

UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Data processing with Pig Latin

Arthur Lesuisse

Directeur :
Prof. Stijn Vansummeren

Mémoire présenté en vue de
l'obtention du grade de
Master en Sciences Informatiques

Acknowledgements

I wish to thank my thesis director Stijn Vansummeren, who let me work independently yet provided me with precious advice when needed. I would also like to thank Alejandro Vaisman for his comments and corrections.

Contents

1	Introduction	1
1.1	Motivations	1
1.2	Problem statement	2
1.3	Thesis structure	2
I	Background	3
2	Parallel & distributed computing	4
2.1	Parallel architectures	4
2.1.1	Shared-memory	4
2.1.2	Shared-disk	5
2.1.3	Shared-nothing	6
2.2	Parallel programming	7
2.2.1	Explicit versus implicit parallelism	7
2.2.2	Explicit versus implicit clustering	7
2.3	Summary	8
3	Dataflow programming	10
3.1	Introduction	10
3.2	Dataflow graphs	11
3.3	Dataflow systems	11
3.3.1	Historical overview	11
3.3.2	Small- versus large-grain	12
3.4	Dryad	13
3.4.1	Target architecture & setting	13
3.4.2	Programming model	13
3.4.3	Program execution	15
3.5	Summary	15
4	MapReduce	16
4.1	Introduction	16
4.2	Programming model	17
4.2.1	Description	17
4.2.2	Examples	17
4.2.3	Extensions	18
4.3	Detailed operation	19
4.3.1	Typical setting	19

4.3.2	Mapping phase	20
4.3.3	Reducing phase	20
4.3.4	Fault tolerance	20
4.3.5	Example of execution	21
4.4	Summary	21
5	Pig Latin	23
5.1	Introduction	23
5.1.1	Overview	23
5.1.2	Motivations	23
5.2	Language description	25
5.2.1	Data model	25
5.2.2	Programming model	25
5.3	Examples	27
5.4	Compilation into MapReduce	28
5.4.1	Outline	28
5.4.2	Example	28
5.4.3	Limitations	29
5.5	Summary	30
II	Comparison	31
6	Naiad	32
6.1	Programming model	32
6.1.1	Operators	32
6.1.2	Connectors	34
6.1.3	Example programs	34
6.2	Implementation	37
6.2.1	Target platform	37
6.2.2	Specifics	38
6.2.3	Differences and limitations compared to Dryad	38
6.3	Pig Latin to Naiad compiler	39
6.3.1	Compilation algorithm	40
6.3.2	Implemented optimizations	42
6.3.3	Other possible optimizations	42
6.4	Summary	43
7	Join algorithms in MapReduce and Naiad	44
7.1	Repartition join	44
7.1.1	Repartition join in Naiad	46
7.1.2	Repartition join in MapReduce	46
7.2	Broadcast join	47
7.2.1	Broadcast join in Naiad	47
7.2.2	Broadcast join in MapReduce	48
7.3	Semi-join	49
7.3.1	Semi-join in Naiad	49
7.3.2	Semi-join in MapReduce	50
7.4	Hash join	50

7.4.1	Hash join in Naiad	51
7.4.2	Hash join in MapReduce	51
7.5	Summary	52
8	Analysis of MapReduce versus DFG	53
8.1	MapReduce cheating tricks & equivalence to DFG	53
8.2	Theoretical argumentation	55
8.3	Experiments	59
8.3.1	Setting	59
8.3.2	First experiment: compilation & optimization	59
8.3.3	Second experiment: framework flexibility	62
8.3.4	Unperformed experiments	64
8.4	Summary	65
9	Conclusion	66
A	Listings & source code	69
A.1	Hash join operator	69
A.2	Naiad & Pig Latin compiler	70
B	Internship report	71
B.1	Introduction & scope	71
B.1.1	Motivations	71
B.1.2	Choosing a suitable engine	72
B.2	Overview of Pig's storage abstraction layer	72
B.3	Implementation of a MongoDB to Pig adapter	73
B.3.1	Converting data	73
B.3.2	Splitting data	74
B.4	Additional optimization possibilities	75
B.4.1	Pushing down operators	76
B.4.2	Collected grouping	76
B.4.3	Exploiting indexes	76
B.4.4	Associated Pig design issues	77
B.5	Conclusion	77
	Bibliography	78

Chapter 1

Introduction

1.1 Motivations

The Internet is growing fast, and more and more people can access it by the day; companies all around the globe have long begun to exploit it as the biggest marketplace in the world. One of the largest associated business trends is the gathering of all kinds of data about how people use it and what they do with it. Such data is valuable since it can be used, for example, by content providers to adapt and streamline their offerings depending on the customer. As storage is becoming cheaper, companies will often aggressively collect all kinds of web-generated data before even considering what they might want to do with it.

Due to the Internet's massive scale and the systematic nature of this data gathering, companies quickly find themselves with gargantuan quantities of data, which they need to analyze to get something out of it. High-performance parallel computing is needed to do such analysis in reasonable time.

Traditionally, such tasks were accomplished using large and expensive parallel supercomputers. Nowadays, with the size of the data exploding, the cost of acquiring and operating these machines is becoming prohibitive, especially in light of the high level of parallelism desired. The industry, led by Google [9], has thus started to adopt so-called *shared-nothing* architectures, often as clusters of “commodity” computers. Such systems are cheap to build, and offer almost unlimited scalability for *data-parallel* problems with very large quantities of data.

In order to exploit these clusters for this very-large-scale business data analysis, it is desirable to have clustering platforms to help scatter the data-parallel programs across them. Google's MapReduce [9] is one such platform, providing a simple, low-level programming model for data-parallel programs, and a corresponding runtime engine for running them in a highly scalable way. Along with its clones, it currently dominates the very-large-scale computing market. Dryad [17] from Microsoft Research is another such system providing an even lower-level programming model, but is currently only used internally at Microsoft.

Engineers and users quickly identified the need for higher-level languages to help them write programs for these platforms. Pig Latin [20] is one such programming language designed for ad-hoc data processing, currently targeting the open-source MapReduce clone Apache Hadoop [12].

1.2 Problem statement

While MapReduce-like systems have become highly popular because of their open-source implementations, we believe that, being initially designed to be programmed by humans, they are unsuitable for this use as targets for high-level languages. The goal of this thesis is to investigate the limitations of MapReduce with respect to that, see how practical systems work around these limitations, and how they can be lifted by the use of a more general programming model like that of Dryad.

We contribute answers to these questions, and provide an argumentation against the MapReduce model, backed by small-scale experimental data. To that end, we use the Pig Latin language and its implementation as a basis for analysis and experimentation. As a side effect, we also contribute a solution to the following problem from the Pig Latin call for Academic Student Projects [22]:

Investigate a new backend system to Pig (e.g. a Dryad-like system), with a corresponding compiler to run Pig Latin on the new backend.

Finally, in the conclusion of this document, we lay out the basic idea for another possible programming model inspired by both MapReduce and Dryad, motivated by the knowledge acquired during the course of this thesis.

1.3 Thesis structure

This thesis is separated into two parts. The first part provides the necessary background information. Chapter 2 provides relevant background from the parallel and distributed computing landscape. Chapter 3 provides background on *dataflow programming*, the programming model underlying MapReduce, Pig Latin and Dryad. Finally, Chapter 4 details the MapReduce model and its implementation, and Chapter 5 presents the Pig Latin language and outlines the current status and limitations of its compilation to MapReduce.

The second part contains our contributions. Chapter 6 describes Naiad, our implementation of a Dryad-like system for experimental purposes, as well as our Pig Latin compiler targeting it. To fuel our argumentation with practical cases, Chapter 7 outlines different algorithms for the relational join operation, common of Pig Latin programs, and possible implementation over MapReduce and Naiad. Finally, Chapter 8 contains our main contributions, providing our argumentation and experimental results.

Part I

Background

Chapter 2

Parallel & distributed computing

Parallel and distributed computing have become an important trend and research area in computer science. By *parallel* we mean programs whose performance can be increased by running simultaneously on multiple processors; *distributed* adds the idea of loose coupling between these processors.

Multiple reasons can be found for this new found importance. Among the most often cited is the fact that Moore’s law no longer applies, roughly meaning that individual processors produced by the industry no longer become exponentially faster with each generation, implying the need to use multiple processors in parallel to “keep up”.

Another reason that is more relevant to this thesis is that, due to the rise of the Internet, the amount of data that we want to analyze with computers seems to grow even faster than Moore’s law. Is it thus becoming impossible to manage such masses of data using increasingly powerful single computers; we need computer systems that can be scaled with the data, which is made possible through parallel and distributed computing.

This chapter presents some of the parallel and distributed computing landscape relevant to the rest of the thesis.

2.1 Parallel architectures

Parallel computing imply the use of hardware architectures with multiple processors that can operate simultaneously. These architectures are often classified according to which resources are common to, or shared between the processors in the machine (also implying which resources beside processors are multiplied, i.e. *not* shared). This classification is especially relevant when discussing parallel database and data processing systems [25].

2.1.1 Shared-memory

Shared-memory computers, which include most commodity desktop computers nowadays, use multiple tightly-coupled processors that share the same main-memory (as well as all other resources, like secondary storage, network access, etc...). A single instance of an operating system runs on the computer and manages the assignment of processes or tasks to processors. Programming it for parallel execution requires managing the contention of access to these shared resources.

Such architectures are most efficient for relatively small parallel problems, as tight coupling between processors means fast data exchange between them and high utilization of the computer’s resources, given that contention issues are managed properly.

Cheaper commodity machines use a single memory bank shared between their processors; an architecture called *Symmetric Multi-Processing (SMP)*. In order to lessen the interference of memory access contention, larger-scale machines such as shared-memory supercomputers use more complex memory schemes; an architecture called *Non-Uniform Memory Access (NUMA)*. In such a system, each processor has fast access to its own memory bank, and data can be exchanged between banks by special high-speed interconnects.

The downside of these systems for larger problems is that they do not scale well, especially from a cost point of view. Adding processors increases resource contention, and has a limit above which all the hardware must be changed to accommodate them. In a NUMA system, more processors means more advanced and expensive interconnects. As a result, large shared-memory supercomputers have prohibitive costs, and are only used when this tight coupling is absolutely necessary to achieve good performance (e.g. typically for large-scale physical simulations such as weather forecasting).

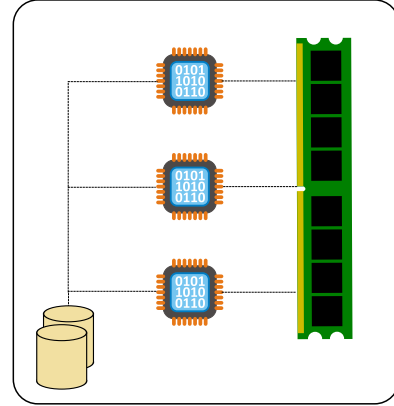


Figure 2.1: Shared-memory architecture

2.1.2 Shared-disk

In a *shared-disk* architecture, multiple computers, each with its own memory and processor, are connected to one another in a *LAN* (local area network). Additionally, each is connected to a shared secondary storage system, i.e. the “disk” (see fig. 2.2). Each computer runs its own instance of an operating system, and accesses the shared disk as if it were connected locally. Such a parallel system made from loosely coupled computers is called a *computer cluster*; individual machines within a cluster are sometimes referred to as *nodes*.

Parallel programming requires managing contention of access to the disk (usually automated in some measure by the disk itself), and network communications between individual computers. Usually, an instance of the program will run on each computer; these instances coordinate themselves through the LAN, and use the shared disk for loading input and storing output data. Often, at least one of these instances is considered the *master*, and acts as a central authority for coordination and the division of labor.

Such architectures are much less costly to scale than shared-memory systems, as one can easily add a new computer to obtain increased performance, provided that the software is written with the ability to take advantage of additional computers. Of course, there is still the issue that adding more computers might increase disk access contention and the need for storage space. However, such shared disks are usually complex networks themselves that can be scaled up separately to increase capacity and availability.

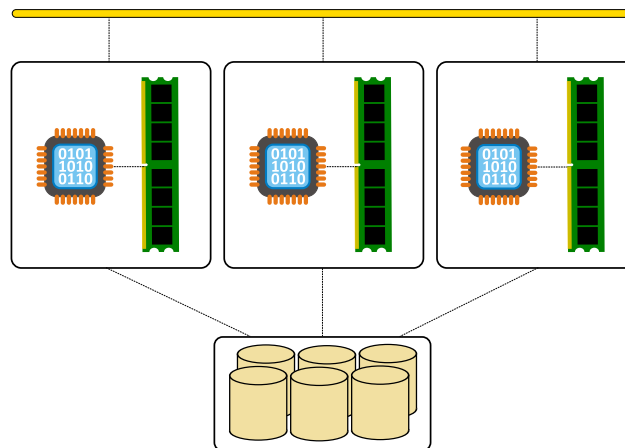


Figure 2.2: Shared-disk architecture

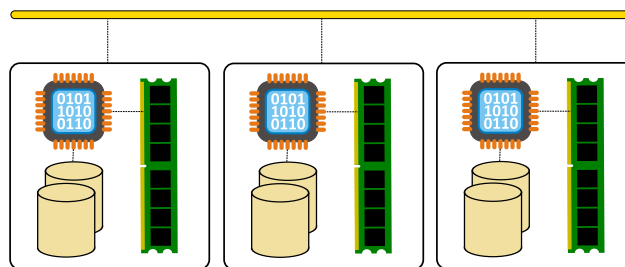


Figure 2.3: Shared-nothing architecture

Though here separated from shared-memory systems, it should be noted that both architectures are not mutually exclusive. In fact, the biggest supercomputers nowadays use a hybrid of the two in order to alleviate the scaling problem of shared-memory systems that we mentioned earlier. Such systems are made from a number of shared-memory subsystems connected together in a shared-disk architecture.

2.1.3 Shared-nothing

Of most recent usage are so-called *shared-nothing* architectures, comprised of completely independent computers, each with its own processor, memory and disk, connected together in a local area network, thus forming a cluster (fig. 2.3). Programming them is essentially similar as with shared-disk systems, except with respect to storage. Usually, the computers and LAN themselves act as a shared disk, with each participating machine contributing its own storage space to the whole. To that end, a special program called a *distributed filesystem (DFS)* runs on every machine.

This adds an important new aspect to consider when programming the system: that of *partitioning*. Unlike a shared-disk system in which each computer has equal access to all the data, now each computer has privileged access to data in its own disk, and much slower access to the rest, being limited by the relatively slow LAN. It is thus very important to control where different parts of the data reside, in order to assign labor in a way that minimizes the amount to be transferred over the LAN.

Discussion The reason why the industry is currently switching to shared-nothing architectures is because they scale the best. Assuming software that can take good advantage of these architectures, obtaining a more powerful system is as simple as plugging in additional hardware; the cost of the system scales almost linearly with its processing capacity. However, they can be considered the most difficult to program, as one needs to handle LAN coordination, data partitioning, the DFS, and failure handling issues which become increasingly common when adding computers. For this reason, it is highly desirable to create programming models that allow writing efficient programs while making abstraction of these issues. This idea is central to the rest of this thesis.

2.2 Parallel programming

We have outlined different kinds of hardware parallel architectures, and stated how they were difficult to program. In this section, we delve a little deeper into this issue. We first discuss the simpler parallelism issues associated with shared-memory systems and writing parallel programs in general. We then move on to more complex issues associated with writing parallel programs for clustered systems, i.e. shared-disk and shared-nothing architectures.

2.2.1 Explicit versus implicit parallelism

The most traditional way of writing parallel programs for shared-memory machines is to add explicit parallelizing instructions to the code. Examples of such instructions include instructions for creating new tasks/processes/threads to be run in parallel, and instructions to perform synchronization and communication between them. This practice is known as *explicit parallelism*.

Writing and debugging software using explicit parallelism is known as an extremely difficult and error-prone task, which is why many attempts have been made to take that difficulty away from the programmer.

Some compilers (e.g. for C/C++: [8, 15]) try to identify what parts of the code can be *parallelized automatically*; however, this is very hard to do accurately for imperative programming languages. As a result, there is interest in expressing programs in ways that make the potentially parallel parts of the system more obvious to detect by compilers; this is known as *implicit parallelism*. For example, in a purely functional programming language, all data dependence can be known at compile time; thus, independent computations can be run in parallel.

The problem does not end there however, as the compiler or runtime environment would then have to figure out what computations are *worth* parallelizing. This is non-trivial, especially in a clustered environment where the cost of scheduling work to a distant machine might be far from negligible.

2.2.2 Explicit versus implicit clustering

Purely *explicit clustering* is nothing more than the practice of writing networked applications. This requires programmers to manage complex issues like networking, failure handling, load balancing, and more. This approach is thus seldom used in high-performance computing contexts, where these issues are especially important. Instead, most computer clusters are managed using specialized software that can make some parts of this process implicit.

Clustering & task parallelism

The simplest clustering frameworks consist of a set of tools and mechanisms for scheduling independent jobs onto different machines of the cluster. Tasks are submitted globally to the system, which then assigns them to physical machines, using some load balancing scheme to distribute the work among them. These systems exploit *task-level parallelism*: separate, independent tasks can be run in parallel.

Additionally, these systems often provide so-called *message passing* libraries, which let programmers write parallel programs that can spread across the cluster. The programmer then has to write explicitly parallel code, using the system's message passing facilities to provide synchronization and communication among the concurrently running parts of his program.

Operating system-level tools have been devised to abstract these facilities away from the user: the so-called *Single System Image* systems. These leverage the tools already present in the underlying operating system, making them usable at the cluster's level. For example, OpenSSI (based on Linux) provides users with a seemingly "normal" UNIX system, but actually shares single IO, process and IPC (*inter-process communication*) spaces across the entire cluster, giving the illusion of a single machine [1].

This results in users being able to profit from the load-balancing facilities when simply running a program as they would on a single machine, and to write parallel programs using the standard UNIX IPC mechanisms, that will end up running on multiple machines across the cluster. This constitutes purely implicit *clustering*, as the users don't have to know that they're working with a cluster of machines. On the other hand, it is still explicit *parallelism*, because they still have to write multi-process programs if they want to take advantage of parallel execution.

Clustering & data parallelism

Even with such facilities to ease clustering, writing explicitly task-parallel programs requires the programmer to devise parallel algorithms, splitting the work into sets of tasks that will best utilize the machines in the cluster. Techniques for implicit task-level parallelism (such as compiler auto-parallelization of imperative languages) are, as of now, not useful for writing highly parallel programs for large clusters.

There is however a large class of problems called *data-parallel* problems, which are much more suited to implicit parallelization and clustering. A data-parallel problem is one in which the computation to perform can be divided into tasks that each apply to a different part of the input data. For example, the simplest form of data-parallel problem involves a list of data elements with which the same operation must be performed.

Data-parallel problems are a natural match for clusters. The amount of parallelism in the problem scales with the size of the data; thus, data-parallel problems of any size can be solved efficiently given a large enough cluster. Moreover, by restricting oneself to data-parallel problems, one can create highly efficient programming models enabling both implicit parallelism and implicit clustering.

2.3 Summary

Writing parallel code involves a trade-off: explicitly parallel programming leaves the programmer in control of what parts will be executed where, leading to the best po-

tential performance at the cost of difficulty, meaning that the code will take longer to write and debug. Also, such work can only be accomplished by skilled programmers with experience in writing parallel code and dealing with the associated quirks.

The same trade-off then appears at another level, when one wants to run their parallel programs on computer clusters. Thus, it is highly desirable to make implicit parallelism and clustering efficient, if only for some classes of problems, so that these problems can be solved easily by “normal” programmers, and the resulting programs can be run efficiently on large computer clusters.

The MapReduce and Dryad systems, which we will describe later on, both address these issues for the class of data-parallel problems.

Chapter 3

Dataflow programming

The MapReduce, Pig Latin, Dryad and Naiad systems discussed in this thesis all take root in the concept of dataflow programming, a family of programming models in which programs are modeled as the flow of data between their operators. In this chapter, we present the concept of dataflow programming and its relevance with respect to parallel and distributed computing. Finally, we provide a brief exposition of Dryad, whose concepts we reuse in our own system Naiad.

