Integrating Pig and Stratosphere

VASILIKI KALAVRI

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:146

KTH Royal Institute of Technology

Dept. of Software and Computer Systems

Degree project in Distributed Systems

Integrating Pig and Stratosphere

Author: Vasiliki Kalavri
Supervisors: Per Brand, SICS
Examiner: Associate Prof. Vladimir Vlassov, KTH, Sweden

Abstract

MapReduce is a wide-spread programming model for processing big amounts of data in parallel. PACT is a generalization of MapReduce, based on the concept of Parallelization Contracts (PACTs). Writing efficient applications in MapReduce or PACT requires strong programming skills and in-depth understanding of the systems' architectures. Several high-level languages have been developed, in order to make the power of these systems accessible to non-experts, save development time and make application code easier to understand and maintain. One of the most popular high-level dataflow systems is Apache Pig. Pig overcomes Hadoop's one-input and two-stage dataflow limitations, allowing the developer to write SQL-like scripts. However, Hadoop's limitations are still present in the backend system and add a notable overhead to the execution time. Pig is currently implemented on top of Hadoop, however it has been designed to be modular and independent of the execution engine.
In this thesis project, we propose the integration of Pig with another framework for parallel data processing, Stratosphere. We show that Stratosphere has desirable properties that significantly improve Pig's performance. We present an algorithm that translates Pig Latin scripts into PACT programs that can be executed on the Nephele execution engine. We also present a prototype system that we have developed and we provide measurements on a set of basic Pig scripts and their native MapReduce and PACT implementations. We show that the Pig-Stratosphere integration is very promising and can lead to Pig scripts executing even more efficiently than native MapReduce applications.

Referat

Att skapa effektiva applikationer i MapReduce eller PACT kräver goda programmeringskunskaper och djup förståelse utav systemens arkitektur. Flera högnivå-språk har utvecklats för att göra de kraftfulla systemen tillgängliga för icke-experter, för att spara utvecklingstid och för att göra applikationernas kod lättare att förstå. Ett utav de mest populära systemen för högnivå-dataflöden är Apache Pig. Pig överkommer Hadoops ett-input och tvånivå-begränsningar och låter utvecklaren skriva SQL-liknande skript. Dock är Hadoops begränsningar fortfarande närvarande i backend-systemet och lägger till ett synligt tillägg till exekutionstiden. Pig är för nuvarande implenterat ovanpåHadoop, dock har det designats för att vara modulärt och oberoende utav exekutionsmotorn.
I det här exjobbs-projektet presenterar vi integration utav Pig med ett annat framework för parallel dataprocessering, Stratosphere. Vi visar att Stratosphere har önskade egenskaper som signifikant förbättrar Pigs prestanda. Vi presenterar en algoritm som översätter Pig Latin-skript till PACT-program som can köras påNephele-exekutionsmotorn. Vi presenterar ocksåett prototypsystem som vi har utvecklat och vi bidrar med mätningar utav ett set av grundläggande Pigskript och deras MapReduce och Pact-implementationer. Vi visar att Pig-Stratosphere-integrationen är väldigt lovande och kan leda till att Pigskript exekuteras mer effektivt än MapReduce applikationer.

Acknowledgment

I deeply thankful to my supervisor, Dr. Per Brand, who guided me and encouraged me through this work. I would also like to express my sincere gratitude to his PhD students at the Swedish Institute of Computer Science, Martin Neumann and Alex Averbuch, for their insights, ideas and valuable feedback throughout this work. Also, my examiner Vladimir Vlassov, my MSc coordinator, Johan Montelius and my opponent, Nick Rutherford, for their reviews and comments that helped me improve greatly the final version of this document.
I am also truly indebted to the Stratosphere development team and the DIMA research group of TU Berlin. They offered me great support by always answering all my questions very quickly and clearly through the stratosphere-users mailing list. I also was highly benefited and honored by their invitation to present my work at TU Berlin. My presentation and interaction with their team was a great source of motivation and inspiration.
I owe a very special thanks to my dear EMDC friends and classmates, for their suggestion of tools, technical advice and support, but mainly for their vivid spirit and cheerfulness. They offered me invaluable moments of recreation that boosted my self-confidence and helped me face with courage even the hardest problems.
Last, but certainly least, I would like to thank my family for their immeasurable support, their sincere faith in me and their patience, without which I would never have managed to complete the present work.

