changed to use `mappedValues` instead of `combinedValues`.

```
1  #I "/home/chr/Documents/git/mapdeduce/FInteractiveTest/
     FInteractiveTest/bin/Debug/"
2  #I "/home/chr/Documents/git/mapdeduce/FInteractiveLib/
     FInteractiveLib/bin/Release"
3  #I "/home/chr/Documents/git/mapdeduce/Expulsion/Expulsion/bin/Debug
     /"
4
5  #r "System"
6  #r "System.Net"
7  #r "FsPickler.dll"
8  #r "FExpression.dll"
9  #r "FInteractiveLib.dll"
10 #r "Expulsion.dll"
11 open Nessos.FsPickler
12 open FInteractive.client
13 open Microsoft.FSharp.Quotations
14 open System.Net.Sockets
15 open FInteractive.Broadcast
16 open System.IO
17 open System
18
19 let fileWriter : StreamWriter = File.CreateText("/home/chr/Desktop/
     FSBench1.csv")
20 fileWriter.WriteLine("Start bench," + DateTime.Now.ToLongTimeString
     ())
```

```
21
22
23 let iDataset = FInteractiveClient.Dataset.Words("/usr/bench/data1",
     "192.168.2.4", "7040")
24 let mappedValues = iDataset.Map
25                                 <@ fun (x : string) -> (x, 1) @>
26                                 "StringInt"
27 let reduceFunc = <@ fun (x : string, y : int list) ->
28                                     (x, (List.sum y ))
29                                     @>
30
31 let combinedValues = mappedValues.Combine reduceFunc "StringInt"
32 let wordReduce = combinedValues.Reduce
33                                 reduceFunc
34                                 "StringInt"
35                                 "/usr/bench/WordcountOut"
36                                 16
37 fileWriter.WriteLine("finish all ," + DateTime.Now.ToLongTimeString
     ())
38 fileWriter.Close()
```

Listing C.1: IFMapReduce Wordcount algorithm

C.2 FMapReduce

```
1  namespace mapreducetest
2  open FMapReduce.FMR
3  open System
4
5  type fmapreducetester() =
6      inherit FMapReduce()
7          override this.distributedMap (key : string) (value : string
             ) =
8              for line in key.Split('\n') do
9                  if line.Length > 0 then
10                     for word in line.Split(' ') do
11                         if word.Length > 0 then
12                             this.yieldKV word (string 1)
13
14         override this.distributedReduce (key : string) (value :
             string array) =
15             this.yieldKV key ((value.Length).ToString())
```

Listing C.2: FMapReduce Wordcount algorithm

C.3 Spark

```
1  import java.io.{InputStream, OutputStream, DataInputStream,
     DataOutputStream}
2  import java.nio.ByteBuffer
3
4  import scala.collection.mutable.ArrayBuffer
5  import scala.xml.{XML, NodeSeq}
6
7  import org.apache.spark._
8  import org.apache.spark.serializer.{DeserializationStream,
     SerializationStream, SerializerInstance}
9  import org.apache.spark.SparkContext._
10 import org.apache.spark.rdd.RDD
11
12 import scala.reflect.ClassTag
13
14 val startTime = System.currentTimeMillis
15
16 val file = sc.textFile ( "/usr/bench/data1" )
17 val counts = file.flatMap(line => line.split(" ")).map(word => (
     word ,1)).reduceByKey(_ + _)
18 counts.saveAsTextFile("/usr/bench/sparkWCOut")
19
20 val time = (System.currentTimeMillis - startTime) / 1000.0
21 println("Completed in %f seconds".format(time))
```

Listing C.3: Spark Wordcount algorithm

APPENDIX D

RUNTIME ESTIMATION

```
module runtimeEstimation
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape
open System
open System.IO

type operationValues() =
    let addition = 1
    let subtraction = 1
    let multiplication = 2
    let division = 2
    let stringConversion = 10
    member this.StringConversion = stringConversion
    member this.Addition = addition
    member this.Subtraction = subtraction
    member this.Multiplication = multiplication
    member this.Division = division

type runtimeEstimation() =
    let estimatedRuntime = ref 0
    member this.incrementRuntime value =
     estimatedRuntime.Value <- estimatedRuntime.Value + value
    member this.getRuntime = estimatedRuntime.Value

let operations = new operationValues()

let max x y =
    if x > y
```

```
        then x
        else y

let rec calcRuntime quotation =
    let runtime = new runtimeEstimation()
    let rec traverse quotation =
        match quotation with
        | Application(expr1, expr2) ->
            // Function application.
            runtime.incrementRuntime 1
            traverse expr1
            traverse expr2
        | IfThenElse(_, expr2, expr3) ->
            runtime.incrementRuntime (max (calcRuntime expr2) (
              calcRuntime expr3))
        | SpecificCall <@ (+) @> (_, _, exprs) ->
            runtime.incrementRuntime operations.Addition
            List.map traverse exprs  |> ignore
        | SpecificCall <@ (-) @> (_, _, exprs) ->
            runtime.incrementRuntime operations.Subtraction
            List.map traverse exprs |> ignore
        | SpecificCall <@ ( * ) @> (_, _, exprs) ->
            runtime.incrementRuntime operations.Multiplication
            List.map traverse exprs |> ignore
        | SpecificCall <@ ( / ) @> (_, _, exprs) ->
            runtime.incrementRuntime operations.Division
            List.map traverse exprs |> ignore
        | Lambda(param, expr) ->
            // Lambda expression.
            if expr.ToString().Contains("ToString") then runtime.
              incrementRuntime operations.StringConversion
            Console.WriteLine(expr)
            traverse expr
        | WhileLoop(param, body) ->
            runtime.incrementRuntime ((calcRuntime body) * 10)
        | ShapeVar v ->
            Console.Write("")
        | ShapeLambda (v,expr) ->
            Console.WriteLine(expr)
            traverse expr
        | ShapeCombination (o, exprs) ->
            List.map traverse exprs |> ignore
    traverse quotation
    runtime.getRuntime
```

Listing D.1: Rough runtime estimation