3.1 Introduction

We explained in Chapter 2 the difference between explicit and implicit parallelism, and stated that writing explicitly parallel programs was considered very difficult. One reason for that is related to the concept of a program’s *execution state* (henceforth shortened to just *state*). This notion encompasses the program’s memory, i.e. its variables, and the position of the instruction pointer (or pointers in the case of a parallel program). Imperative parallel programming is made difficult because the precise effect of an instruction depends on this state, which itself depends on the exact sequence of preceding instructions, which is usually nondeterministic in a parallel setting. The program’s state is often said to be *hidden* from the programmer in an imperative setting: when writing a particular instruction, there is no simple way of knowing what state the program will be in when that instruction is executed.

Dataflow programming, in its several forms, is a programming model close to pure functional programming: there are no variables, and instructions have no *side-effects*: they compute values from their operands without otherwise changing the program’s state. The programmer no longer specifies the exact sequence of instruction in their intended execution order, but rather the chain of operations to be applied to the data. The actual order of execution is then *implicit* and can be derived by the compiler (or dataflow-specialized hardware): an instruction can be executed as soon as its operands have been computed.

As an example, consider the following program:

```
x = [some value]
a = f(x)
b = g(x)
c = h(a, b)
```

In an imperative setting, the `f` function would be called before the `g` function. It

is then the programmer’s responsibility to ensure that this is the correct order: for example, it is not excluded that the `f` function changes the value inside its argument variable `x`, in which case reversing the order would produce a different result.

In a dataflow setting however, `x` is not a variable but a *value*, and cannot be changed. Because the compiler knows that, it also knows that the `f` and `g` functions do not depend on each other, and thus that the second and third instructions can be executed in any order. Moreover, such instructions that do not depend on each other can be executed in parallel; dataflow programming languages thus offer implicit parallelism.

3.2 Dataflow graphs

Although dataflow programs can be represented as “traditional” textual instructions sequences as shown above, they exhibit a structure that lends itself very well to graphical representation under the form of a *dataflow graph* (*DFG*). In such a graph, vertices represent distinct instructions, while edges represent the flow of operands from one instruction to another. Cycles are not permitted, as an instruction cannot depend upon its own result; a dataflow graph is thus a directed acyclic graph. Fig. 3.1 shows a DFG representation of the textual example above.

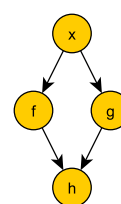


Figure 3.1: DFG representation of a dataflow program

This can be considered a more natural representation of dataflow programs. Contrary to the textual example which used single-assignment variables in an attempt to emulate the imperative style, a dataflow graph clearly shows the *absence* of variables: there are no more `a`, `b` and `c` labels. Additionally, this representation makes visually clear the implicit parallelism present in the program: any two vertices not linked by any path are eligible for parallel execution.

Note that dataflow graphs are not a programming model per se, but rather a family of programming models. While this particular example is rather straightforward, the interpretation of a dataflow graph in general depends on the particular semantics of the programming model used. For example, consider a vertex with two outgoing edges: do these edges represent copies of the output value from the vertex, or does the vertex produce distinct values on each edge? Only with an actual system in mind is it possible to answer such a question.

3.3 Dataflow systems

3.3.1 Historical overview

Many dataflow languages and systems have been devised since the introduction of the dataflow model. The initial goal of most of these systems was to write programs for parallel supercomputers. Among the earliest well-known general-purpose textual dataflow language are LAU, Lucid, SISAL, Lustre, ... Hardware dataflow architectures have also been devised, most notably the MIT’s *tagged token* architecture; the machine language for such architectures is dataflow graphs, to which higher-level dataflow or functional languages can be compiled. More recent initiatives include Prograph, an object-oriented dataflow language in which dataflow graphs were visually composed on

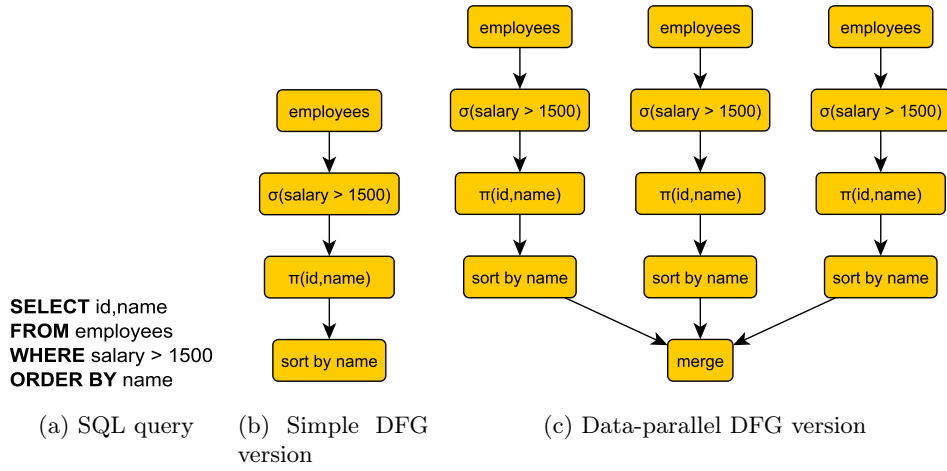


Figure 3.2: Simple SQL query with possible DFG translations

the screen. Its implementation was however sequential, making it more an exercise in visual programming than in implicit parallelism.

Another area of particular success for dataflow languages, particularly visual ones, is that of domain-specific languages. Indeed, domains such as audio/video, instrumentation, or signal processing in general lend themselves easily to this paradigm of “boxes that plug into one another”. Current such systems include LabView, Max, Pure Data, and many more. Again, the goal of such system is much less parallel execution than providing an adequate domain-specific programming model.

Johnston et al. [18] provide a more complete overview of the historical landscape of dataflow programming.

3.3.2 Small- versus large-grain

All of the systems that we mentioned so far are limited to small- to medium-grain parallelism, i.e. parallelization of single instructions or small routines that treat relatively small amounts of data. These can work well on shared-memory parallel computers where work can be quickly dispatched to different processing units, and data can be exchanged between processing units with minimal latency.

Things are however different with the kind of system relevant to this thesis, i.e. systems for processing web-scale datasets using large clusters of machines, interconnected through relatively slow networks, forming a shared-nothing architecture. These systems have very different requirements than those for which traditional dataflow languages were designed. For example, small-grain parallelism is extremely inefficient for such systems, as the scheduling and data transfer latencies far exceed the execution time of individual instructions.

Larger grain, shared-nothing dataflow architectures have been used for some time “under the hood” for implementing parallel relational database systems [11], as pioneered by Teradata. Indeed, relational queries expressed in a declarative language such as SQL can easily be translated into dataflow graphs, in which database tuples flow between relational operators. Additionally, in such settings where database tables are scattered or partitioned across the cluster, parts of the DFG for a query can be repli-

cated, taking advantage of data-parallelism. Fig. 3.2 shows an example SQL query and how it can be translated into a DFG, then further data-parallelized.

Note that the semantics of dataflow graphs in such systems is a little different than what we have shown before: vertices no longer represent operations on single datums but on streams of data. As a consequence, another form of implicit parallelism appears in such a DFG: a linear chain of operations can be run as a pipeline; a form of task parallelism. In the last example, the π (projection) operator could process a batch of tuples while the σ (selection) operator processes the next batch.

More general-purpose dataflow systems for shared-nothing architectures are uncommon, but some do exist, such as Dryad from Microsoft Research, which we shall present in more detail.

3.4 Dryad

3.4.1 Target architecture & setting

To the best of our knowledge, Dryad is the only existing production-level general-purpose dataflow system that targets large shared-nothing clusters. Rather than specialized parallel computers or clustered database machines, it is intended to run on farms of up to thousands of commodity, unreliable, general-purpose computers, interconnected by standard network technologies (typically Ethernet). Parallelization thus has to be of a very-large-grain nature, and additional measures must be taken to cope with frequent failures and the disparity of hardware.

Input data is taken from (and output data written to) a *distributed filesystem* (DFS) across the cluster. In a DFS, files are split into smaller chunks, and chunks are scattered to individual machines of the cluster. Chunks are also automatically replicated, with copies placed on different machines, providing higher availability and some measure of fault tolerance. Unlike parallel database systems we mentioned earlier in which the system is responsible for organizing relational data into partitioned, ordered and indexed tables for efficient querying, Dryad files are simply files, and any additional organization is left to the user.

3.4.2 Programming model

Dryad programs are specified as dataflow graphs. However, vertices in these graphs are no longer individual instructions or routines, but complete programs that can be written using either a pure sequential style, or an asynchronous event-driven style. Edges represent communication *channels* between these programs, which can either be files, or FIFO or TCP pipes, as chosen by the user.

Dataflow graphs are composed using a special C++ API consisting of four graph-composing operators, which we call *connectors* in this thesis. These connectors are used to build larger graphs out of smaller subgraphs or individual vertices; together they can be used to generate any dataflow graph, with an emphasis on easily generating data-parallel ones. Table 3.1 gives a quick survey of these operators; a more complete and formal description of their operation is given by Isard et al. [17] in their original article.

Symbol	Name	Description	Example
\wedge	replication	The graph on the left hand side of this operator will be replicated a number of times as specified by the integer on the right hand side. Throughout this thesis, we also use the superscript exponentiation notation as a shortcut for this connector (e.g. X^n for $X \wedge n$).	
\geq	pointwise composition	Each output of the left hand side is connected to a corresponding input on the right hand side. If the numbers of outputs and inputs do not match, additional links are created in a round-robin fashion.	
\gg	bipartite composition	Each output of the left hand side is connected to every input on the right hand side. Used e.g. for data partitioning.	
\parallel	merge	The resulting graph contains all the distinct edges and vertices present in either operand, i.e. it produces the union of both graphs.	

Table 3.1: Dryad connectors

3.4.3 Program execution

Top-level vertices in a Dryad DFG, i.e. those without predecessors, represent chunks of an input file in the distributed filesystem. Conversely, bottom-level vertices represent parts of the output data, which will be written to filesystem chunks whose concatenation will constitute the output file. Upon execution of a Dryad program, it is the framework's responsibility to distribute the DFG vertices to machines of the cluster, in a way that will maximize the cluster's utilization and minimize the amount of data that will be transferred over the slow network (i.e. attempting to move the computation close to the data rather than the reverse).

During execution, the Dryad framework monitors the state of the computation, attempting to balance the load between cluster nodes and detecting failures and straggling computers. In the event of such a detection, the framework can reassign and restart the implicated vertices onto another machine; to that end, Dryad expects vertex programs to be fully deterministic and referentially transparent. Edges implemented as temporary files offer a form of *checkpointing*: if a subsequent vertex fails, its input data can be so recovered without re-executing the whole computation. The framework can even decide to schedule the same vertex more than once at the same time, in a process called *speculative execution* intended to prevent such failures. These features allow obtaining very good scalability on large clusters where failures are common.

Dryad's design allows for much usage flexibility: Isard et al. [17] claim their implementation to provide performance and latency rivaling specialized parallel database systems from single parallel computers up to small clusters, and to scale well to large clusters of thousands of computers.

3.5 Summary

Dataflow programming is a very abstract and general model which is most often used as a base to devise concrete, more specific systems. As we will see, MapReduce and Pig Latin both specialize dataflow programming, each in its different way: the former by constraining the form of dataflow graphs, and the latter by constraining the data model and operators that can be used.

Systems like Dryad however attempt to constrain nothing, and to provide the most general-purpose dataflow-based system possible. We believe that a well-devised such system could act as a sort of "assembly language" for data-parallel systems and programs, and could be used to efficiently implement all kinds of higher-level systems.

Chapter 4

MapReduce

This chapter presents MapReduce, a programming model for writing data-parallel programs, and implementation strategy for running them on very large clusters in a highly scalable way. MapReduce and its implementations are seeing ever-increasing popularity since their introduction, and are now the de-facto standard for data processing at the largest scales.

Foreword As a programming model, MapReduce can be implemented in many different ways depending on the target environment. The technical details we describe in this chapter are common to the best-known implementations targeting large clusters, like Apache Hadoop [12] and Google’s original implementation. The contents of this chapter is based on the exposition by Dean and Ghemawat [9].

4.1 Introduction

MapReduce is a computing paradigm for writing data-parallel programs with fully implicit parallelism and clustering. It is inspired by the common `map` and `reduce` functional programming primitives. These operate on collections of data: `map` takes a list, applies a *mapping function* to each element of it, and returns the resulting list. `reduce` takes a list and returns a scalar, by successively applying a *reducing function* to the first two elements of the list, and replacing them with the result of this function.

While a MapReduce program consists in essence of a mapping function and a reducing function, there are some differences with respect to the functional programming equivalents. For example, the mapping function does not have to produce one result for every input, but can produce several, or none at all. Another difference is that MapReduce manipulates data as key-value pairs. The mapping function transforms each key-value pair it receives into zero or more new pairs. The reducing function aggregates all pairs with the same key, and yields zero or more final pairs.

The important fact is that this mapping function can be replicated to different machines, to run simultaneously on different parts of the data, thus exploiting data parallelism. The reducing function can also be so replicated by partitioning the key space and running one reducer for each partition.

The original MapReduce implementation by Dean and Ghemawat, as well as its open-source clone Hadoop, both target large shared-nothing clusters of computers. They are now of widespread usage for solving problems in the petabyte scale on clusters

of thousands of computers. For example, MapReduce is used most notably for indexing web pages for Google’s web search engine, the original purpose of the framework.

4.2 Programming model

4.2.1 Description

Data in MapReduce is modeled as key-value pairs (henceforth referred to as just *pairs*). Let K be the set of all keys and V the set of all values. We use the notation $list(S)$ to denote an ordered list with elements in the set S , allowing duplicates. The programmer submits to the system:

- a *mapping function* ($map : K \times V \rightarrow list(K \times V)$) that takes one pair and outputs any number of new pairs
- a *reducing function* ($reduce : K \times list(V) \rightarrow list(K \times V)$) that takes a key and a set of pairs corresponding to that key, and outputs any number of new pairs.

The execution semantics are as follows. Let us assume that the input data is the list of pairs $\{(k_1, v_1), \dots, (k_n, v_n)\}$.

1. The mapping function will be called for every pair (k_i, v_i) from the input data. This is called the *map phase*.
2. The resulting intermediary pairs $I = \bigcup_{i=1}^n map(k_i, v_i)$ will be collected together and sorted by key, where \bigcup denotes list concatenation instead of set union. We call this the *shuffle phase*.
3. The reducing function will be called for each intermediary key k in I with the list of corresponding intermediary values $V = \{v \mid (k, v) \in I\}$. This is known as the *reduce phase*.
4. The pairs output by the reduce function constitute the result of the computation.

The first step is trivially parallelizable, since all calls to the mapping function are independent of one another. The second step can take advantage of parallel sorting algorithms, like merge-sort. Finally, the third step can be parallelized if the intermediary I pairs are partitioned by their key; different partitions can then be treated by different computing nodes running the reducing function.

Note that MapReduce is a specialization of DFG programming models: any MapReduce computation can be seen as a dataflow graph like that of fig. 4.1. MapReduce constrains the general form of the graph, allowing only to change the mapping and reducing functions.

4.2.2 Examples

Here we present examples of MapReduce solutions to two simple data-parallel problems. In the following examples, the function `emit(key,value)` is taken to add its arguments to the output as a key-value pair.

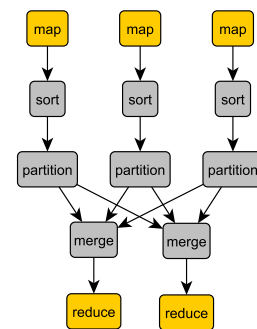


Figure 4.1: Dataflow of a MapReduce program

Counting specific words Let us say that we have a large collection of documents in which we want to count the occurrences of certain words. This can be expressed as follows (in pseudocode):

```
function map(key, value) = {
    # key = anything, value = chunk of text
    foreach word in splitwords(value) {
        if word_is_of_interest(word)
            emit(word, 1)
    }
}

function reduce(key, values) = {
    emit(key, count(values))
}
```

This assumes an input function that respects word boundaries. The `word_is_of_interest` function specified which words need to be counted. When a matching word is found, it is send as the key of a key-value pair, with the number 1 as a place-holder value. Reducers will simply count the number of values associated with each specific word.

Reverse web-link graph [9] The following MapReduce pseudocode computes the reversed graph of web page links for a set of pages:

```
function map(key, value) = {
    # key = webpage URL, value = webpage content
    foreach linkurl in find_links(value) {
        emit(linkurl, key)
    }
}

function reduce(key, values) = {
    emit(key, values)
}
```

The `find_links` function is assumed to recover the URL address of each link in the HTML code of the provided web page. For each of these parsed addresses, a key-value pair is outputted with the parsed link as a key and the originating address as a value. The reducers will output each link along with the address of every page that contains the link.

4.2.3 Extensions

The implementations studied in this chapter also allow the user to specify:

- the *input function* used to parse input pairs from filesystem bytes;
- the *output function* used to serializes output pairs back to filesystem bytes;
- a *combining function* whose use will be outlined in the next section;
- the *partitioning function* that is used to distribute output pairs from mappers to different reducers;

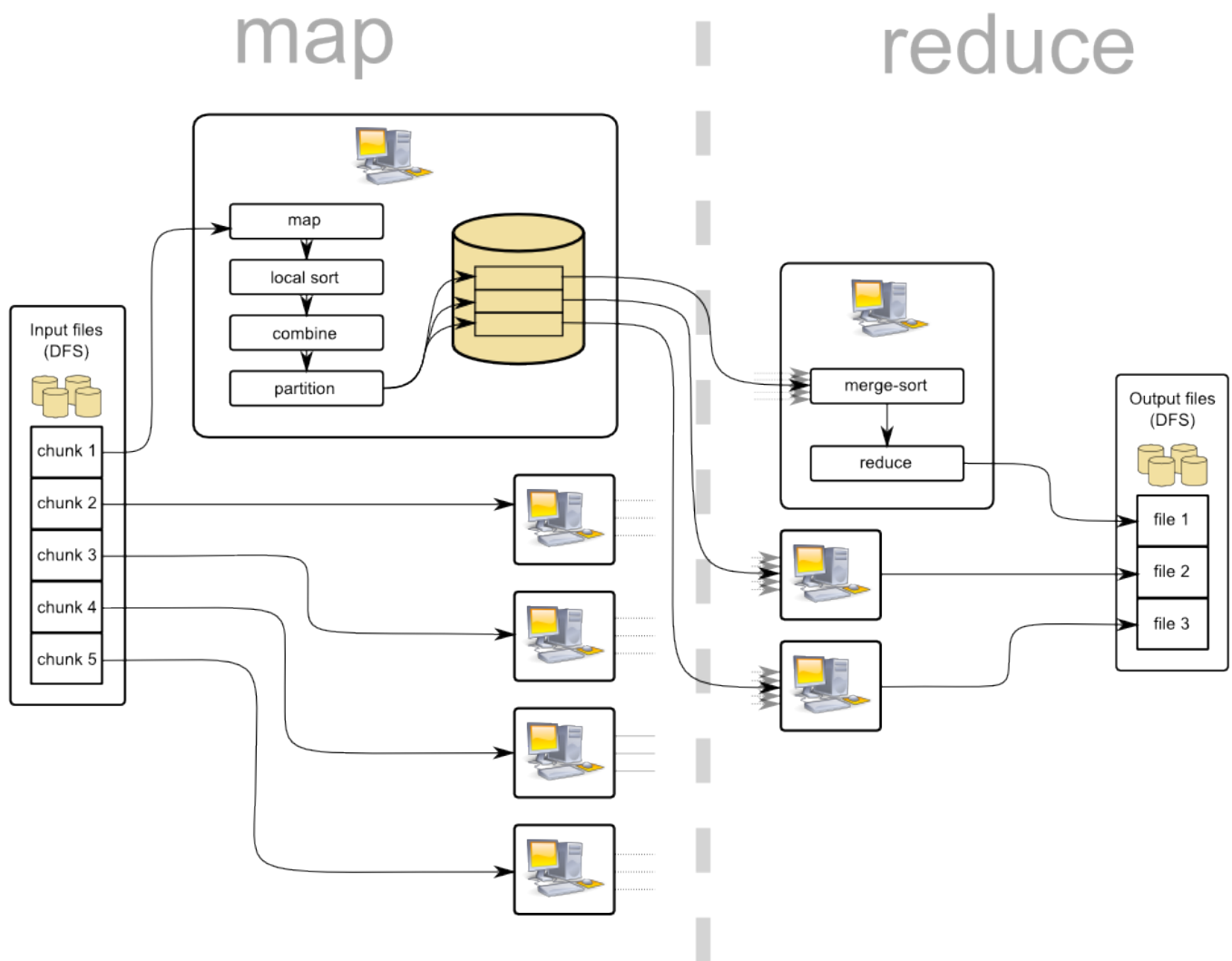


Figure 4.2: MapReduce job execution

- the *comparison function* that is used to sort keys.