Stockholm, 25 June 2012

Vasiliki Kalavri

Contents

List of Figures

Chapter 1

Introduction

Large-scale data management and analysis is currently one of the biggest challenges in the area of Distributed Systems. Industry, as well as Academia, are in urgent need of data analytics systems, capable of scaling to Terabytes or even Petabytes of data. Such systems need to efficiently analyze web data, search indices, log files and scientific applications data, such as climate indicators or DNA sequences. Most of the recent approaches use massive parallelism and are deployed on large clusters of hundreds or even thousands of commodity hardware.
The most popular framework MapReduce (20), was first proposed by Google, and its open-source implementation, Hadoop (3), is nowadays widely used. However, it has several limitations, such as accepting only one input data set in the Map or Reduce functions. Another disadvantage lies in the fact that any MapReduce program needs to follow a static pipeline of the form split-map-shuffle-sort-reduce. This form is suitable for simple applications, such as log-file analysis, but complicates the implementation of relational queries or graph algorithms. These observations have led researchers to develop more general-purpose systems, inspired by the idea of MapReduce (27; 12). One of them is Stratosphere (10), which consists of an open-source programming model, PACT, and the Nephele execution engine (32). The system is essentially a generalization of MapReduce and aims to overcome the weaknesses mentioned above.

1.1 Motivation

Using both MapReduce and PACT requires significant programming ability and in-depth understanding of the systems' architectures. Applications usually lead to complex branching dataflows which are often too low-level and inflexible. In order to save development time, avoid bugs and make application code easier to understand and maintain, several high-level languages have been proposed for these systems. Hadoop currently supports JAQL (15), Hive (31) and Pig (22).
One of the most popular high-level dataflow systems is Pig. Pig was developed by Yahoo! engineers and is currently an open-source project under the Apache Software

Foundation. Pig Latin (28), Pig's high-level language, aims to offer the simplicity and declarative fashion of SQL, while maintaining the functionality and expressiveness of MapReduce. Pig compiles Pig Latin into physical plans that are transformed into MapReduce jobs and run on top of Hadoop. Pig overcomes Hadoop's one-input and two-stage dataflow limitations. It also provides built-in functions for common operations, such as filtering, join and projection of data. Having Hadoop as its back-end execution engine, Pig directly benefits from its scalability and fault-tolerance. However, even if not obvious to the users, the limitations and inflexibility of Hadoop are still present in the Pig system. Translating relational operations using the static pipeline of MapReduce causes a notable overhead when compiling Pig Latin, since data has to be materialized and replicated after every MapReduce step.
Pig has been designed to be independent of the execution engine. Its initial implementation is using Hadoop, carrying along all advantages and disadvantages of the framework. Stratosphere offers a superset of MapReduce functionality, while overcoming some of its major weaknesses. It allows data pipelining between execution stages, enabling the construction of flexible execution strategies and removing the demand for materialization and replication in every stage. Moreover, when dealing with multiple data sets, records need not be tagged with an additional field that indicates their origin, as in MapReduce. This limitation is not present in the Stratosphere context, since PACT supports multiple inputs.

1.2 Contributions

In this thesis, we present our work on integrating the Pig System with Stratosphere. We studied the Pig system in depth and analyzed its internal structure in order to design a feasible integration strategy. We identified the separation line of the high-level abstraction offered by Pig and the Hadoop execution engine and developed a prototype implementation that supports a limited set of Pig operations.
We show that it is possible to plug a different execution engine into the Pig system. However, a large part of the Pig stack has to be re-implemented in order to support the new engine. With our Pig to PACT translation algorithm and our prototype, we show that Stratosphere has desirable properties that significantly simplify the plan generation. We developed a set of basic scripts and their native MapReduce and PACT equivalents and we provide a comparison of our prototype with Pig, as well as the native programs. We observed that Stratosphere's relational operators are much more efficient than their MapReduce equivalents. As a result, Pig on Stratosphere seems to have a great advantage over Pig on Hadoop and often results in Pig executing even faster than native MapReduce. Our prototype results are very promising and we strongly believe that exploiting additional Stratosphere features, such as Output Contracts or compiler hints would result in an even more efficient system.