A variety of other facilities are also provided, such as the ability to run map-only jobs without the shuffle and reduce phases, the ability to submit trees of inter-dependent jobs, and more.

Other works also extend MapReduce by providing additional computing phases; e.g. map-reduce-merge [27] provides a *merge* phase to combine the outputs of several MapReduce jobs.

4.3 Detailed operation

4.3.1 Typical setting

As stated before, the typical platform for MapReduce is a large cluster of hundreds, or even thousands of “commodity” computers, communicating via switched Ethernet. Jobs

are submitted to a *master* node, which takes care of assigning parts of the computation to the other (*worker*) nodes.

Data storage across the cluster is typically done using an ad-hoc *distributed file system* (*DFS*), such as GFS [14] for Google’s implementation and HDFS [4] for Apache’s. These filesystems store data partitioned in large *chunks*, with each chunk replicated across multiple machines, enabling concurrent access and failure tolerance. Furthermore, using distributed storage on the same cluster that runs MapReduce jobs enables the execution framework to take advantage of *data locality*, i.e. to assign work to nodes that are nearby the input data, saving network bandwidth.

4.3.2 Mapping phase

When a job is submitted, the master node creates one map task per chunk (usually tens of megabytes) of input data; it then assigns these tasks to worker nodes in the cluster. Each of these tasks proceeds as follows. First, the data chunk is fetched and parsed as a set of key-value pairs. The user’s mapping function is then called on each of these pairs, yielding new key-value pairs that are collected together and sorted by key.

These new pairs can then go through the optional combining phase, where another user-defined function is applied to the set. This can save time and space for some problems. For example, consider a program where the output values of the mapping function are to be summed in the reducing function. A combining function that computes the partial sum would then improve efficiency, by not requiring to store and process the useless intermediary values.

Finally, the data goes through the partitioning phase, where it is separated in one partition per (future) reducing task, for example using a hash function. In each partition, pairs are sorted according to their key. Partitions are saved as local files before the reducing phase begins.

4.3.3 Reducing phase

When a map task is finished, the master node is notified about the location of the files holding the partitioned intermediary data. Each of these locations will be passed on to the corresponding reduce task, which will collect the data and merge-sort it with the output of the other map tasks.

Once all the map tasks are done and all the data has been collected, the reducers call the user’s reducing function, once for every distinct key in sorted order, passing it the set of key-value pairs corresponding to that key. Each reducer serializes its results to a file.

4.3.4 Fault tolerance

Because the target systems consists of many general-purpose commodity computers, hardware failures are common and have to be expected. MapReduce uses re-execution as a failure handling mechanism.

While a job is executing, the master node periodically pings busy workers to ensure that they’re still alive. If a worker node doesn’t answer in time, the master considers it dead. Then, all tasks that were scheduled on the dead worker are restarted onto another worker machine.

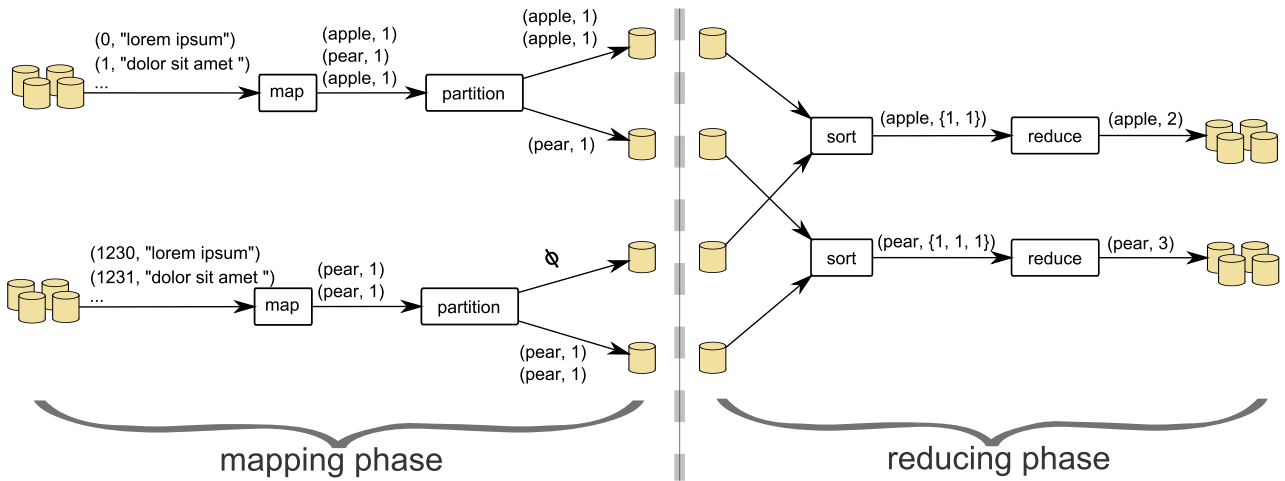


Figure 4.3: Example of MapReduce execution flow

This mechanism requires the `map` and `reduce` functions to be *referentially transparent*, i.e. they must always produce the same output for a fixed input. If this condition is met, then the re-execution of tasks because of failures will not alter the results of the computation.

4.3.5 Example of execution

Consider the word-counting example described in the previous section, and that we want to count the respective occurrences of the words “apple” and “pear”. Using a text-specific input function, mapper nodes will generate key-value pairs where the values contain pieces (e.g. lines) of the input text from the distributed filesystem. Figure 4.3 gives an example of the flow of pairs in the system during the mapping and reducing phases, assuming two mappers, two reducers, and a partitioning function that differentiates the words “apple” and “pear”.

In this example, input text is first parsed in key-value pairs in which keys represent line numbers, and values the corresponding lines of text. The map function extracts the words “apple” and “pear” from every line it finds them in; then, the framework partitions them so that one reducer receives the instances of “apple” and the other one those of “pear”. Finally, reducers sum the counts of their respective words, yielding the final counts as results.

4.4 Summary

MapReduce has the advantage of providing a very simple programming model: two simple, sequential functions become a huge parallel and distributed program, capable of scaling almost infinitely to deal with about any amount of data.

Although this simple, constrained programming model greatly simplifies that kind of highly scalable implementation, it is also a drawback for the programmer. Though many data-parallel problems easily fit in this map-reduce structure, not all of them do so perfectly. Several extensions to the basic model have been proposed, adding additional and optional phases to the computation so as to ease the efficient implementation of

some programs.

Additionally, writing programs as map and reduce functions proves to be cumbersome in practice, especially in domain-specific settings which would be better served by higher-level tools. For this reason, languages such as Pig Latin have been developed as a way to further ease programming for MapReduce systems.

Chapter 5

Pig Latin

In this chapter, we present Pig Latin, a language for data processing, originally developed by Yahoo! to ease expressing relational-like programs for running on Hadoop. We start by outlining the language’s goals, style and the operators it provides. As examples, we provide Pig Latin version of the same programs we’ve shown in Chapter 4. Finally, we describe the way in which Pig Latin programs are compiled into MapReduce ones for execution on Hadoop, outlining the limitations that we perceive to this compilation strategy.

5.1 Introduction

5.1.1 Overview

Pig Latin [20] is a dataflow programming language; every statement expresses a transformation of some data set into another, using relational-style operators. These transformations chain together to compute the final result set. Though Pig Latin uses the same kind of operators as database query languages such as SQL, programs are written in a more procedural-like fashion: the user has to write individual successive transformations separately rather than pack them all up in a large query (see table 5.1 for a comparative example).

Pig Latin programs are not meant to be executed procedurally however. As with other textual dataflow languages, the order of statements is not necessarily the order of their execution, and the “variables” defined in the code are not actual variables, but merely *labels* that represent data flow between successive relational operators.

The only current implementation of Pig Latin, Apache Pig, targets the Hadoop MapReduce platform.

5.1.2 Motivations

While MapReduce was created to ease writing data-parallel programs for large clusters, programmers soon found its programming model to be too cumbersome for frequent usage. They found that although many tasks they used it for were similar in concept, code reuse was difficult, and writing a new program often amounted to a lot of boilerplate. There was need for a higher-level language that could easily express common data analysis tasks to be executed on MapReduce. As an answer to that need, researchers at Yahoo! developed Pig Latin.

SQL version
<pre>-- lists users that have clicked at least 50 times on a link to some -- specific URL. SELECT user FROM user_clicks WHERE targeturl = 'http://example.com/url' GROUP BY user HAVING SUM(clicks) >= 50</pre>
Pig Latin version
<pre>clicks = LOAD 'user_clicks' AS (user, sourceurl, targeturl, clicks); forurl = FILTER clicks BY targeturl=='http://example.com/url'; grouped = GROUP forurl BY user; morethan50 = FILTER grouped BY SUM(forurl.clicks) >= 50; result = FOREACH morethan50 GENERATE \$0; DUMP result;</pre>

Table 5.1: Pig Latin style vs. SQL style

Why not SQL?

Pig Latin was developed with the programmer in mind. SQL allows the production of “sentences” of a specific form, with the goal to resemble English and be usable by non-programmers. This strongly constrains how queries can be written, and feels unnatural to many programmers. Complex SQL queries are known to be hard to write and read.

Pig Latin expresses the same kinds of operations but with a much more comprehensible step-by-step presentation that resembles imperative programming more closely and is thus more familiar to the programmer. All in all, creating Pig Latin instead of using SQL was in good part a practical and aesthetic choice by the developers of Pig who also were the intended end-users.

Why not parallel databases?

Although efficient, shared-nothing parallel database management systems (*DBMSs*) implementing query languages like SQL do exist, their design goals are very different from Pig’s. To grossly summarize, Pig (like MapReduce) is designed for ad-hoc analysis of arbitrary, usually flat, possibly unformatted data. *DBMSs* are on the other hand designed for efficient storage, lookup and querying of structured data. A deeper exploration of the differences between both kinds of systems is made in Chapter 8.

Contenders

Other platforms with similar design goals do however exist. Google’s Sawzall [21] provides a domain-specific language for writing MapReduce program. It provides a lightweight syntax for writing a mapping function and specifying which aggregation

operation (from a fixed set) must be used in the reducing function. It is however arguably more limited than Pig Latin, as Sawzall programs can only express single MapReduce jobs instead of arbitrarily complex transformations. Semantically, though both are mostly equivalent, Sawzall more closely maps to the MapReduce model, while Pig Latin attempts to be higher-level by providing relational-like operators.

SCOPE [5] provides a programming model close to Pig Latin's with a more SQL-like syntax, implemented on top of Microsoft's Dryad. Another project, DryadLINQ [28], provides means of executing queries expressed with LINQ (Microsoft .NET's *Language-Integrated Query*) on the Dryad platform. LINQ's approach differs from specialized languages like Pig Latin in that it allows to use the same programming languages (e.g. C#) and data models as when writing "regular" programs for the .NET platform. The DryadLINQ and SCOPE projects both seem very promising; unfortunately, not many details have been disclosed about them.

5.2 Language description

5.2.1 Data model

Unlike relational database systems, Pig Latin offers a *nested* data model that can represent complex structures. In addition to *atomic* types such as numbers and strings, Pig Latin provides the following aggregate types:

- *tuples*, which are ordered collections of values (called *fields*) of any types;
- *bags*, which are unordered collection of tuples, possibly of different lengths and containing fields of different types; and
- *maps*, which are collections of key-value pairs, where the key is of an atomic type and the value is of any type. Keys are unique inside the map, and values can be looked up using their key.

Tuples are thus analogous to their relational counterparts, with the addition that they can contain nested structures. Bags are analogous to (unindexed and unordered) relations, with the exception that the tuples contained can each have a different *schema*, i.e. different fields with different types.

Fields inside tuples can be named; they can be dereferenced either by name or by position. The same goes for tuples inside bags. Values inside maps can be dereferenced using their corresponding key.

5.2.2 Programming model

One of the most important design goals of Pig Latin is parallel execution. To that end, it provides a limited set of relational operators, **most** of which lend themselves to efficient parallelization. Most Pig Latin statements consist in an assignment of the result of such an operator to a variable. Table 5.2 gives a list of some of these operators along with their descriptions.

Other operators and functions, can be used where appropriate within relational statements. These include arithmetic and comparison operators, as well as functions that operate on collections, such as counts, averages, etc... Pig Latin also allows *user-defined function*, written in other programming languages, to be used in this way.

Operator	Description
FOREACH ... GENERATE ...	Applies a transformation to each tuple in an input bag, yielding an output bag. Note: to exploit Pig Latin's nested data model, some operators can be nested in the transformation specified by GENERATE ... in order to operate on nested data.
DISTINCT ...	Removes duplicate tuples from a bag, yielding a new bag.
[CO]GROUP ... BY ...	Groups tuples in a bag by one (or more) of their fields. The result is a bag in which the tuples contain the grouping keys as first field, and the bag of corresponding tuples as a second field. If COGROUP is used, several bags can be grouped together in an operation similar to relational joining.
FILTER ... BY ...	Removes tuples in a bag that don't satisfy a given condition.
JOIN ... BY ...	Joins two or more bags in the relational sense. An additional parameter (USING) can be set to choose the join algorithm used.
UNION ...	Computes the concatenation of two or more bags (not the set union !).
LOAD ...	Loads a bag from the filesystem, using a specified (built-in or user-defined) loading function. The schema of the loaded data can be optionally specified.
STORE ... INTO ...	Writes a bag to the filesystem, using a specified (built-in or user-defined) storing function.

Table 5.2: Some of Pig Latin's relational operators

This allows custom ad-hoc data processing to be inserted within Pig Latin scripts, an important design goal.

Pig Latin is *statically* and *weakly* typed: every variable has a definite type, but values are implicitly cast when needed (and possible). When reading tuples using the `LOAD` operator (c.f. table 5.2), fields without an explicitly specified type are assumed to be of the atomic type `bytearray`.

5.3 Examples

As examples, here follow translations in Pig Latin of the examples of MapReduce programs given in section 4.2.2.

Counting specific words To simplify, we assume here that we want to count the occurrences of the words “apple” and “pear”.

```

1 text = LOAD 'mydocument.txt' USING TextLoader()
2       AS (line:chararray);
3 words = FOREACH text GENERATE FLATTEN(TOKENIZE(line))
4       AS (word);
5 filtered = FILTER words BY (word=='apple')
6           OR (word=='pear');
7 grouped = GROUP filtered BY word;
8 counts = FOREACH grouped GENERATE $0, COUNT($1);
9 STORE counts INTO 'resultfile.txt';
```

The first statement loads the input document using the built-in `TextLoader()`, which outputs a bag in which each tuple contains one line of text. The end of the instruction (line 2) declare those tuples to each contain one field named `line` of type `chararray`.

The next statement breaks each line into words using the (built-in) `TOKENIZE` function. The `FLATTEN` function, also built-in, is used to “un-nest” the result of `TOKENIZE`, a bag, into its constituent tuples. The output produced by the whole instruction is thus a bag of tuples each containing one word of the input. We declare these tuples to contain one field with the name `word`, leaving the type implicit.

We then remove undesired words in line 5, and group the remaining identical words together in line 7. Finally, we count the occurrences of each word and output tuples of the form (“apple”, 3) to the filesystem.

Reverse web-link graph For this example, we assume that the web pages URLs are stored along with their contents in a tab-separated file. A more realistic scenario might involve a custom input function that actually fetches the pages. We also assume the existence of a user-defined function `find_links` that extracts the URLs of links from a page’s content and returns them as a bag.

```

1 pages = LOAD 'webpages'
2       AS (url:chararray, content:chararray);
3 links = FOREACH pages GENERATE url,
4       FLATTEN(find_links(content)) AS target;
5 reverselinks = GROUP links BY target;
6 STORE reverselinks INTO 'resultfile.txt';
```

The first instruction loads webpage URLs and their content, presumably from the filesystem. The second statement gathers links from each page and outputs a bag of tuples of the form (`url`, `target`) for each hyperlink from page `url` to page `target`. Again, `FLATTEN` is used so that the results are not grouped by source page but flattened as a single bag.

Finally, links are reversed by grouping them by `target`, and the results are stored on the filesystem.

5.4 Compilation into MapReduce

5.4.1 Outline

As with other textual dataflow languages (cf. Chapter 3), a textual Pig Latin program defines a corresponding dataflow graph, called a *logical plan* in this context. While parsing statements, the Pig compiler gradually builds this plan as its internal representation of the program. When it encounters a `STORE` or `DUMP` statement, it isolates the part of the plan that leads to that statement by means of a reverse topological sort, and passes it through an optimizer. The optimizer performs some database-like optimizations, such as pushing up filters, inserting projections to prune out unnecessary columns, etc.. Finally, the logical plan is compiled into a *physical plan* to be executed.

The physical plan is then transformed into a series of MapReduce jobs to be executed in sequence, by grouping adjacent operators into mapping or reducing functions. Statements that introduce map/reduce boundaries include `[CO]GROUP` and `JOIN`; each such statement thus corresponds to a different MapReduce job. Other statements such as `FILTER` and `FOREACH` are pushed either in the `reduce` function of a preceding boundary-inducing statement, or in the `map` function of a following one. Inside the `map` and `reduce` function themselves, a regular non-parallel dataflow engine is used to interpret the corresponding parts of the physical plan.

After some additional optimizations, including the attempt to make use of MapReduce’s *combiner* feature, the resulting graph of MapReduce jobs is submitted to Hadoop in topological order.

5.4.2 Example

To illustrate the compilation of Pig Latin, let us take a slightly more complex version of the word counting example presented in Subsection 5.3. In this version (provided in Listing 5.1), the specific words to be counted are extracted from a second dataset rather than being hard-coded. Also, all words are passed through a hypothetical `LOWERCASE` function in order to perform case-insensitive comparisons.

Fig. 5.1a shows the logical plan for this example. Note that it is a direct translation of the textual program into its DFG: all labels have simply been replaced by dataflow links. Fig. 5.1b shows the final result of the compilation, the MapReduce execution plan. The program was split into two MapReduce jobs: one for the join and one for the grouping.

For the first job, the logical join operator was replaced by the “local rearrange”, “union” and “package” operators. These operators, in combination with the implicit sort and shuffle phases between the map and reduce phases, implement a variant of the sort-merge join algorithm (see Chapter 7 for more detail). Also note that the compiler has inserted a `STORE` operation at the end of the reduce function. This is required to

Listing 5.1: More complex word counting program

```

text = LOAD 'mydocument.txt' USING TextLoader()
      AS (line:chararray);
words = FOREACH text GENERATE FLATTEN(TOKENIZE(line))
      AS (word);
lcwords = FOREACH word GENERATE LOWERCASE(word)
      AS(word);
goodwords = LOAD 'goodwords.txt' USING TextLoader()
      AS (goodword);
lcgoodwords = FOREACH goodwords GENERATE
      LOWERCASE(goodword) AS (goodword);
filtered = JOIN lcwords BY word,
      lcgoodwords BY goodword;
grouped = GROUP filtered BY word;
counts = FOREACH grouped GENERATE $0, COUNT($1);
STORE counts INTO 'resultfile.txt';

```

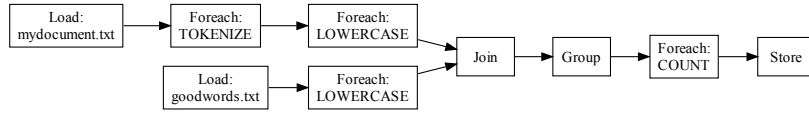
store the intermediary results, as the rest of the computation takes place in a second, separate MapReduce job.

The second job is slightly more complex. Usually, Pig Latin's **GROUP** operator is compiled, like joins, by simply taking advantage of the sort and shuffle phases of MapReduce which essentially perform a grouping themselves. In this case however, Pig as detected that we do not need the actual grouped results, but only the count of each group. It has thus inserted a combining function (see Chapter 4) to compute the intermediary counts just after the mapping phase. The second part of the **COUNT** function (actually a sum) takes place in the reduce phase to sum the intermediary counts together. This optimization can be applied to functions other than **COUNT**, even user-defined ones, as long as they are declared as *algebraic* and describe how they can be split in this fashion.

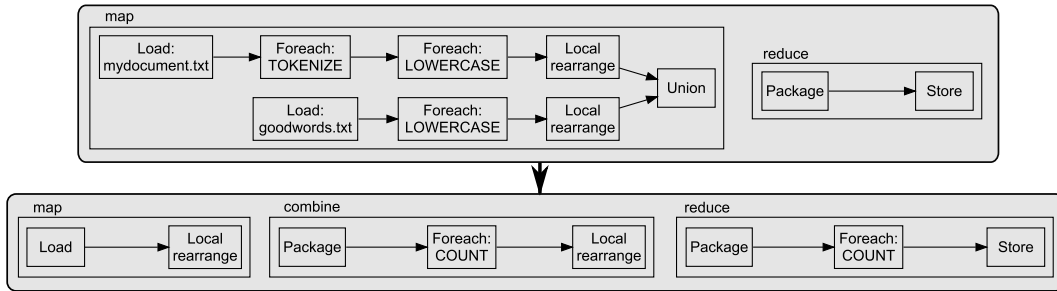
5.4.3 Limitations

We can use this last example to outline some of the limitations of the current implementation of Pig Latin on top of Hadoop. A first one comes from the fact that longer Pig Latin queries must be split into chains of distinct MapReduce jobs. This introduces unnecessary latency in the form of synchronization barriers between subsequent jobs, and I/O overhead due to the unnecessary storing and loading of intermediary data. The underlying reason for this limitation is the inflexibility in specifying a job's dataflow.

Second, this two-level structure of a physical plan inside a MapReduce plan is difficult to optimize in practice. Though this difficulty cannot be formally quantified, the last example outlines a practical case. In that example, the compiler transformed the **GROUP** operator into its own MapReduce job; it normally does this to use the shuffle phase as a way to partition tuples according to their grouping key. In that particular program however, data are already correctly partitioned beforehand, because the shuffle phase of the first job used the same key. As a consequence, the entire program could be better compiled as a single job, with the grouping and counting taking place in the reduce function. While such an optimization would be much beneficial, it proves



(a) Logical plan



(b) MapReduce plan

Figure 5.1: Pig compilation plans (simplified) for the enhanced word counting example

awkward to implement in practice and is currently absent from Pig.

Finally, we can notice that not all the implicit parallelism present in a Pig Latin program is exploited. In our last example, the program starts with two data-parallel branches that end up running sequentially in a map function. The dataflow engine running inside the map and reduce functions also do not take advantage of possible pipelining opportunities. Though these limitations are not overly important in a web-scale setting where the problem is “too parallel” already, lifting them might broaden the scope of Pig Latin to more database-like workloads.

5.5 Summary

We have explained Pig Latin’s current compilation strategy and some of its limitations. Although these limitations could be fixed despite using a MapReduce framework as a back-end, and so are not inherent to MapReduce itself, they would all be naturally lifted by the use of a more general DFG platform instead, as we will show using our own implementation.

Part II

Comparison

Chapter 6

Naiad

Unfortunately, most of the work regarding Dryad has been happening behind closed doors so far. We thus do not have access to a usable implementation, nor to implementation details, nor even to the precise semantics and programming model of the system. In order to fulfill the goals of this thesis, we thus needed to design and implement our own model of large-grain parallel computation based on dataflow graphs. We call the resulting system *Naiad*; it attempts to mimic the principles exposed in [17], sometimes departing from them slightly.

In this chapter, we start by presenting Naiad’s programming model, along with the usual examples. We then outline some of the details of our implementation, and the differences between our system and Dryad. Finally, we present our Pig Latin to Naiad compiler back-end.

6.1 Programming model

6.1.1 Operators

In our system, as with Dryad, a program is specified as a DFG of *operators* interconnected by communication links. Isard et al. [17] state that Dryad operators are “usually written as sequential programs”, but that asynchronous interfaces can be used instead to support “an event-based programming style”.

In contrast, and in order to allow for the as much implementation flexibility as possible, we model operators as purely asynchronous, message-driven programs. Our basic interface reacts to messages of the form (i, d) , which indicates that datum d was handed by the upstream operator connected through the i^{th} incoming edge. We also provide an interface for processing incoming data as batches rather than individual elements, and another for writing synchronous programs that react only when at least one datum is present on each incoming edge, more closely following the traditional dataflow paradigm.

Upon receiving a message, an operator may perform any work it wants, and hand out zero or more messages of the form (o, d) , indicating that datum d should be sent to the operator connected downstream through the o^{th} outgoing edge. The framework will translate this output number o to the corresponding input number for the next operator, and dispatch the message. Finally, an operator may react to a special *EOF* message, signaling the end of data from all of its predecessors. The operator can then perform any final action desired, before the framework broadcasts the *EOF* message

Listing 6.1: Filter operator definition

```

Filter(predicate:  $D \mapsto \{\top, \perp\}$ ) {
  process(input, datum) {
    if(predicate(datum))
      broadcast(datum)
  }
}

```

Listing 6.2: Partitioning operator definition

```

Partition(func:  $D \mapsto \mathbb{N}$ ) {
  process(input, datum) {
    emit(func(datum) % nbOutputs, datum)
  }
}

```

to all successors.

Explicit management of these input and output numbers is superfluous or irrelevant for some operators, but essential for others in order to control data flow. For example, a simple filtering operator will disregard the input number, and broadcast its results to all of its outputs. A partitioning operator will choose the output to send to according to a specific partitioning function. Finally, a relational join operator might use the input number to determine which input relation the data belongs to.

Listings 6.1 and 6.2 describe respectively the filtering and partitioning operators. Let us use this opportunity to introduce the sort of pseudo-code notation that we will use for describing Naiad operators throughout the rest of this document. First note that these operators can be parametrized, in this case with functions (for which we use the standard mathematical notation, i.e. $A \mapsto B$ where A is the domain of the argument and B that of the image). These operators define the **process** procedure which is used to treat incoming messages like described above. This procedure can make use of the following information and tools provided by the runtime framework to specific running instances of the operator:

- the **nbOutputs** and **nbInputs** values, which indicate the number of successors (resp. predecessors) to this operator within the DFG;
- the **emit** procedure which hands out a datum on a particular output number; and
- the **broadcast** procedure which hands out copies of a datum on every output.

Additionally, although not featured in these examples, operators can make use of state variables specific to each instance, and define a **finish** procedure to be called once the *EOF* message has been received on all inputs. Finally, operators can take parameters such as, in these respective examples, the specific filtering predicate and partitioning function. Note that D denotes the set of all data elements.

6.1.2 Connectors

Many operators need to be linked together into a *directed acyclic graph* (*DAG*) in order to constitute a Naiad program. To this end, we use the same connectors as Microsoft’s Dryad, recalled for convenience in Chapter 3, Section 3.4; see [17] for a more detailed and formal explanation. Together, these connectors can generate any directed acyclic graph, and thus any DFG program, given the operators.

6.1.3 Example programs

Simple example

We can now show an example of a simple, complete Naiad program. We assume a dataset comprised of integers; we want to sum all the odd and even numbers separately, and print out the difference between both sums. In sequential pseudocode, this could be written as follows:

```

even = 0
odd = 0
foreach integer i in data:
    if i % 2 == 0:
        even += i
    else:
        odd += i
end.
print(even - odd)

```

Such a program is most likely useless, and could be implemented in ways much smarter than we’re about to show. Nevertheless, it provides a simple demonstration of everything that was explained so far in this chapter, and also outlines some of the quirks of our programming model. Let us start by providing pseudo-code definitions for the different operators that we will use:

<pre> Input() { process(input, datum) {} finish() { foreach integer i in data chunk: broadcast(i) } } Sum() { sum = 0 process(input, datum) { sum += datum } finish() { broadcast(sum) } } </pre>	<pre> Parity() { process(input, datum) { emit(datum % 2, datum) } } Diff2() { result = 0 process(input, datum) { result += (-1 * input) + datum } finish() { broadcast(result) } } </pre>
--	--

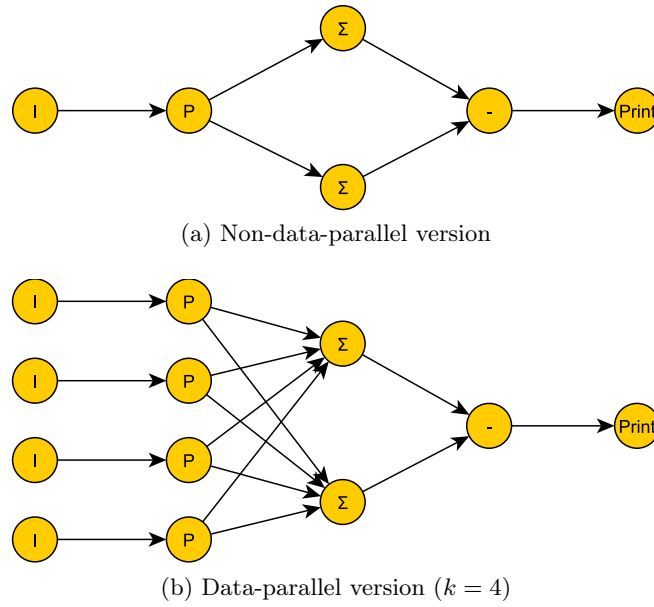


Figure 6.1: Example Naiad program

In this example, **Input** is an operator that acts as a data source; its process routine is empty since it never receives any messages. Instead, the `finish` procedure is defined to inject the input data into the rest of the program. This trick allows data sources to be defined like normal operators despite our “push” model of dataflow.

Let us start by giving a non-data-parallel version of our program (the **Print** operator, not defined above, simply prints out all incoming data):

```
Input >= Parity >= Sum2 >= Diff2 >= Print
```

Input integers are split according to parity, then summed separately; then the difference between the sums is calculated and printed. This program outlines a practical difficulty when programming with this model: that writing a program by composing operators often requires specific knowledge of the operators internal behaviors. In this case, it is necessary to know that **Parity** outputs even numbers on its left output, and that **Diff2** computes left input minus right input. If either of these behaviors were reversed, then the program would compute the wrong result.

It is also required to know that **Parity** expects two output, and **Diff2** two inputs; otherwise the program will work but might not compute anything useful. Also note that, given the behavior of **Diff2**, all **Sum** operators could be replaced by no-ops, but not omitted. As we will later see, other practical programs require inserting no-ops to correctly match inputs to outputs. Though these quirks can make our system somewhat difficult to program by hand, they also allow for the most flexibility, which fits our goal of a good target for compilers.

Let us now create a data-parallel version of the same program, using the same operators. Let us assume that the input data is divided into k chunks, reflecting the assumption of a shared-nothing architecture in which data files are scattered in a distributed filesystem. Let us further assume, for simplification, that different instances of the **Input** operator will automatically read a different chunk of the data (a practice for which our implementation provides programming helpers). Our program can then be expressed as follows:

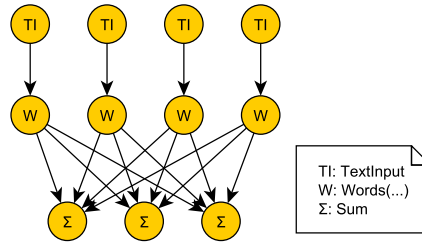


Figure 6.2: Counting specific words in Naiad

```
(Input >= Parity)k >> Sum2 >= Diff2 >= Print
```

Note that this will yield exactly the same program as before for $k = 1$; figure 6.1 illustrates both versions.

Counting specific words

We show here a more concrete example: the counting of specific words, for which we presented a MapReduce version in Chapter 4, Section 4.2.2. We define the following two operators, assuming that we want to count occurrences of a set of m words noted $W : \{w_0, \dots, w_{m-1}\}$.

```

TextInput() {
  process(input, datum) {}
  finish() {
    foreach line l in text data chunk:
      broadcast(l)
  }
}

Words(W : {w0, ..., wm-1}) {
  process(input, line) {
    foreach word u in line:
      if(u ∈ W):
        emit(i : wi = u, 1)
  }
}

```

Given the same `Sum` operator as in the previous example and assuming the input file to be divided in k chunks in the DFS, we can write the programs as follows (note that, for simplification, this program omits to do anything with its output):

```
(TextInput >= Words(W))k >> Summ
```

This example is written in a way that outlines the advantage of the flexibility of DFG models as opposed to MapReduce. The operators used and the program's dataflow are specifically tuned for this particular problem, making this implementation theoretically more efficient than the MapReduce version that we presented, without being significantly more complicated. Indeed, the operations of sorting and grouping that happen between the map and reduce phases are made unnecessary. However, if a

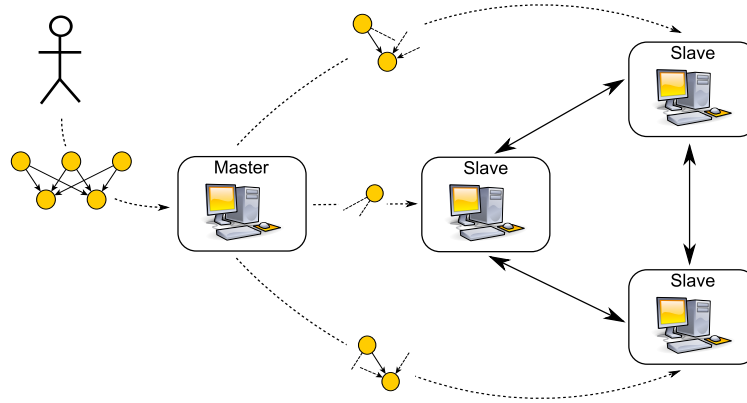


Figure 6.3: Naiad architecture overview

higher level of abstraction was desired instead, one could easily solve the same problem using more generic Naiad operators (e.g. **Map** and **Reduce**).

6.2 Implementation

6.2.1 Target platform

For our experiments, we’ve produced an API for writing Naiad programs in the Scala [29] programming language, as well as a back-end to this API for running those programs on a small cluster of computers. This API closely maps the notation for operators that we used above, and provides Dryad’s connectors for graph building. The target platform is the Java Virtual Machine (JVM), a choice made to ease the subsequent integration with the Pig platform for our experiments.

Given our asynchronous, message-based semantics, it is apparent that message-passing frameworks are a good fit for our implementation. We chose the Akka middleware [16], a scalable and distributed implementation of the *actors concurrency model* for the Scala language. This framework provides us with cluster management, transparent remoting and data serialization/deserialization, and many more features that greatly ease the implementation of our system.

For quick reference, the actor concurrency model can be summarized as follows:

- the entities, called actors, have an internal state and can react to messages
- an actor may react to a message by any combination of the following:
 - changing its internal state,
 - asynchronously sending message(s) to other actor(s), or
 - spawning new actors
- distributed implementations, like Akka, allow flexible and transparent placement of actors on machines of the clusters.

6.2.2 Specifics

A Naiad cluster is composed of a *master* node and any number of *slave* nodes. A program to be executed is handed out to the master node. The process running on this node then assigns individual operators or subgraphs to different slave nodes (fig. 6.3). Not all operators need be assigned to nodes initially; our current implementation uses a fixed *capacity*, i.e. the maximum number of operators that can run simultaneously on a specific slave node.

When assigning an operator to a slave node, the master must do the following:

1. select the slave node among those available,
2. send the operator's program code to the elected slave,
3. provide all nodes currently running a predecessor with the location of the new operator.

The first step is the least trivial, and is very important for obtaining good performance. It depends on the cluster's topology, the current state of the execution, and even the semantics of the specific program. For a trivial example, consider a relational cross product operator, followed by a counting operator. It would be very wasteful to send the output of the cross product on the network; thus, both operators should best be scheduled on the same node. Real-life examples of this problem will be presented in Chapter 7. Our current approach in the software is a combination of general heuristics and user-provided hints. Another, perhaps better approach might consist in some form of auto-tuning of operator placement at runtime, though it might be more complicated to implement, especially along with scalability features like checkpointing.

Slave operation can be summarized as follows. When a slave receives the program code for an operator, it starts up a new actor, and replies to the master with a handle to it. After starting up, if the operator has no predecessors in the DFG, it is considered a data source and the framework simulates an end-of-file situation to trigger processing. This actor will respond to incoming messages by executing the operator's code. During the course of its life, it will also be provided by the master with handles to the actors running the next operators in the DFG. The sequence diagram on fig. 6.4 shows an example of execution of a small Naiad program on a cluster of two slaves of capacity one each.

The operator API we presented earlier models data processing on one datum at a time; however, due to network latency and other reasons, it is much preferable to actually process it in batches. To this end, the `emit` procedure that we have used above does not send the output datum, but stores it in a buffer specific to the output number specified. When this buffer exceeds the specified batch size, it is sent to the target actor, if known. Otherwise, it is either archived to memory or spilled to secondary storage, according to availability.

6.2.3 Differences and limitations compared to Dryad

Our implementation is currently limited to the scope of our (small-scale) experiments, and therefore does not implement important scalability features such as checkpointing and speculative execution, which Dryad does. We also do not support runtime modification of the dataflow graph. Note however that our basic model, like Dryad's, allows for much flexibility in implementing those features.

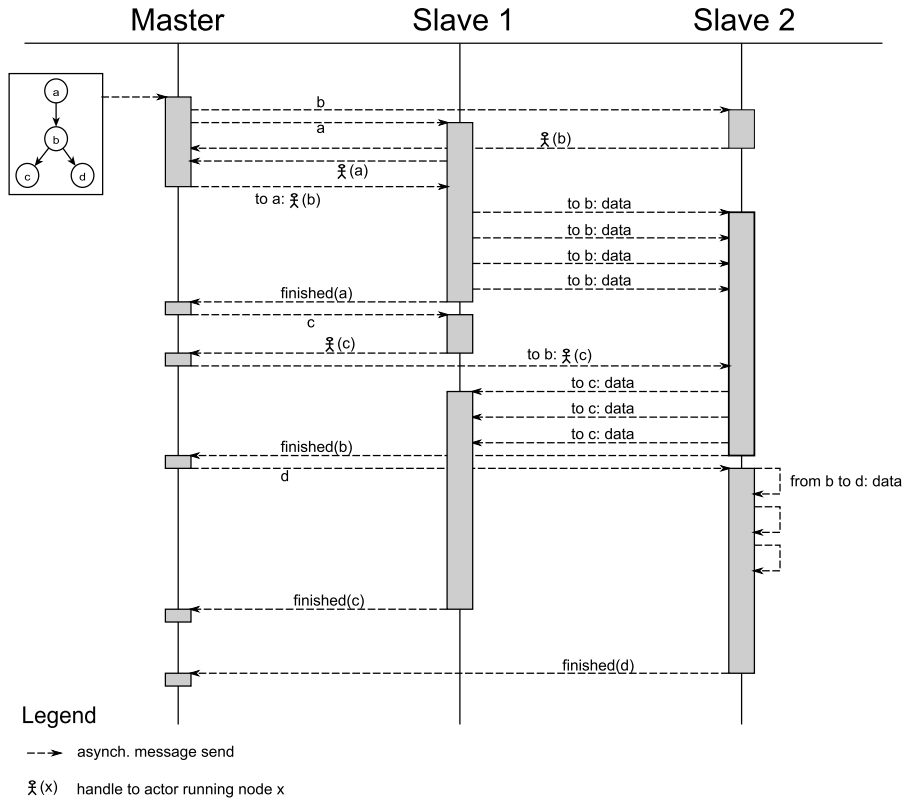


Figure 6.4: Example execution of a small program

Our system also lacks Dryad’s notion of *channel types*: Dryad channels default to temporary files, but can be manually specified by the user as sockets or local pipes. This approach has some quirks [17], and we believe ours to be simpler. Because different operators running on the same node run as threads within the same JVM, local pipes are replaced by reference passing, which is both faster and less quirky. Inter-node communication happens over sockets by default, and stale data that gets too large for main memory is stored to the disk automatically. Our implementation thus transparently covers all of Dryad’s channel types, and the problem of selecting them falls back to that of assigning operators to slave nodes.