1.3 Structure of the thesis

The rest of this thesis is structured as following: In Chapter 2, we provide the necessary background and explain the concepts that are needed in order to follow this work. We briefly present the MapReduce programming model and the Hadoop Distributed File System. We also discuss query optimization techniques, most of which are present in the Pig system. In Chapters 3 and 4 we present the Pig and the Stratosphere systems respectively. Chapter 5 describes our work, the integration strategies, the decisions we took and how we designed the system. We present here out translation algorithm, as well as the implementation in detail. Chapter 6 contains the evaluation of our prototype, describes the experimental setup and presents and discusses our results. In Chapter 7, we shortly comment on related work and compare it with our system, while we provide our conclusions, open issues and vision for the future in the last chapter.

Chapter 2

2 Background

2.1 The MapReduce Programming Model

MapReduce is a programming model based on the idea of data parallelism. It is designed to efficiently execute programs on large clusters of commodity machines, on top of which a distributed file system is deployed. The file system used in the open-source implementation of MapReduce, Hadoop, is the Hadoop Distributed File System, HDFS (30), which is briefly introduced in the next section. MapReduce aims at exploiting data locality and therefore tries to move the computation where the data reside, instead of moving the data close to the code. Its architecture is inspired by functional programming and consists of two second-order functions, Map and Reduce, which form a static pipeline, as shown in Figure 2.1.
Data are read from the underlying distributed file system and are transformed into key-value pairs. These pairs are then grouped into subsets and serve as input parameters for parallel instances of the Map function. A user-defined function must also be specified as a parameter and is applied to all subsets independently. The output of the Map function is another set of key-value pairs. These new set is then sorted by key and partitioned according to a partitioning function. This sorted data feed the next stage of the pipeline, the Reduce function. The partitioning stage of the framework guarantees that all pairs sharing the same key will be processed by the same Reduce task. In a similar way, a user-defined function is again applied to all pairs in parallel, in order to produce the output. Each parallel instance of the

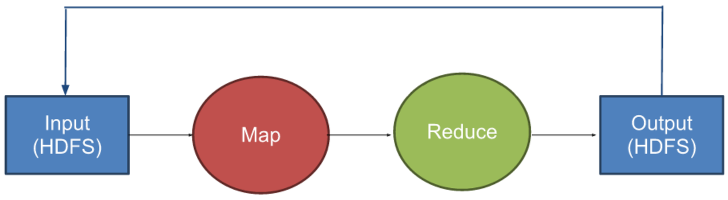


Figure 2.1: MapReduce Pipeline

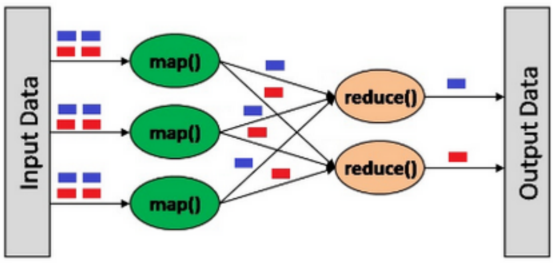


Figure 2.2: Stages of a MapReduce job

Reduce function creates a file in the distributed file system with its results. Figure 2.2 illustrates the stages of a MapReduce job.
One of the most important advantages of the schema described above is that most of the parallelization complexity is handled by the framework. The user only has to write the first-order function that will be wrapped by the Map and Reduce functions. However, this advantage comes with loss of flexibility. Each job must consist of exactly one Map function followed by one Reduce function, and no step can be omitted or executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be passed from one job to the next through the file system. This limitation can frequently add a significant overhead to the execution time.

2.2 HDFS

HDFS is the distributed file system used by the Hadoop project. Hadoop MapReduce jobs read their input data from HDFS and also write their output to it. HDFS is also used by the Pig and Stratosphere systems. HDFS has been very popular because of its scalability, reliability and capability of storing very large files.
HDFS has two main types of nodes: the datanode and the namenode. Typically, a Hadoop instance has a single datanode and a group of datanodes are organized together into a Hadoop cluster. The main responsibility of a datanode is to store blocks of data and to serve them under request over the network. Datanodes can communicate with each other in order to rebalance data load and also to achieve replication. The default replication strategy offered by HDFS is storing a file on three nodes. The other type of node, the namenode, is unique in a HDFS cluster and is responsible for storing and managing metadata. HDFS architecture is shown in Figure 2.3. The namenode stores metadata in memory, which limits the number of files that can be stored by the system. If there is a need to store a large number of small files, the namenode is the bottleneck of the system.
The main advantage of using HDFS when running MapReduce jobs is that it is aware of the data locations. This means that MapReduce tasks will be scheduled to