Dryad has a feature called *dynamic graph refinement*, which allows writing rules to modify and optimize the dataflow graph at runtime, according for example to the scheduling location of vertices and the amount of data transiting between them. Isard et al. [17] give examples of how this can be useful in practice.

6.3 Pig Latin to Naiad compiler

In order to test our hypotheses that DFG frameworks such as Naiad are a better match than MapReduce as a target for data processing languages such as Pig Latin, we developed a new back-end compiler and runtime interface to Apache Pig targeting the Naiad platform.

Our compiler sits on top of Apache Pig 0.8, retrofitted with structures from previous versions which allowed plugging one’s own back-end compiler and runtime platform in

place of Hadoop. To execute a query, Pig submits to our back-end the optimized logical plan of the query (a dataflow graph of Pig Latin operators). Our compiler then produces a data-parallel Naiad graph from the logical plan and submits it to a Naiad cluster. It completely implements about 90% of Pig Latin’s operators.

6.3.1 Compilation algorithm

The compilation process is very simple due to the fact that both the input and the output are programs modeled as dataflow graphs. We basically need to replace every logical operator in the input with a data-parallel implementation of the operation. The outer compilation algorithm is as follows (in pseudocode):

```

var subplans = new HashMap
var result =  $\phi$ 

for each vertex  $x$  in input in topological order:
  preds = {subplans( $y$ ) for each predecessor  $y$  of  $x$ }
  subplans( $x$ ) = compileOp( $x$ , preds)
  if  $x$  is a leaf:
    result = result || subplans( $x$ )

return result

```

The `subplans` variable remembers the compiled Naiad plan of each Pig logical operator visited. At each step, this plan is generated for the current operator by calling the `compileOp` function with the operator to compile and its already-compiled predecessors. This function will instantiate a Naiad implementation of the supplied operator, link it to its compiled predecessors using Naiad connectors, and return the new Naiad plan so obtained. Finally, the compiled plans of every leaf operator in the input are merged together using the `||` Naiad connector.

The action of the `compileOp` function is determined by which specific operator it is called with. In general, its action is to instantiate a Naiad vertex implementing the Pig Latin operator, replicate it to provide data-parallelism, then link it with its compiled predecessor in the correct way.

Replication How a vertex is replicated depends on the operator to compile. By default, a `LOAD` operator will be replicated to match the number of HDFS blocks in the input file, creating one Naiad vertex for each block; alternatively, a smaller number can be specified manually to group multiple blocks. “Map-like” operators such as `FILTER` are replicated to match the level of parallelism of their compiled predecessor. For other operators, the desired level of parallelism is directly specified in the input Pig Latin program, and defaults to one (i.e. non-parallel).

Linking “Map-like” operators are straightforward to link. Others require to insert additional operations such as partitioning (e.g. `GROUP BY`, `JOIN`). As examples, table 6.1 gives the compilation routines for these different kinds operators, showing how they are replicated and linked. Though here given as distinct, these routines can be generalized to the majority of Pig Latin’s other operators. Some algorithms like exotic joins or sorts however may require more complex, ad-hoc wiring.

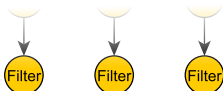
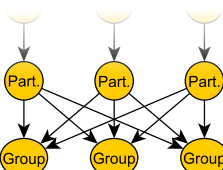
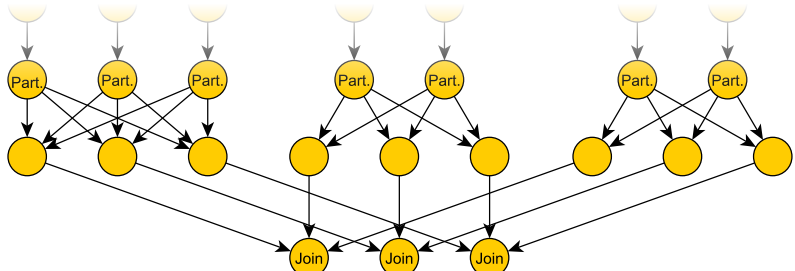
FILTER
<pre>pred₀ >= Filter(...) nb0uts(pred₀)</pre> 
GROUP BY
<pre>pred₀ >= Partition(...) nb0uts(pred₀) >> Group(...) ^j</pre>  <p>Note: j is specified in the input Pig Latin program (here $j = 3$).</p>
JOIN (hash, k-way)
<pre>(pred₀ >= Partition(...) nb0uts(pred₀) >> NoOp^j) (pred₁ >= Partition(...) nb0uts(pred₁) >> NoOp^j) ... (pred_k >= Partition(...) nb0uts(pred_k) >> NoOp^j) >= HashJoin^j</pre>  <p>Note: j is specified in the input program, and k is the number of the relations in the join (here, $j = 3$ and $k = 3$). Chapter 7 describes the join algorithm in more detail.</p>

Table 6.1: Examples: compilation of some operators

6.3.2 Implemented optimizations

Metadata tracking

This direct translation makes it easier to perform additional optimizations in the process. In Chapter 5, we mentioned a specific optimization that was absent of Apache Pig: Pig doesn't track how intermediary data is partitioned or sorted. It will thus sometimes compile MapReduce jobs whose sorting and shuffle phases are unnecessary because of previous instructions. We've mentioned that implementing this with the MapReduce back-end is difficult, as it requires walking across a two-level structure of a physical plan inside a MapReduce plan.

On the other hand, notice that adding this optimization to our compiler is simple. Along with the intermediary compiled plans, we need to remember what sorting and partitioning plans (if any) were used. For example, when compiling a Pig Latin `ORDER BY x` operation, we must remember that data coming out of the resulting Naiad plan will be both sorted and partitioned by its field `x`. Now if the next operation is e.g. `GROUP BY x`, we know that we can chain this operation directly to the results of `ORDER BY x`, without inserting an unnecessary partitioning step.

Our implementation currently supports this for partitioning only, but does not track ordering information at the moment.

Early partial aggregation

In order to be on par with Apache Pig, we also implemented an optimization mimicking the use of the combiner feature of MapReduce: we track `GROUP BY` operators whose results are used only as input to an algebraic functions such as `COUNT` or `SUM`. We replace those with early partial aggregation to compute e.g. intermediary sums, followed by global aggregation to put the intermediary results together. This dramatically decreases the amount of data that has to be globally aggregated, as large numbers of items will be replaced by e.g. their count or sum.

Note that, though our simple implementation imitates MapReduce's combiner, more complex and useful strategies for partial aggregation are possible with DFG frameworks. For example, DryadLINQ [28] uses Dryad's dynamic graph refinement feature to generate aggregation trees at runtime, taking account of the cluster's network topology to minimize network interference.

6.3.3 Other possible optimizations

Though our compiler is most basic, it would be possible to perform more advanced optimizations. Since shared-nothing parallel database engines use similar semantics as Pig Latin and also model query plans as dataflow graphs, most optimizations performed by such engines could be directly transposed to our system. Here follow a few of the simpler examples.

Metadata-based algorithm selection The selection of Naiad operators and algorithms to use for implementing a particular operation could be motivated by metadata such as sorting information. For example, faster algorithms for `JOIN` or `GROUP BY` can be used on sorted data. Currently, a Pig Latin program has to manually specify which algorithm to use, relying on the user to guarantee the required metadata properties.

Parallelism level selection Our compiler chooses the level of data-parallelism of an operation without much consideration. A more sophisticated compiler could estimate the amount of data expected to transit between subsequent operation, and use that to select the appropriate level of data-parallelism. Such a heuristics could also have other uses, such as for influencing the framework’s scheduling decisions.

Sampling-based partitioning For now, most operations that require partitioning are subject to data skew: the partitioning function does not attempt to see if the generated partitions will be of comparable size. The exception is our implementation of Pig Latin’s `ORDER BY` operator which is a parallel sample-sort: the data is sampled and partitioned by range so that each range is of approximately equal size. Though required by the sample-sort algorithm, other aggregation operators might benefit from sampling-based partitioning in the event of data skew.

Dynamic optimizations As stated before, Microsoft’s Dryad, unlike Naiad, includes a mechanism to refine the dataflow graph at runtime, using information not available at compilation time. This could be used as a base to implement some of the optimizations above, as well as other useful ones suggested by Isard et al. [17] such as their strategies for partial aggregation already mentioned.

6.4 Summary

Although Naiad omits important scalability features and thus by no means offers the same industrial strength as Dryad or MapReduce, it serves our purpose by providing a programming model and execution semantics close to Dryad’s. This model has shown us to be practically very suitable as a back-end for Pig Latin. We found that developing this simple compiler was simple and straightforward, especially in light of the complexity of Apache Pig’s Hadoop back-end.

This leads us to believe that DFG models might be generally better than MapReduce as compiler targets, although in a way that is difficult to quantify scientifically. Nonetheless, the following chapters attempt to provide the intuition that this is the case.

Chapter 7

Join algorithms in MapReduce and Naiad

In order to provide concrete arguments for our comparative analysis of MapReduce versus DFG with respect to the compilation of languages like Pig Latin, we will focus on one common operation that these languages provide: the relational join. In particular, when relevant, we will assume the common scenario where we want to join a large log relation L with a smaller reference relation R . For example, L might contain a log of user activity on a website, and R might contain static user data such as name, address, etc...

This chapter thus recalls a series of well-known join algorithms, with outlines of how they could be implemented in MapReduce (mostly based on [3]) and Naiad. We consider unordered and unindexed relations, reflecting the common use case for such systems (as opposed to specialized database management systems). As we will see, most of these algorithms are not strictly MapReduce problems, and require extraneous features and/or framework-specific tweaks for efficient MapReduce implementation.

In the following, we assume that L and R are physically files in a distributed filesystem. They thus each consist in a set of chunks, which we call here *logical splits*, scattered across different machines of the cluster. We use:

- l to denote the number of logical splits in L , and r that in R ; and
- k to denote the number of logical splits in the output (i.e. the number of reducers in a MapReduce setting).

l and r depend on input size, filesystem settings and user-provided parameters. Depending on the algorithm, k may be equal to l or r , or provided by the user.

7.1 Repartition join

The repartition join is a variant of the well-known sort-merge join, and is the most commonly used join algorithm for MapReduce, and the default in Pig. Let us first recall how sort-merge join works. Both relations are first sorted according to the desired join key. Then, an interleaved scan of the sorted relations is performed in a way very similar to the merge phase of the mergesort algorithm, during which corresponding tuples are joined and outputted (see table 7.1 for an example).

Step	R (sorted)	S (sorted)	Output
1	$\rightarrow(a, k, l)$ (b, o, p) (c, m, n)	$\rightarrow(b, x)$ (c, z)	
2	(a, k, l) $\rightarrow(b, o, p)$ (c, m, n)	$\rightarrow(b, x)$ (c, z)	(b, o, p, x)
3	(a, k, l) $\rightarrow(b, o, p)$ (c, m, n)	(b, x) $\rightarrow(c, z)$	
4	(a, k, l) (b, o, p) $\rightarrow(c, m, n)$	(b, x) $\rightarrow(c, z)$	(c, m, n, z)

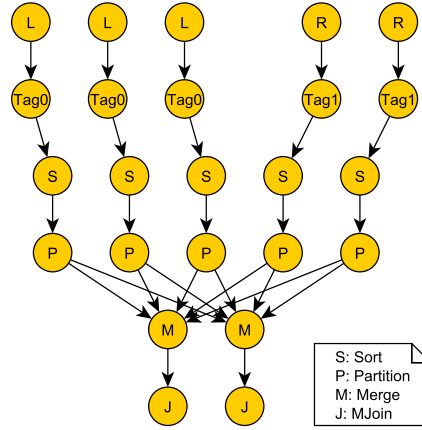
Table 7.1: Example: merge phase of a sort-merge join. For both relations, the first field is used as the join key.

The repartition join algorithm differs in that rather than individually sorting both relations, we sort their concatenation, after tagging each tuple with a number representing which relation it belongs to. We then end up with a single list in which tuples with corresponding join keys appear contiguously. We can then perform a linear scan on this list, thus treating every join key in sorted order. As an example, if we apply the sorting and tagging phases to the relations from the example in table 7.1, we obtain the following list:

$$\begin{array}{l}
 (a, k, l, 0) \\
 (b, o, p, 0) \\
 (b, x, 1) \\
 (c, m, n, 0) \\
 (c, z, 1)
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \end{array} \right\}
 \left. \begin{array}{l} \\ \\ \end{array} \right\}$$

The resulting list of tuples is ordered by join keys (here the first field). The braces show the groups of tuples that will be joined by the final joining operation, i.e. tuples with the same join key and different tags. This procedure can be done as a scan of the list during which we buffer tuples that have the same join key. Once a tuple with a different join key is reached, we output the cross-product between buffered tuples with tag 0 and those with tag 1. We then clear the buffers and continue with the next join key.

This latter procedure can be made more efficient if the relations were to be sorted by both the join key and the originating relation number (as is actually the case in this example). In this way, for any given key, all tuples from the first relation would appear before the corresponding tuples from the other relation, making the final scan easier as only tuples from the first relation have to be buffered (see [3]).

Figure 7.1: Naiad repartition join for $l = 3, r = 2$ and $k = 2$

7.1.1 Repartition join in Naiad

We informally introduce the following Naiad operators:

- **Tag n** tags each incoming datum with number n then immediately broadcasts it
- **Sort** sorts and buffers incoming data, and flushes it all on *EOF*
- **Merge** merges sorted chunks of data (i.e. the same merge operation as in merge-sort)
- **MJoin** performs the scan and join as described above

The repartition join can then be implemented as (see fig. 7.1):

$$(L \gg \text{Tag0})^l \parallel (R \gg \text{Tag1})^r \gg \text{Sort}^{l+r} \gg \text{Partition}^{l+r} \gg \text{Merge}^k \gg \text{MJoin}^k$$

Sort, **Partition**, **Merge** and **MJoin** are all assumed to operate using the desired join key. Note that the **Partition** step is only required to parallelize the **Merge** and **MJoin** operations, i.e. when $k > 1$. Otherwise, the program can be written as:

$$(L \gg \text{Tag0})^l \parallel (R \gg \text{Tag1})^r \gg \text{Sort}^{l+r} \gg \text{Merge} \gg \text{MJoin}$$

Also note that, although we used parallel merge-sort, any distributed sorting algorithm can be used instead; only the **Tag n** and **MJoin** operations characterize the repartition join algorithm. A more general form of the algorithm would thus be:

$$(L \gg \text{Tag0})^l \parallel (R \gg \text{Tag1})^r \gg (\text{some distributed sort}) \gg \text{MJoin}$$

The \parallel connector acts as a concatenation operator between tagged tuples from L and R , which are then sorted and joined.

7.1.2 Repartition join in MapReduce

The repartition join is essentially a MapReduce problem: **Tag n** can be written as a map function, **MJoin** as a reduce function, and the framework itself implements the **Sort**, **Partition** and **Merge** phases as part of MapReduce's shuffle phase. The algorithm could be written as (in pseudocode):

```

function map(key, value) {
  # key = null, value = tuple
  joinkey = get_join_key(value)
  if(input_file == L):
    emit(joinkey, tag(value, 0))
  else: # input_file == R
    emit(joinkey, tag(value, 1))
}

function reduce(key, values) {
  # key = join key, values = tuples
  first = {}
  second = {}
  foreach tuple in values {
    if(gettag(tuple) == 0):
      add tuple to first
    else: # gettag(tuple) == 1
      add tuple to second
  }
  foreach t1 in first {
    foreach t2 in second {
      emit(null, join(t1, t2))
    }
  }
}

```

Because we use the join key as the MapReduce key at the output of the map function, the framework will collect tuples with the same join key together during the shuffle phase. The reduce function will thus be called for each distinct join key, with the list of tuples having that key. These tuples are then separated according to their tag, and their cross-product is outputted.

Note however that this implementation still needs a small framework-specific feature: writing $\text{Tag}n$ as a map function requires:

- that a single job can be assigned multiple (i.e. two) input files, and
- that the map function can inquire which input file its specific running instance is working on, in order to know n .

Without these features, it would still be possible to implement the repartition join by using two map-only pre-processing jobs for tagging the input. This however would impose important overhead, as both relations would effectively have to be duplicated in the distributed filesystem before joining could take place.

7.2 Broadcast join

Often, in practical scenarios, the reference table R is considerably smaller than the log table L . In the particular case where R is small enough to fit in a single node's main memory, a much more network-efficient strategy is possible. By broadcasting R in its entirety to every node, we can avoid moving L on the network altogether, and perform the join locally for each split of L with any efficient sequential join algorithm such as hash join.

7.2.1 Broadcast join in Naiad

Here we use the HashJoin operator defined in appendix section A.1. If $r = 1$, we can simply write the algorithm as:

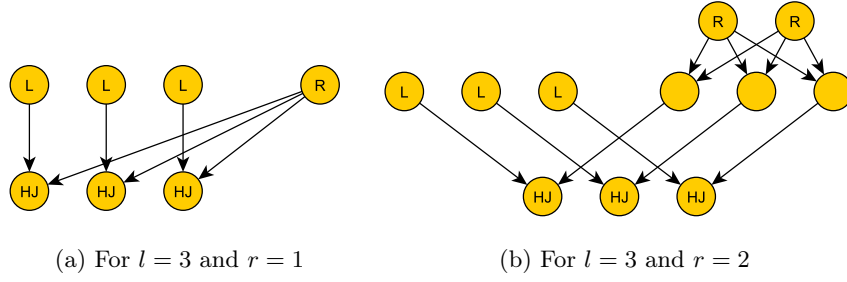


Figure 7.2: Broadcast join with Naiad

```

J := HashJoinl
(Ll >= J) || (R >> J)

```

Here we see the importance of scheduling heuristics and hints to achieve the desired effect. Indeed, efficient execution requires:

- that R , the operator reading relation R , gets scheduled as early as possible, and
- that each **HashJoin** gets scheduled on the same node as its preceding L , in order to achieve the effect of not moving relation L over the network.

If $r > 1$, no-ops need to be inserted in order to correctly match inputs to outputs, as our **HashJoin** uses input numbers to know which relation a tuple comes from:

```

J := HashJoinl
(Ll >= J) || (Rr >> NoOpl >= J)

```

Again, these no-ops will cause no performance drop, but only at the condition that each is scheduled on the same node as its **HashJoin** successor. Fig. 7.2 illustrates both versions.

Note that these implementations may create unnecessary network overhead when several **HashJoin** operators end up running on the same node (a likely situation since l is usually larger than the number of nodes). In this case, the framework will wastefully end up sending R multiple times on the same network link. This can be solved, for example, by creating a new kind of operator, of which each instance reads the entire R relation and caches it to the local host if it isn't already. Other such tricks can be used instead, such as copying R beforehand, or increasing the distributed filesystem's replication factor. There is however no natural way to express such a broadcasting operation in Naiad, nor does it seem possible to optimize such broadcasts automatically.

7.2.2 Broadcast join in MapReduce

Broadcast join can be easily implemented as a map-only job, without the sort, shuffle and reduce phases. This however requires the framework to allow defining, in addition to the map function, an *init* procedure with the ability to set up state variables to be accessible across distinct invocations of the map function within the same task. This procedure would fetch and hash the R relation. Then, tuples of L would be streamed through the map function, each time joined with the corresponding tuples in R . Such an implementation is provided by Pig under the name of *fragment-replicate* join, and

behaves most identically to our Naiad implementation above. Blanas et al. [3] provide a slightly more complex and better optimized version.

Note that the same limitation arises that we need to locally cache R at every node if we don't want to transfer it an unnecessary number of times. Again, a number of tricks can be used, but it is worthy to note that neither MapReduce nor DFG allow us to express this broadcasting operations in a natural way. This hints at a common limitation of both paradigms, which we expand upon in the conclusion of this thesis.