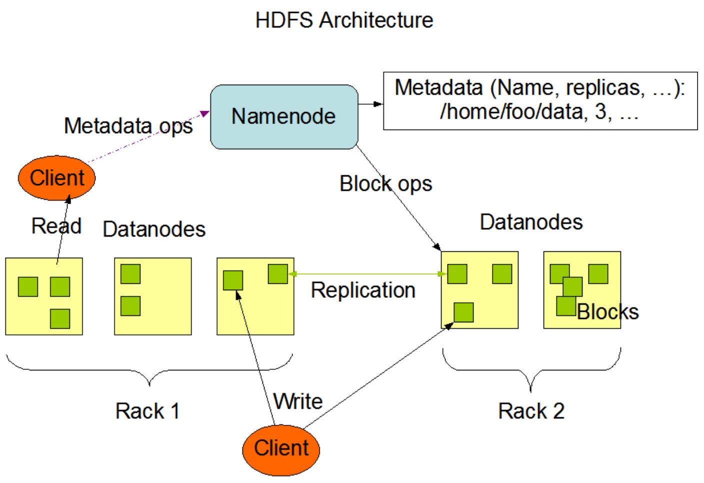


Figure 2.3: HDFS architecture

run on those nodes that would minimize network traffic and avoid unecessary data transfers.

2.3 Query Optimization

Traditionally, relational database systems consist of two main sub-systems: the query optimizer and the query execution engine. The query execution engine implements a set of physical operators, i.e. different execution strategies for relational operations. It is a common case that physical operators do not map to relational operators in a one-to-one fashion. Physical operators are often represented as trees or graphs, which are called execution plans. In these plans, edges represent the flow of data from one operator to the next, so they are also called data-flow graphs. The execution engine is responsible for executing the plan and producing the result of the given query. The query optimizer generates the input for the execution engine. It accepts the parsed high-level language query and is responsible for producing an efficient execution plan to be executed. Essentially, the optimizer's job is equivalent to a complex search problem, having as parameters a search space of all possible plans, a cost estimation function and an algorithm to conduct the space. The goal of the optimizer is to minimize the cost of the produced plan using an efficient search algorithm.
One of the most influential systems in the area of query optimization has been the System-R optimizer (13). Its algorithm uses dynamic programming and the concept of interesting orders. The latter provide information on how different possible orderings of operations could impact the cost of subsequent operations. Its cost estimation framework collects statistical information from data that has been stored in the past. Then, given an operator and the statistical information for each of its inputs, it provides a statistical summary of the operator's output, as well as the

estimated cost for executing this operator. Other popular systems are Exodus (26) and its successor, Volcano (24), which provide extensible tools and a more powerful search engine. They combine dynamic programming with goal-directed search and branch-and-bound pruning techniques. Both Exodus and Volcano, as well as Cascades (25) belong to the family of rule-based optimizers. These optimizers use a dynamic set of rules that consist of a pattern and a substitute. If a pattern is found in the expression tree, the substitute expression will be added in the search space. Rules and patterns can be very complex and might differ depending on the optimization goals of each system. A detailed overview of query optimization techniques can be found in (18).

2.4 High-Level Languages of Data-Analytics Frameworks

SQL has long been the standard of high-level languages for relational queries. Most of the high-level languages developed for large-scale data analytics frameworks have borrowed a lot of ideas and concepts from it. Google actually built their own SQL implementation on top of the MapReduce framework (17). Other systems offer high-level languages that share ideas from declarative programming languages or scripting languages, or provide Java/Scala interfaces for easy application developing. In this section, we briefly present the most influential systems and their associated high-level languages.

One of the most widely-used systems is Hive (31). Initially developed by Facebook (2), Hive is a data warehousing solution developed atop Hadoop. It provides an easy way to store, summarize and query large amount of data. Hive's high-level language, HiveQL, allows users to express queries in a declarative, SQL-like manner. Very similar to Pig, HiveQL scripts are then compiled to MapReduce jobs and executed on the Hadoop execution engine.

Cascading (1) is a Java application framework that facilitates the development of data processing applications on Hadoop. It offers a rich Java API for defining and testing complex dataflows. It abstracts the concepts of map and reduce and introduces the concept of flows, where a flow consists of a data source, reusable pipes that perform operations on the data and data sinks. Cascading quickly gained popularity among the industry and Twitter (11) even developed and open-sourced a Scala API for it, Scalding (9).