7.3 Semi-join

If the reference table R is very large and contains many entries, it is possible, depending on the situation, that only a minority of those entries are actually referenced in the log table L . For example, R might contains data about all users of a web application, while L might represent a few hours of activity logs, and thus reference relatively few users despite being huge. Then, broadcasting R would imply broadcasting much more tuples than is actually necessary. We can proceed more efficiently by first filtering R to keep only tuples that have a match in L , using the following steps:

1. gather the set S of all distinct join keys in L , which we expect to be relatively small;
2. broadcast S next to every split of R , and produce R_l by taking only tuples of R whose join key appear in S ;
3. join L with R_l using the broadcast join algorithm.

7.3.1 Semi-join in Naiad

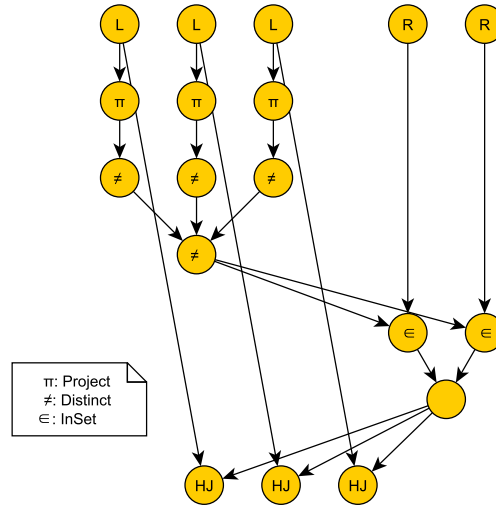
We use of the following Naiad operators which we shall describe but not define:

- **Project**, shortened π , which extracts and outputs join keys from each incoming tuple;
- **Distinct**, shortened \neq , which only outputs distinct values coming from its inputs, for example by building a hash table; and
- **InSet**, shortened \in , which takes a set S of values on one input, and outputs tuples from the other input whose join keys appears in S .

The semi-join algorithm can then be expressed for example as follows (see fig. 7.3 for illustration):

```
J := HashJoinl
A := Ll
B :=  $\in^r$  >> NoOp
(A >=  $\pi^l$  >=  $\neq^l$  >>  $\neq$  >> B) || (Rr >= B >> J) || (A >= J)
```

Note that, once again, correct assignment of operators to cluster nodes is crucial to achieve the desired effect. Also note that the algorithm exhibits the same problem has the previous one, namely that it contains unoptimized broadcasting operations, which prevent us from producing a useful implementation given the current state of our runtime framework.

Figure 7.3: Semi-join in Naiad for $l = 3$ and $r = 2$

7.3.2 Semi-join in MapReduce

We describe here the MapReduce implementation of the semi-join as given (in more detail) in [3]. This algorithm is split in three distinct phases, each consisting of a MapReduce job.

First phase is a MapReduce job to extract all distinct keys from L . The map function uses a hash table, initialized in the init procedure, to output only records with distinct join keys. The reduce function further eliminates duplicates and output the complete set of distinct keys.

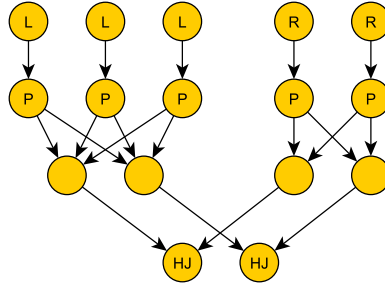
Second phase is run as a map-only job over R , again without sort, shuffle or reduce phases. The init procedure loads the output of the first phase, and stores it in a hash table. Then, the map function outputs each record from R whose join keys are present in the hash table.

Final phase is the broadcast join between L and the output of the second phase, and thus also a map-only job as described above.

Blanas et al. [3] find this algorithm inefficient, because of the overhead of running three MapReduce jobs, and especially of scanning L twice. It would be useful to know if using a DFG framework might improve this; we have however been unable to produce an efficient Naiad implementation, as will be discussed in Chapter 8.

7.4 Hash join

Though classically one of the most used join algorithms both in sequential and parallel settings, it is the least natural to write in MapReduce. Indeed, while the repartition join was the most natural because of (almost) being a purely MapReduce problem, hash join is the most alien and not a MapReduce problem at all.

Figure 7.4: Hash join for $l = 3$, $r = 2$ and $k = 2$

As a recall, parallel hash-join consists in hash-partitioning both input relations according to join keys, then running a sequential hash-join on each pair of corresponding partitions.

7.4.1 Hash join in Naiad

Unlike its MapReduce counterpart, the Naiad version of hash join is pretty natural and straightforward, given some of the same operators we have been using so far (illustration on fig. 7.4):

```

J := HashJoinl
(Ll >= Partitionl >> NoOpk >= J) || (Rr >= Partitionr >> NoOpk >= J)

```

We assume `Partition` to operate on the desired join keys, and to use the same hash function as the `HashJoin` operator. Like before, `NoOps` are overhead-free at the condition that they be scheduled correctly by the framework.

7.4.2 Hash join in MapReduce

As we mentioned, hash join is more awkward to formulate as a MapReduce problem, for reasons that outline the relative rigidity of the MapReduce platform. Indeed, the issue is that while it could make good use of the map, shuffle and reduce phases, it doesn't require the implicit sorting which happens in between these phases, and which cannot be explicitly skipped on implementations such as Hadoop. This is why sort-merge-like joins are often used with these platforms, despite the known fact that hash join usually works better in practice. To the best of our knowledge, no implementation of hash join for MapReduce has been proposed in literature yet.

It can, however, be implemented as a sequence of three map-only jobs, that themselves implement the required shuffle phase. The first two jobs perform the partitioning operation for the respective input relations, and write the resulting partitions directly as intermediary files instead of yielding them to the framework. The third job, instead of getting its input data from the framework, gathers and reads itself the corresponding partitions of L and R to perform the join (see fig. 7.5).

Note that this is bad usage of the MapReduce framework, as we need to either discard or re-implement core features of the platform, such as managing input and output, which are especially non-trivial with respect to scalability requirements such as fault tolerance. For maximum parallelization, we also need to manage job scheduling manually: the first two jobs can be run simultaneously, but must be finished before the

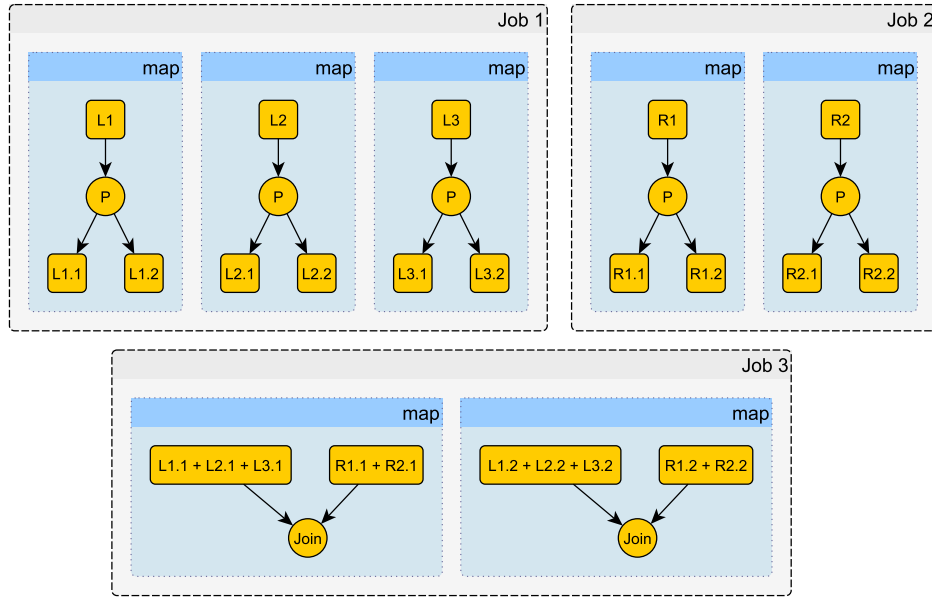


Figure 7.5: Hash join in MapReduce

last one can start. Still, the fact that it *possible* (if convoluted) to do all of this will be used for our further comparison of MapReduce against DFG frameworks.

7.5 Summary

We have outlined four common types of join algorithms. Although all of them can technically be implemented in MapReduce, we saw that only the repartition join algorithm really has a MapReduce dataflow; the other ones need to work around the MapReduce model somewhat in order to fit.

Although we focused on joins, other operations can have efficient and useful algorithms that are not strictly MapReduce problems. For example, Pig Latin’s default algorithm for **ORDER BY** operations is a parallel sample-sort, another algorithm with a non-MapReduce dataflow.

In the next chapter we show that, although not necessary less efficient from a performance point-of-view, this kind of usage of MapReduce amounts to using a DFG framework that is particularly difficult to program.

Chapter 8

Analysis of MapReduce versus DFG

In Chapter 7, we have seen how common join algorithm can be (and are being) implemented in MapReduce. However, as will be further detailed in this chapter, these algorithms are not native MapReduce problems and require “cheating” on the pure MapReduce programming model, by either requiring specific framework features or working around them.

This chapter will start with showing that if we accept this cheating, then MapReduce is equally powerful as DFG programming models for expressing scalable parallel algorithms. In particular, we will show that these cheating tricks themselves contain the differences between both paradigms. Then, we will argue in favor of DFG models against MapReduce for the purpose of compiling higher-level languages such as Pig Latin, using experimental results from our platform as supporting data.

8.1 MapReduce cheating tricks & equivalence to DFG

Tricks

We have seen that efficient MapReduce implementation of popular join algorithms requires, among others, the following four tricks:

1. *Sideways input/output*: using effectful map and/or reduce functions to read and/or write data besides the framework-assigned key-value pair streams. Used by the broadcast join, semi-join and hash join algorithms.
2. *Map-only jobs*: building jobs that do not fit into a map-reduce structure as a sequence of map-only jobs without shuffle or reduce phases. A map-only job is a MapReduce job with no reduce function, and thus consists only in the map phase whose output is considered as that of the job. Used by the broadcast join, semi-join and hash join algorithms.
3. *Task differentiation*: getting different tasks from the same job to perform different work, i.e. to run different map (or reduce) functions. Used by the repartition join and hash join algorithms.
4. *Job-level scheduling*: for algorithms written as multiple inter-dependent jobs, obtaining job-level parallelism by scheduling these jobs according to their constraints of precedence. Used by the hash join and, in some measure, the semi-join algorithms.

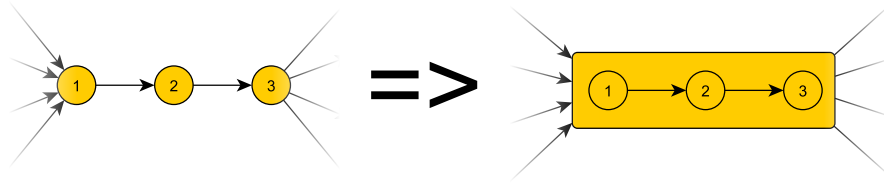


Figure 8.1: Embedding linear lists into single vertices

By nature, any MapReduce framework allows the use of the first trick. The Hadoop framework explicitly provides means of using the last three. Pig currently leverages all four of these tricks.

Equivalence

We will now show that any DFG program can be transformed into a functionally equivalent and equally scalable set of MapReduce programs that makes use of these four tricks, by performing the following steps.

1. We transform each vertex of the DFG program into a map task.
 - Vertices with more than one incoming and/or outgoing edges can be rewritten using trick no. 1 to simulate the additional channels.
 - Top-level (input) vertices are replaced with the MapReduce framework's means for data input. Hadoop for example allows to define custom data input classes, which can be used to encapsulate a Naiad input vertex.
 - As an optimization, any subgraph of the DFG program that is effectively a linear list can be treated as a single vertex performing the same computation (e.g. see fig. 8.1).
2. Then, these tasks can be grouped into map-only jobs, using tricks no. 2 and 3.
3. Finally, these jobs can be submitted to the MapReduce framework with the correct precedence constraints as per trick no. 4.

Synchronization constraints of the resulting program depend on the assignment of vertex-emulating map tasks into map-only jobs. It is however always possible to do this repartition without adding any constraint not present in the original DFG program; the simplest way to do so would be to put each task in its own map-only job.

We say that the original and transformed versions of the program are equally scalable because the level of parallelism of both versions are identical and limited only by the amount of data. Any additional parallelism in the DFG version, such as that offered by pipelining, can only contribute by a constant factor. Because of producing data-parallel programs, both MapReduce and DFG can be considered maximally scalable.

Note that the inverse transformation of a MapReduce program into a DFG one is trivial and require no sort of tricks, as MapReduce really is a special case of DFG.

Example Let us provide an example of this transformation using the word counting program described in Chapter 6, whose graph is recalled for convenience in fig. 8.2a. Fig. 8.2b shows the results of the first transformation step. W and Σ vertices have each been made into a map function; each TI vertex has been integrated as data input into the map task of its W successor.

Fig. 8.2c shows the result of the second transformation step: map tasks have been grouped into two map-only jobs. Note that this arrangement of jobs respects the synchronization constraint of the original program; namely, no Σ operator can start before all the W ones have finished processing. These jobs can then be submitted to the MapReduce framework in the right order as per the third step.

When we presented our Naiad version of this word-counting program, we mentioned how it was theoretically more efficient than the naive MapReduce version that we have shown in Chapter 4. For the anecdote, note that this is still the case with this transformed version, although no one would likely write it that way in MapReduce.

For another example of this transformation, consider the hash join algorithms presented in section 7.4: the MapReduce version described and illustrated closely corresponds to the Naiad version with this transformation applied.

8.2 Theoretical argumentation

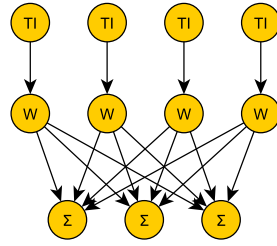
Given this equivalence, one can wonder: why use MapReduce as a target platform for higher-level languages in the first place? Indeed, by using these tricks, these languages are effectively targeting an awkward and convoluted implementation of DFG, making compilation and optimization of programs more tricky than it should be. Why not target a DFG framework instead?

An instruction set analogy

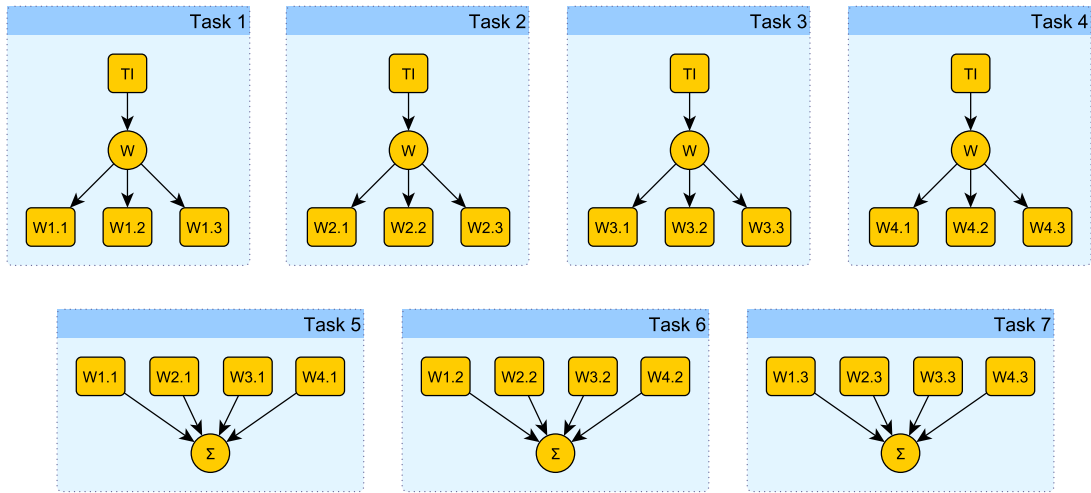
The reason why MapReduce became so popular is that it is easy both to implement in a highly scalable way, and to write programs for. With respect to that last aspect, an analogy can be drawn with CISC and RISC instruction sets. CISC processors were intended to be programmed by humans; to that end, they feature complex instructions that performed large multi-cycle operations. Likewise, MapReduce was designed for human programmers and performs large jobs that each include sorting, partitioning and aggregation.

RISC instructions sets on the other hand were designed specifically for use as compiler targets. They provide a set of basic, orthogonal instructions, several of which must be chained to replicate the effect of a complex CISC instruction; in effect, they provide a lower-level programming model. This is similar in principle with the way DFG is more general and lower-level than MapReduce: a larger DFG program is required to replicate the effect of a single MapReduce one.

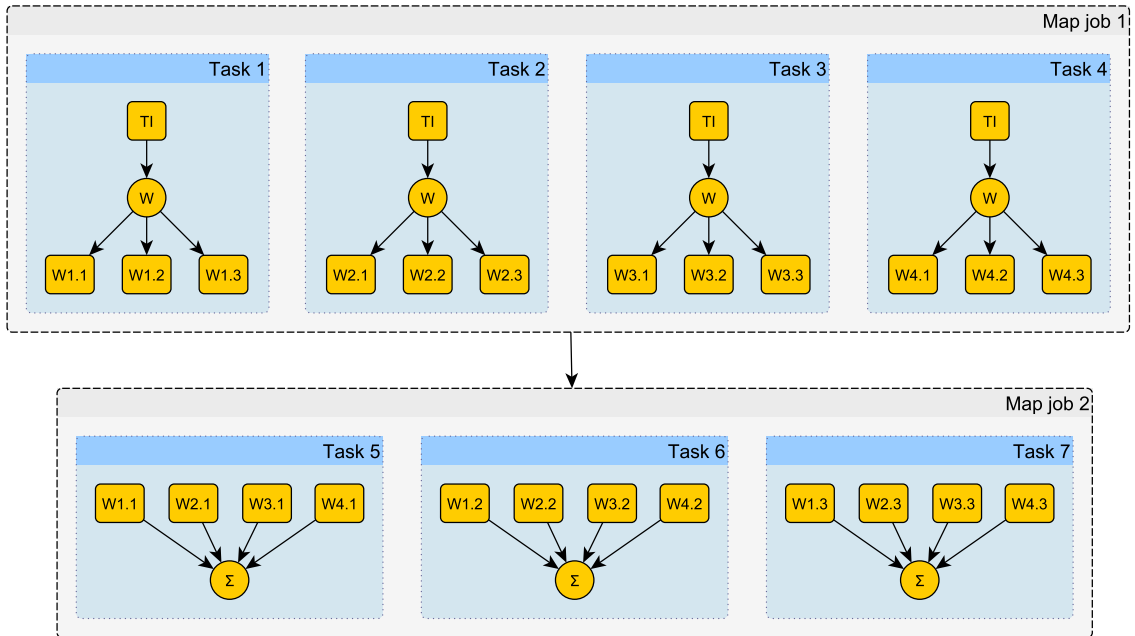
This analogy provides the intuition as to why we believe DFG to be a better choice than MapReduce as a compiler target for higher-level languages. The fact that we were able to produce a working compiler for Pig Latin with only about one man-month of work supports that hypothesis. As we will see, experimental results indicate that the resulting compiler is at least as efficient as the one targeting MapReduce.



(a) Original program



(b) First step



(c) Second & third steps

Figure 8.2: DFG to MapReduce transformation example

Implementation flexibility

While we argue that DFG makes writing a compiler simpler, it can also be argued that implementing the DFG framework itself is more complex than with MapReduce. There is a larger degree of freedom in design, and non-trivial questions must be answered depending on the end goal of the framework. How to manage vertex scheduling? How do vertices exchange data? Do we prefer aggressive pipelining, or frequent checkpointing? Some choices will lead to a faster but less scalable implementation and vice versa. Though implementing MapReduce gives rise to the same dilemmas, they are much simplified by the constrained programming model.

On the other hand, such flexibility may be useful to broaden the applicability range of programs written for DFG frameworks. It would be useful, for example, to be able to transform a high-speed, low-latency parallel database system into a highly scalable data mining system by only tuning framework parameters. Pig Latin, currently limited to the latter kind of system, could be chosen as either depending on the situation.

MapReduce, DFG and parallel databases

To better illustrate that last point, let us delve a little into shared-nothing parallel databases and how they differ from MapReduce systems.

Database systems are usually designed for high speed and low latency rather than high scalability. They completely manage the data they are entrusted with; they manage how it is sorted, indexed, formatted and partitioned across the cluster, with the objective of making queries fast. They allow queries composed of operations from a fixed set, which they know how to best optimize based on their data storage schemes. They are usually designed to run on smaller, more reliable clusters, and thus make little or no use of scalability features such as replication, checkpointing or speculative execution which increase latency.

MapReduce systems on the other hand are designed for ad-hoc processing of any kind of data. Though the user may decide to structure, format, sort and partition their data to allow for more efficient treatment, the framework itself makes no use of this information; optimizations based on such metadata are left to the user. They allow any query expressible as map and reduce functions, which themselves can contain anything. Following Google's original implementation, they have no regard for latency but focus on massive scalability; if the system is full or too slow, one can blindly add more computers to increase its processing capacity, thanks to the scalability features mentioned above.

Thus, the difference between both kinds of systems may be divided in three aspects:

1. the extent to which the system manages how data is physically stored;
2. the constraints the system puts on what queries can be expressed; and,
3. the assumptions that the system makes about the target cluster.

Many practical systems attempt to blend the different approaches to these aspects. Systems like Hive [24] or HadoopDB [2] bring MapReduce closer to database systems with respect to the first and second aspect. Database systems often allow user-defined function, and some [13] even include map and reduce instructions, bringing them closer to MapReduce with respect to the second aspect. Good database systems allow some degree of control over the third aspect.

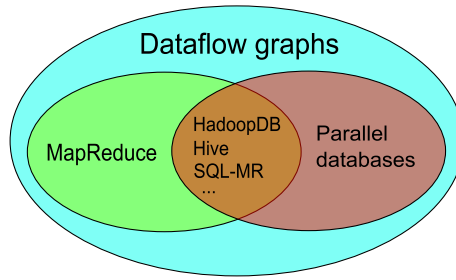


Figure 8.3: MapReduce, DFG and databases

These differences are thus in no way intrinsic and merely result from different goals and scopes. Several papers (e.g. [10, 19, 23]) comparing MapReduce with parallel database systems indeed conclude that both kinds of system differ by their application scope; MapReduce outperforms databases on simple transforms of huge datasets on very large clusters, while databases shine for complex queries on smaller datasets and clusters. We argue that DFG frameworks are general enough to cover both situations and everything in-between, as they can be easily used to implement either kind of system.

Example: HadoopDB As we mentioned above, attempts to broaden the scope of either kind of system are made by integrating features from one kind into the other. Let us take HadoopDB as a popular example of such a “hybrid” system. Its goal is to blend the excellent scalability of Hadoop/MapReduce with features from database systems enabling more efficient processing of structured data. It allows to store relational tables with metadata such as schemas, indexes, and so on to allow generating more efficient query plans. It provides features like updates, ubiquitous to database systems but normally impossible with Hadoop due to its heavyweight HDFS filesystem.

To provide all this on top of Hadoop and HDFS requires tricks, compromises, and a complex architecture. We can also expect HadoopDB’s query compiler to be subjected to the same limitations as Pig Latin’s. As a result, while HadoopDB succeeds in providing an extremely scalable database system, it remains much less efficient than actual parallel databases at smaller scales, with homogeneous clusters and in the absence of failures.

By taking features from one kind of system and integrating it into the other, we do obtain a useful hybrid (e.g. with HadoopDB, a system faster than MapReduce and more scalable than a database). However, we believe that this kind of approach is not the best. Instead, we believe that a flexible, general-purpose DFG framework would constitute a better basis for both kinds of systems by allowing to cherry-pick desirable features.

For example, let us say that we want a highly scalable database system. We think of HadoopDB’s approach as taking a highly scalable system and “tweaking” it into being a database. Our approach would be to have a DFG-based database system, and make it scalable by increasing checkpointing, reducing task granularity, enabling speculative execution, and so on.

A simplistic experiment will be made to illustrate the point that a DFG framework can be easily adapted to different kinds of requirements, by performing the reverse operation: taking a Pig Latin program and running it in a way that encourages performance

over scalability, by assuming a reliable and homogeneous cluster.

8.3 Experiments

In this section, we present two experiments with our Naiad system and its Pig Latin compiler. The first experiment compares the runtime performance of our compiler against the Apache Pig compiler targeting Hadoop. The second experiment outlines our point about the relative flexibility of DFG systems.

8.3.1 Setting

Cluster configuration

Our experiments were run on a small cluster of four computers linked by a 100Mbit/s Ethernet LAN. Each computer has a dual-core CPU running at a frequency of 800MHz and a gigabyte of main memory. Software-wise, each system runs a Linux installation, the Hadoop HDFS filesystem software, and either Hadoop MapReduce or the Naiad runtime depending on the experiment. A fifth computer is used as the coordinator, i.e. the Hadoop and Naiad master node.

Data

Since our experiments involve joining a reference table with a log table, we wrote routines to generate suitable data for these tables. The reference table, `users`, has two fields: a unique `name` field, and a larger `userdata` field of random padding. The log table, `log`, has three fields: a `user` field referencing the users table, a numeric field `x`, and a `logdata` field of random padding.

The generating procedure can be parametrized to specify the on-disk file size of either table, as well as the fraction of users from the `users` table that will be referenced in the `log` table (thus allowing to choose the selectivity of the join between them). In addition, we can specify a fraction of entries in the `log` table which correspond to no user in the reference table. For both experiments, we used values of respectively .8 and .1 for these fractions.

Both tables reside on a HDFS distributed filesystem and are formatted as comma-separated values: each line represents a tuple, and commas separate tuple fields

8.3.2 First experiment: compilation & optimization

Evaluation metric

In this experiment, we want to compare our compiler's performance against that of the one targeting Hadoop. The difficulty is that we do not want to measure performance difference between the Hadoop and Naiad platforms, but between programs compiled with Pig-Hadoop and Pig-Naiad.

Hadoop and Naiad themselves have architectural differences that make them incomparable in performance. Hadoop is industrially strong and designed to run reliably on clusters of thousands of computers. Naiad was developed for the very purpose of these small-scale experiments, doing away with failure tolerance and checkpointing.

To palliate this problem, we will only compare either framework with itself. The running time of the Pig Latin query we use for this experiment will be compared to

that of a baseline, hand-optimized MapReduce program specific to the task. Since we have our own implementation of MapReduce on top of Naiad, we are able to run this baseline program on either platform, and compare the execution time with those of the Pig query running on the same platform. This method factors away the difference between the platforms and exposes that between the compilers. The following test hints this measurement to be sufficiently accurate for our need.

Each test is ran many times, but rather than an average, we keep the lowest obtained running time as the result. This is to eliminate interference due to chance or framework quirks, and only retain the running time of the program itself. For example, the scheduling strategy of our framework is not perfect and not always deterministic, yielding some small variance in a program’s running time. By keeping only the best run, we simulate a framework with a close-to-perfect scheduling behavior.

The results in this section are measured with a 4 gigabytes log table and a 1 gigabyte reference table. Given the padding settings used, this amounts to about one million users and four million log entries.

Test query

The query for this experiment consists in joining the `users` and `log` tables, grouping the result of the join by user, and computing the sum of the log field `x` for each user. This is expressed in Pig Latin as follows:

```
users = LOAD 'users' USING PigStorage(',')
      AS (name:chararray,userdata:bytearray);
users2 = FOREACH users GENERATE name;
log = LOAD 'log' USING PigStorage(',')
     AS (user:chararray,x:double,logdata:bytearray);
log2 = FOREACH log GENERATE user,x;
joined = JOIN users2 BY name, log2 BY user PARALLEL 16;
grouped = GROUP joined BY user PARALLEL 16;
sums = FOREACH grouped GENERATE group,SUM(joined.x);
STORE sums INTO 'sums' USING PigStorage(',');
```

As explained in Chapter 5, the Apache Pig compiler will transform this query into a sequence of two MapReduce jobs: one for the `JOIN` operation, and another for the `GROUP` operation. This is because Apache Pig lacks the optimization to make it realize that the tuples are already grouped by user after the join.

This query is useful with respect to our evaluation metric, as it can actually be written as a single, efficient MapReduce program. Instead of computing the join, then the group, then the sum, we can write the following MapReduce program than computes the sum directly:

	Hadoop	Naiad
Baseline MapReduce	220s	105s
Pig Latin	390s = 1.77×	190s = 1.80×

Table 8.1: Results for the first test

```

function map(key, value) {
  # key = null, value = line of text
  fields = split value around ','
  if(input file == 'users'):
    emit(fields[0], '1')
  else: # input file == 'log'
    emit(fields[0], '2' + fields[1])
}

function reduce(key, values) {
  # key = user name, values = tuples
  sum = 0.0
  ref = false
  foreach value in values {
    tblnum = first character of value
    data = rest of value
    if(tblnum == '1'):
      ref = true
    else: # tblnum == '2'
      sum += parse_number(data)
  }
  if(ref && sum > 0.0):
    emit(null, key + ',' + sum)
}

```

This is a variant of the repartition join in which, rather than joining tuples, we immediately sum values of `x` for entries of the `log` table, and output the sum if an entry from the `users` table has been found.

First test: validating the metric

To get the feel that our chosen evaluation metric is suitable, we first tweak our Pig to Naiad compiler such that it produces an output program as close as possible to what the Apache compiler would produce. To that end, we force the use of the repartition join as a join algorithm, and disable our “metadata tracking” optimization not present in Apache Pig. Hadoop’s combiner optimization is used, as well as our equivalent early partial aggregation. Table 8.1 gives the obtained results.

These results mean that the Apache Pig compiler yields a program that is 1.77 times slower than the baseline MapReduce program executed on Hadoop, and our Pig to Naiad compiler one that is 1.8 times slower than the same baseline program running on our own implementation of MapReduce on top of Naiad. From this, we interpret that both compilers produce similar results when compiling programs to use the same algorithms and optimizations. The slightly worse figure for Naiad hints at a small overhead in our implementation of Pig Latin operators.

Several versions of this “calibration” test were devised, each time yielding very close results. This encourages us to believe that our evaluation metric is relevant.

Second test: hash join algorithm

Apache Pig does not provide the hash join algorithm, and we have explained in Chapter 7 why MapReduce is ill-suited for implementing it. Since hash join is known to usually perform better than forms of sort-merge joins, it might constitute one of the most obvious reasons why to implement Pig Latin on top of DFG instead. Here we measure the improvement obtained for this query by using hash join instead of repartition join.

We measured a running time of 178 seconds, or 1.69 times worse than the baseline program, i.e. about 7% better than with the repartition join. We conclude that, as was expected, hash join is beneficial for such queries.

Third test: metadata tracking

For this last test, we enabled our metadata tracking optimization. This means that no useless partitioning step will be inserted between the `JOIN` and `GROUP` operations, yielding a program closer to the baseline MapReduce one. Though a useless partitioning step does not seem more harmful than a no-op at first glance, the difference lies in the fact that, since the framework cannot know that only one outgoing link of a partitioning operator will carry data, it has no incentive to schedule that particular link locally. The optimized version will thus avoid any data serialization and network transfer between the `JOIN` and `GROUP` phases.

We measured a running time of 165 seconds, or 1.57 times better than the baseline program. This is thus about 8% better than without the optimization, and 15% better than the version with the repartition join. Though relatively small with this query, the improvement could be much more dramatic in situations where larger amounts of data must transit between the join and group operators.

Conclusion

The results of this first experiment suggest the following of our points:

- DFG is as good a back-end as MapReduce, given similar compilers;
- DFG allows implementing useful algorithms that are ill-suited to MapReduce; and,
- some useful optimizations are easier to integrate into a compiler targeting DFG (cf. chapters 5 and 6).

More importantly, a relatively small amount of work yielded a compiler that is as or more efficient than the much more complex one from Apache Pig (at least for this particular query). Although our compiler is slightly incomplete and not industrially strong, we can grossly compare this difference in complexity by observing that Apache Pig's physical layer is composed of about 25 thousands of lines of Java code, while ours is about one thousand lines of Scala code (n.b. these counts exclude blank lines and comments).

8.3.3 Second experiment: framework flexibility

We mentioned our belief that the flexibility of implementation of DFG frameworks can broaden the application scope of higher-level languages targeting them. One of the

reasons for that comes from the possibility of applying pipelining: rather than each DFG vertex consuming its whole input and producing its whole output, we can have them processing smaller batches, enabling them to work in an “assembly line” fashion. This can provide the following benefits:

- pipelining (if well-tuned) can lessen processing times by increasing resource utilization;
- a query can start producing early results a long time before its completion, which can be useful in real-time scenarios, e.g. when building a webpage to show query results;
- it allows *continuous querying*, i.e. queries over moving data streams rather than fixed datasets (as in e.g. [6]).

Although pipelining is partially applicable to MapReduce [7], it is limited by the sorting step between the map and reduce phases. Sorting is indeed a blocking operation, i.e. it cannot produce results until all input data is known. In contrast, a DFG program does not mandate such a blocking operation. We present the following small experiment to show how an implementation of Pig Latin on top of DFG can provide it with the benefits outlined in the first two points above (we shall not delve into the last one).

Data, query & setting

We use the same data generating procedure as in the first experiment, but with a smaller `log` table of only one gigabyte (or about one million log entries). The query is a simple relational join; it is the same as before without the last two operations of grouping and summing. Since our hash join implementation is fully pipelining, the complete query can be run as a pipeline. In this scenario, all Naiad vertices in the compiled query are running simultaneously from the start. We can make that happen by tweaking some of Naiad’s parameters.

1. We reduce the parallelism level of the `JOIN` instruction from 16 (as in the first experiment) to four, the number of computers in our test cluster.
2. We also force the parallelism level of each `LOAD` operation to four (cf. Section 6.3).
3. We remove the limit on how many operators can be hosted on a Naiad slave node at the same time.
4. Finally, we reduce the batch size parameter (cf. Section 6.2) to favor rapid descent of data into the pipeline.

From the same query, we thus obtain a program that is optimized towards latency rather than scalability, and provides the benefits of pipelined execution.

Experiment

We compare running times of this query with and without the parameter tweaks listed above, yielding the results in table 8.2. We found the pipelined version to be faster and much more predictable than the normal version. Also, as expected, we found the pipelined version to start producing results much sooner.

	Normal	Fully pipelined
Best running time	53s	47s
Average running time	59s	48s
Earliest results	42s	17s

Table 8.2: Results of the second experiment

Note that these results must be taken with a grain of salt. The performance of the non-pipelined version depends much more on the scheduling strategies of the framework than that the pipelined version. Additionally, pipeline tuning is a delicate subject in itself, into which we haven't much delved in the scope of this thesis.

Conclusion

Though we cannot make any strong conclusion from this experiment, it outlines the fact that DFG frameworks allow more variation in the way programs are run than MapReduce. As stated before, fully pipelined execution can bring more uses to a language like Pig Latin than what it was initially designed for.

It is however important to remember that full pipelining is a trade-off. It removes important scalability properties like the tolerance of failures, stragglers, network congestion, etc... Whatever schemes we use to counter these events, any of them happening on even a single computer will inevitably slow down the complete program.

Full pipelining also places additional constraints on memory. For example, recall that our hash join operator remembers all of its input in main-memory hash tables. Thus, in order for the pipelined program to work on our cluster, we must ensure that each computer has enough memory to hold a quarter of each input relation. This makes it impossible to join relations larger than the total main-memory in the cluster.

8.3.4 Unperformed experiments

Semi-join Although relevant to the comparison between DFG and MapReduce, we have been unable to perform experiments with the semi-join algorithm. The Naiad model's limitation regarding data broadcasting (cf. Section 7.2), along with other quirks in our implementation, prevent us from easily producing an efficient version of the algorithm.

Speedup, scaleup, ... Although standard metrics used in many works regarding parallel and distributed computing, we have not measured the properties of speedup or scaleup of our system. Recall that speedup indicates the measure by which the running time of a program changes with the addition of computational resources, and scaleup that by which it changes with the addition of both resources and input data in equal proportions. Perfect speedup would mean that the computation time decreases linearly, and perfect scaleup that it remains constant.

We believe these metrics to be irrelevant to this thesis. It should be obvious that both MapReduce and Naiad, as programming models, have the potential for excellent figures of speedup and scaleup. Any actual measurement would be indicative only of either framework's implementation quality, and would not fuel the comparison between the models themselves.

8.4 Summary

We have given the intuition that, although not theoretically more efficient, DFG frameworks ease the development of more efficient compilers and systems than MapReduce by allowing the natural expression of a more diverse set of algorithms, by providing more freedom on how programs are executed, and by generally being “simpler to target”. Although very limited in scope, our experiments go towards that conclusion.

As we mentioned, MapReduce is often compared to shared-nothing databases systems, with the conclusion that both do not serve the same purpose. Since both kinds of systems are theoretically rooted in dataflow programming, our belief is that a well-designed DFG framework could serve both purposes by providing a lower-level model upon which either kind of abstraction can be built.

We see DFG models as potential “assembly languages” for data-parallel programs.

Chapter 9

Conclusion

In this thesis, we have compared the increasingly popular cluster computing model MapReduce with the more general model of dataflow graphs (DFG), with respect to the compilation of higher-level languages. Using the Pig Latin language and its implementation as a base, we have studied how such a language is compiled for execution on a MapReduce platform, focusing our exposition on one common operation of the language: the relational join.

Using the relational join as an example, we have shown that the pure MapReduce model is insufficient for efficient implementation of such a language, and that practical systems such as Apache Pig use a series of tricks to step out of this model and provide efficient algorithms. We have identified these tricks, and shown that they actually contain the difference between MapReduce and DFG-based models. Targeting MapReduce while using them comes to targeting a convoluted DFG framework; so convoluted that some algorithms, such as hash join, remain unimplemented on top of MapReduce, despite being standard in other contexts.

We have given the intuition as to why a constrained target programming model like MapReduce’s complicates the implementation of compilers and optimizers, and how a DFG-based model, though arguably more difficult to program directly, was much easier to target. This intuition is reinforced by the following practical work we have accomplished during the course of this thesis.

To gain practical insight into the matter, we have developed Naiad, a distributed implementation of the dataflow graph model inspired by Microsoft’s Dryad, as well as a new back-end to the Pig Latin compiler targeting it. We have used it to perform experiments, showing that useful algorithms and optimizations that would be impractical to integrate into the original MapReduce-based back-end were simple to implement in ours. We have also shown that by changing framework parameters, the same program could be executed in ways not possible with a MapReduce framework, such as full-program pipelined execution.

We have briefly written about shared-nothing parallel databases, which are usually considered by the literature to be either competing or complementary to MapReduce-based systems. We have argued that a well-designed DFG-based framework could subsume both and be used to implement either, conciliating both approaches.

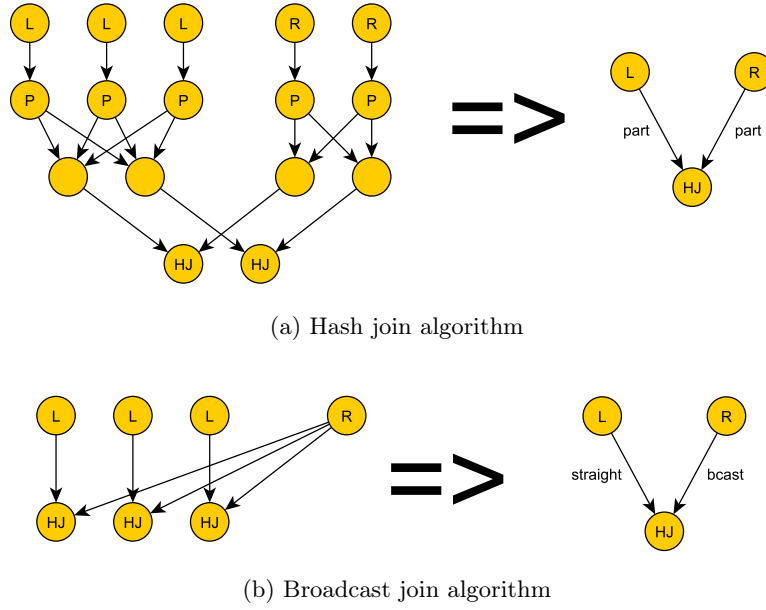


Figure 9.1: Proposed new DFG-based model: examples

Future directions

This thesis lacks experiments at scale, and the study of related concepts regarding the implementation of a DFG framework. Features like checkpointing, re-execution and speculative execution, essential for good scalability properties and simple to implement on a MapReduce platform, become more complex in a DFG system, especially in the presence of the kind of pipelining between operators that our implementation performs. Work must be done to see how these features can be efficiently integrated while retaining the flexibility of our implementation.

Vertex scheduling, an important issue for a DFG framework, is done in our system using intuition-based heuristics that lack any rigorous scientific basis. A more in-depth study of this issue would be essential for building a production-quality system. As we have outlined, this issue is important not only at the level of the framework but also at that of compilers targeting it, as efficient scheduling may depend on the semantics of the program to run.

Finally, and most importantly, our study has revealed at least one common limitation of MapReduce and our DFG-based model, namely the impossibility to naturally express broadcasts (cf. Section 7.2). To lift this limitation and provide other benefits such as even more implementation flexibility, we propose the following foundation for another kind of DFG-based model, borrowing traits from both Naiad/Dryad and MapReduce.

Rather than the explicitly data-parallel dataflow graphs that Naiad and Dryad use, we propose a model in which data parallelism is implicitly derived, for example from *labels* assigned to edges. For example, an edge could be *straight*, *partitioning*, or *broadcasting*. Figure 9.1 shows what the hash join and broadcast join algorithms that we presented in Chapter 7 might look like with this new model.

Our intuition is that such a model would both be even easier to target and allow for a better implementation, by providing the framework with more useful information about

the program. In particular, broadcasts would now be easy to identify and optimize; we also believe that other sorts of scheduling freedom and opportunities would become available. Such a model would have both the flexibility of Naiad and the simplicity and implicit parallelism of MapReduce.

We thus believe the next step in continuation of our work in this thesis to be the investigation of how this kind of programming model can be defined and implemented.

Appendix A

Listings & source code

A.1 Hash join operator

This is our Naiad operator for hash join. This version is specialized for the examples in this thesis, i.e it only computes two-way inner equi-joins. Our actual implementation follows the same lines but is generalized to all kinds of n -way joins.

We use:

- \vec{T} to denote the set of all tuples,
- K to denote the set of all join keys, and
- $\vec{T} \mapsto K$ to denote the set of functions from \vec{T} to K

```
HashJoin(key0:  $\vec{T} \mapsto K$ , key1:  $\vec{T} \mapsto K$ ) {  
  var table0 = new HashMap  
  var table1 = new HashMap  
  
  process( $i: \mathbb{N}$ ,  $\vec{t}: \vec{T}$ ) {  
     $j = 1 - i$  // the other relation  
  
    for all  $(k, \vec{u})$  in table $j$  such that  $k = \text{key}_i(\vec{t})$ :  
      broadcast( $\vec{t}$  joined with  $\vec{u}$ )  
  
    if(not EOF( $j$ )):  
      insert ( $\text{key}_i(\vec{t})$ ,  $\vec{t}$ ) into table $i$   
  }  
}
```

This is a symmetrical pipelining hash join, as documented for example in [26]: it incrementally builds hash tables for both relations and produces output tuples as soon as available. Following our API for Naiad operators, it treats one incoming tuple \vec{t} at a time, with the input number i indicating which relation it came from.

This particular implementation allows for some usage flexibility: because of the condition on the last statement, we can avoid hashing one of the relations by ensuring that the other one is completely processed first (e.g. through scheduling rules).

A.2 Naiad & Pig Latin compiler

The source code for Naiad and the Pig Latin back-end compiler are available at the following URL:

`http://alturl.com/nf4uh`

It includes a binary version of Apache Pig modified to accommodate our new back-end, and is packaged with a build system which will fetch all other necessary libraries and compilers.

Appendix B

Internship report

This appendix is a re-transcription of the report that I submitted at the end of my internship in the Research & Development division of the Euranova consulting company. This internship provided a good opportunity to obtain familiarity with the Pig and Hadoop platforms and their source code, which was necessary for the completion of thesis.

B.1 Introduction & scope

The goal of this internship was to study Pig Latin's storage abstraction layer, to evaluate the feasibility and eventual benefits of adapting it to another storage engine than the default, and finally to produce a working proof-of-concept of such an adapter for a non-relational database engine of choice.

B.1.1 Motivations

By default, Pig Latin reads data from flat files scattered across a distributed filesystem: HDFS. This filesystem is suitable for its own purpose: storing very large files, across hundreds computing nodes, in a scalable way. For Pig Latin, it means that we can run queries on files of any size, no matter how humongous, as long as these files are distributed across sufficiently many computers.

Before we can do that however, the data we're interested in has to get *into* this filesystem in the first place. Data of that scale is most commonly generated by website databases or logs, which output data at a fast pace, and need to do so without too much latency so as to quickly resume serving the users. Because of its very specific design goal of being scalable, HDFS cannot be used to directly store this data:

- write latency is much too high;
- files are not readily editable, appendable, or even seekable.

Thus, data typically has to be collected incrementally from its source, to be warehoused into HDFS; this middle step can be done manually, or with ad-hoc programs and scripts. This places a technological *gap* between the practices that generate useful data, and that of performing large-scale analysis on it. By creating a bridge between large scale systems like Hadoop and Pig, and smaller-scale databases more suitable for the data-generating processes, we can fill this gap, and leverage Pig for a larger part of the data warehousing/data mining process than before.

B.1.2 Choosing a suitable engine

We then need to choose a target storage engine for our proof-of-concept implementation. The trade-off we need to make is between a system that offers the interoperability benefits outlined in the previous section, and one that scales across a cluster so as to offer performances comparable to those of Pig on HDFS. MongoDB was chosen for a number of reasons, among which:

- it uses a data-model that matches more or less well with Pig Latin's;
- it offers data replication and *sharding*, i.e. partitioning of the data across multiple nodes;
- it can expose sharding internals to clients, which is required for our storage back-end to run Pig Latin queries in a distributed fashion.

Unfortunately, it turned out that some other implementation details of MongoDB make it difficult to use it as an efficient back-end for Pig; this will be explained in Section B.3.2.

B.2 Overview of Pig's storage abstraction layer

Foreword The produced proof-of-concept implementation is bidirectional, i.e. it can be used for both loading from and storing data to MongoDB using Pig Latin. However, the part that is the most relevant to our use case outlined in Section B.1, is the part that loads data *from* MongoDB and *into* Pig. For this reason, the other direction shall be mostly ignored by this report.

A Pig Latin script loads its data using a user-chosen *loading function*. For example, this line of code loads data using the `BinStorage` loading function built into Pig:

```
data = LOAD '/path/to/distributed/file' USING BinStorage() ;
```

Here, `BinStorage` is actually a java class that implements the `LoadFunc` interface, allowing it to be referenced in Pig queries and used by Pig to fetch data. The responsibility of such a class is twofold:

- to set things up before the query is executed; for example, to check that the input file exists, and to provide the `InputFormat` to be used for splitting the data (cf. later);
- to provide the Pig runtime with individual tuples of data.

The `InputFormat` that the loader has to provide defines how the data is to be accessed, how it is scattered across the cluster, and how it can be *split* in order to run computations on different splits in parallel. `InputFormat` is a part of Hadoop, and Hadoop offers a few standard such `InputFormats` that work with HDFS files. For example, though Pig's `BinStorage` loader uses a specialized binary encoding, and its other built-in loader `PigStorage` uses CSV-like files, both use Hadoop's `FileInputFormat` class, as both have to read from HDFS files. A `FileInputFormat` can access the HDFS filesystem and find out how the target file is divided in blocks, as well as the locations of individual blocks.

The responsibility of an `InputFormat` is thus to logically *split* the data. It must provide a list of `InputSplits`, which are then distributed by Hadoop to different computing

nodes, each of which can then execute the query on the data from its own `InputSplit`. Additionally, each `InputSplit` can tell Hadoop which nodes its data is actually stored on, allowing Hadoop to optimize the assignation of tasks according to the location of data. `InputFormat` and `InputSplit` are thus essential for controlling how the execution of queries is going to be parallelized across the cluster.

Finally, not yet cited is the `RecordReader` interface. Also provided by the `InputFormat`, a `RecordReader` is responsible for providing a record-oriented view of a particular `InputSplit`. For example, Hadoop's `FileInputFormat` provides `LineRecordReaders`; each `LineRecordReader` enables line-by-line access to the data in its corresponding `InputSplit`.

B.3 Implementation of a MongoDB to Pig adapter

We thus need to define and implement a `LoadFunc` class, an `InputFormat` class, an `InputSplit` class and a `RecordReader` class. Our `InputFormat` will split the data by exploiting MongoDB's sharding; our `InputSplits` will provide the locations of splits by accessing MongoDB's replication features; our `RecordReader` will extract individual MongoDB objects (*documents*) out of an `InputSplit`; finally, our `LoadFunc` will need to convert these documents into Pig-compatible `Tuples`.

Implementing these interfaces was mostly straight-forward. The main programming difficulty stemmed from the fact that instances of those interfaces are sent back and forth between the client running the Pig interpreter, the master node of the Hadoop cluster, and the slave nodes that run MapReduce tasks. Special care sometimes has to be taken: one needs to figure out what parts of the code are called where, and ensure that the state of objects stays consistent in between. For example, a property or variable set by the Pig interpreter will have to be somehow serialized if we need it to stay available when the object is re-created on a worker node.

Aside from these considerations, two aspects of our implementation deserve particular attention:

- how our `LoadFunc` maps MongoDB documents to Pig Latin tuples; and
- how our `InputFormat` actually splits the data.

We will start by discussing the former point, then move on to the latter.

B.3.1 Converting data

MongoDB stores *collections* of *documents*; a document is an object encoded in BSON, a binary-optimized format with a structure and textual notation similar to JSON. Documents can thus have complex, nested structures which is useful for storing real-world data.

Pig Latin treats *bags* of *tuples*, a tuple being an ordered set of *fields*, and a bag being an unordered set of tuples. The fields inside a tuple can be of any types; in particular, they can themselves be bags or tuples, also enabling the use of complex nested data structures. The basic mapping of Pig Latin types to and from MongoDB types thus seems obvious at first hand (see table B.1).

Pig Latin	MongoDB
tuple	document
field	property
bag	array

Table B.1: Mapping between MongoDB and Pig Latin basic types

Problem	Adopted solution
Some BSON data types don't have Pig Latin equivalents. Some Pig Latin data types don't have BSON equivalents.	Our adapter will fail when encountering [most of] these types.
BSON properties associate a name with a value. Pig fields are, by default, nameless, and can only be referenced by name if the interpreter knows the <i>schema</i> of the tuples that contain them.	Our loader will implicitly provide the schema of the tuples read, by inferring it from the structure of incoming BSON documents. Our storer will fail if Pig can't provide it with the schema of the tuples to write.
BSON arrays and collections are non-uniform: they may contain elements with different data types or structures. Pig bags are uniform: they may only contain tuples, all of which must follow the same structure.	Our loader assumes BSON arrays and collections to be uniform. The structure of composing elements is inferred by sampling the first element in the array or collection.

Table B.2: Data model mismatches & chosen solutions

Things are not so perfect however, as there are some subtle incompatibilities between these types. Table B.2 lists these incompatibilities along the way they are handled by our implementation. Generally, the solutions adopted have been chosen so that our storage back-end can be *symmetrical*, i.e. it has the ability to store data identical to how it was loaded. Put another way, a Pig Latin “identity” query on some given MongoDB data must either fail, or perform an exact copy of its input.

In essence, our implementation thus restricts itself to a BSON subset that is fully compatible with Pig Latin. While this restriction somewhat limits what our adapter can be used for, we believe that most or all of the common use cases are covered by this subset.

B.3.2 Splitting data

MongoDB sharding: introduction

As stated earlier, the way that data is divided into logical splits is critical for good Pig Latin query performance. First, an introduction on how MongoDB sharding works is in order.

A MongoDB collection can scatter across multiple database nodes, called *shards*. One or more central routing server(s) act as proxy(ies): incoming queries are redirected or propagated to the shard(s) containing the relevant data. Sharding is *key-based*: when first creating the collection, the user will have to choose the *sharding key*, which is simply the name a property that all documents in the collection must contain. Documents are grouped into *chunks*, which are ranges of values for that property; MongoDB then assigns chunks to shards.

As an example, let's say we want to insert a new document into a sharded collection. Here is what happens then:

1. The client sends the document to save to a MongoDB routing server.

2. The routing server knows the sharding key for that collection; it looks up the corresponding value in the document to insert.
3. The routing server finds out which range that value is in, and thus what chunk the documents belongs to.
4. Finally, the routing server finds out which shard holds that chunk, and forwards it the insertion request.

The useful thing is that MongoDB can provide its clients with detailed information about shards and chunks, which allows us to skip through routing servers and access shards directly, and thus to control query distribution ourselves.

Implementation

The main idea behind our implementation is to follow the sharding information provided by a MongoDB routing server, and thus to create one `InputSplit` for each MongoDB chunk (see above). Given a chunk, MongoDB can provide us with the list of all hosts that have a copy of it. Then, we can access one of these hosts directly to fetch the data without going through the routing servers anymore. This is especially useful if these hosts are also configured as Hadoop workers, in which case Pig Latin jobs can be scheduled to run locally on hosts that have the data, yielding important performance benefits.

Unfortunately, with large collections, this method is crippled by implementation details of MongoDB. The problem stems from the following facts:

- Data in MongoDB is stored in no particular order
- Multiple chunks in the same shard are not kept separate
- MongoDB collections are entirely memory-mapped

Together, the first two points implicate that reading a particular chunk from a shard will require the use of an *index*, and that random accesses will have to be made all throughout the shard in order to fetch the whole chunk. Adding the third point means that, if the shard's data is sufficiently larger than main memory, fetching a chunk will involve a large number of page faults, and cause thrashing. MongoDB's `mongostat` utility helps verify that this is what happens.

In order to still be able to perform useful experiments with large collections on our current configuration, we implemented additional ways of splitting data that do not suffer from this problem. For example, we have an implementation of `InputFormat` that creates one `InputSplit` for each **shard** rather than chunk. Each shard is then read in "natural order", i.e. in storage order. This enable the host to read the data sequentially, which does solve the problem; it can not be considered a perfect solution however, as one `InputSplit` per shard is not enough for good scalability and failure tolerance properties.

B.4 Additional optimization possibilities

A few optimizations can be added to the basic MongoDB adapter implementation presented above. Pig operators can be pushed down to MongoDB; natural data grouping

present in collections can be exploited; MongoDB’s indexes can be used to provide data in sorted order, making possible the use of optimized algorithms for certain operations.

B.4.1 Pushing down operators

In addition to the `LoadFunc` interface, a Pig loader can implement the `LoadPushDown` interface. Then, Pig can interrogate our loader to figure out what types of operations can be pushed down. Finally, if the situation calls for it, Pig will provide the loader with descriptions of the operations to be pushed down. So far, Pig only allows to push down *projection* operations, i.e. when we’re only interested in a subset of the columns in the input data.

When querying MongoDB, one can ask to return only a defined subset of the fields from the documents that satisfy the query. We used that feature to implement Pig’s projection pushdown as described above. We tested our implementation by running a query that runs through a 500-columns-wide table, from which only two columns are actually used. When using projection pushdown, we measured an improvement of over 300% in running time.

B.4.2 Collected grouping

A Pig loader may also implement the `CollectableLoadFunc` interface. This tells Pig that data will be split such that all instances of any particular value of a given field will appear in the same `InputSplit`. This allows Pig users to ask for a “collected” grouping: an alternate algorithm for “group by” operations that takes place entirely in the Map phase of a MapReduce jobs.

This is trivial to add to our implementation **if** the field that has to be collected happens to be the sharding key for the MongoDB collection. Unfortunately, Pig currently offers no obvious way to verify that, or even to specify what field should actually be collected.

Performance gains were not measured, as the performance problems cited in Section B.3.2 prevent us from easily designing a suitable test.

B.4.3 Exploiting indexes

Finally, a Pig loader can implement the `IndexableLoadFunc` and `OrderedLoadFunc` interfaces. The former tells Pig that data will be read in sorted order, and allows Pig to seek to an arbitrary position among that order. The latter enables Pig to compare between `InputSplits`: indeed, if data was sorted before being split, then splits have a position relative to each other.

When performing a “join” or “cogroup” operation, if the data comes from loaders which implement these interfaces, then the user can opt for a “merge” join or cogroup, that uses the fact that data is ordered to perform the joining or grouping efficiently in the Map phase of a MapReduce job.

Once again, these interfaces are easily implemented **if** the sorting key corresponds to the MongoDB collection’s sharding key. And once again, Pig provides no obvious way for us to know which key it wants data to be sorted on. Our implementation thus assumes that the user will only perform merge joins or cogroups on sharding keys, and will not work correctly otherwise.

Again, our issue with MongoDB’s performance with large collections prevents us from designing a suitable benchmarking test.

B.4.4 Associated Pig design issues

The last three interfaces that we introduced, `CollectableLoadFunc`, `IndexableLoadFunc` and `OrderedLoadFunc` all have design issues that prevent them from being more useful. First of all, as mentioned several times above, Pig doesn’t tell us what field need to be collected or sorted. In existing implementation of those interfaces, this seems to be determined by tight interaction between the loader and the operator that implements the actual grouping or joining.

More generally, these operators interact with the loaders directly and in a non-standard way. For example the `IndexableLoadFunc` as used by the merge join operator follows a completely different code path than a “normal” loader; different methods get called in a different order. Reading and understanding the merge join code is thus actually necessary to implement our loader, which is not how things should be.

Ideally, Pig should ask the loader what fields of the input it can collect or sort. Then, it should *itself* decide which algorithms to use according to the loader’s capabilities. Most of all, these optimized operators and their specialized loaders should be designed so that they fit into the system, rather than as a hack on top of it.

B.5 Conclusion

During this internship, we studied Pig’s storage abstraction layer, discussed the possibility of implementing it over other types of storage engines, provided such an implementation over MongoDB, and documented the associated quirks.

We have seen that Pig provides relatively good support for easily plugging one’s own storage back-end. We have outlined why not any database engine is suitable as a storage back-end for Pig Latin. Finally, we’ve explained how additional optimizations can be made available to Pig when using advanced storage back-ends that can guarantee assumptions about the data, such as sorting order; however, we’ve also shown how Pig’s abstraction layer over these advanced features is still immature and awkward.

Bibliography

- [1] OpenSSI. <http://openssi.org/>. Retrieved on August the 13th, 2010.
- [2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
- [3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovic, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 international conference on Management of data, SIGMOD '10*, pages 975–986, New York, NY, USA, 2010. ACM.
- [4] D. Borthakur. The Hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.
- [5] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.
- [6] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, F. Reiss, and M.A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.
- [7] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21. USENIX Association, 2010.
- [8] Intel Corporation. Intel compilers. <http://software.intel.com/en-us/intel-compilers/>. Retrieved on August the 13th, 2010.
- [9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [10] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 1, 2008.
- [11] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35:85–98, June 1992.
- [12] Apache Foundation. The Hadoop project. <http://hadoop.apache.org/>. Retrieved on August the 13th, 2010.

- [13] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.
- [14] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.
- [15] The Portland Group. Portland compilers. <http://www.pggroup.com/>. Retrieved on August the 13th, 2010.
- [16] Typesafe Inc. Akka project. <http://akka.io/>. Retrieved on July the 22th, 2011.
- [17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):72, 2007.
- [18] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.
- [19] S. Loebman, D. Nunley, Y.C. Kwon, B. Howe, M. Balazinska, and J.P. Gardner. Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help? In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.
- [20] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 1099–1110. ACM, 2008.
- [21] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13:277–298, October 2005.
- [22] Yahoo Research. Yahoo! open source student projects. <http://research.yahoo.com/node/1980>. Retrieved on August the 13th, 2010.
- [23] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. mapreduce and parallel dbms: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.
- [24] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.
- [25] Michael Stonebraker University and Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.
- [26] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1:103–128, 1993. 10.1007/BF01277522.
- [27] H. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

- [28] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P.K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *Proc. LSDS-IR*, page 8, 2009.
- [29] École Polytechnique Fédérale de Lausanne. The Scala Programming Language. <http://www.scala-lang.org/>. Retrieved on August the 6th, 2011.