Similar to Cascading, Google has released FlumeJava (16), a Java library for easily programming and managing MapReduce pipelines. FlumeJava is a powerful system that supports a large number of operations and uses deferred evaluation instead of constructing a dataflow graph. Its main advantage, though, is the system's optimizer, which uses a set of independent graph transformations in order to produce an efficient execution plan.

Another popular query language is Jaql (15). Jaql is less general than the systems we have already introduced in this section, as it is designed for quering semi-structured data in JSON format only. the system is extensible and supports parallelism using Hadoop's Map-Reduce. Although Jaql has been specifically designed for data in JSON format, it borrows a lot of characteristics from SQL, XQuery, LISP, and PigLatin. Recently, Jaql was also integrated with Nephele/PACTs at TU Berlin (19). Last but not least, it is worthwhile to introduce DryadLINQ (33), the programming environment built on top of Microsoft's Dryad (27) distributed execution engine. DryadLINQ uses the .NET Language Integrated Query (5) to provide a rich high-level query language, relying on the .NET library. It transforms LINQ programs into distributed Dryad jobs graphs which are both statically and dynamically optimized.

3 Chapter 3 Pig

Pig is a high-level system that consists of a declarative scripting language, *Pig Latin*, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. Pig offers an abstraction that aims to hide the complexity of the MapReduce programming model and allow users to write SQL-like scripts, providing all common data operations (filtering, join, ordering, etc.) Pig user need not understand the implementation details of MapReduce, nor do they need to care about the data distribution and parallelization techniques. Developing applications using Pig has proven to be much more efficient regarding development time that using MapReduce. Maintenance costs and bugs also seem to be greatly reduced, as much less code is required.
Soon after Pig was released, its creators published a statement that explains the inspiration behind the project and the intentions of the team for the future. This statement is knows as the "Pig Philosophy" and can be summarized in the following four points:

- **Pigs eat anything:** Pig can operate both on structured and unstructured data, relational, nested, or even metadata. It can receive input from files, databases or key-value stores. This flexibility comes from the fact that apart from the built- in functions for loading data, users can write their own and customize them according to their needs.

- **Pigs live anywhere:** Pig is a system for parallel data processing. Although its first implementation uses the Hadoop execution engine, it is intended to be independent of the underlying framework and easily ported on top of other systems. The present thesis is based on this design characteristic.

- **Pigs are domestic animals:** Although Pig is a high-level system that aims to abstract information of the underlying implementation, its users are provided with a great amount of freedom in customizing and extending the system. Users can provide their own load and store functions, they can turn off optimization rules and can also integrate Java or scripting language code with Pig.

- **Pigs fly:** The idea behind this statement is that Pig's primary goal is performance and all future should be performed towards this direction.

3.1 System Overview

The Pig System takes a Pig Latin program as input and produces a series of MapReduce jobs which are then executed on the Hadoop execution engine. The series of transformation steps that Pig programs go through is shown in Figure 3.1. The Pig Latin script is first sent to the Parser. The Parser is responsible to check for syntax and type correctness, schema inference, existence of streaming executables referenced, definition of variables used and the ability to instantiate classes corresponding to UDFs (User-Defined Functions). The output of the Parser is a DAG, the Cannonical Logical Plan. Each Pig Latin statement corresponds to a logical operator which is added as a new node to the graph. The edges of the graph define the dataflow between operators. The initial Logical Plan is then processed by the Logical Optimizer, which performs logical optimizations, such as projection pushdown. The logical optimizations performed are explained in detail in a following section of this chapter. Next, the Logical Plan is transformed into a Physical Plan which is then compiled to to a set of MapReduce jobs. Next, the Map-Reduce Optimizer performs optimizations, such as adding a combiner stage for early partial aggregation, when possible. Finally, the MapReduce jobs DAG is topologically sorted and jobs are submitted to Hadoop.
Pig offers three execution modes:

- **Interactive Mode:** In this mode, the user is provided with an interactive shell, *Grunt*, to which it can submit Pig Latin commands. No compilation or plan execution is happening in this stage, unless the STORE command is used. The user is provided with two very useful commands in this mode: DESCRIBE and ILLUSTRATE. The DESCRIBE command displays the schema of a variable and ILLUSTRATE displays a small set of example data for a variable, helping the user in understanding the program semantics. These two commands are extremely useful for debugging programs containing complex nested data.

- **Batch Mode:** Instead of using the interactive shell and writing the commands one by one, a Pig Latin script is more often written inside a file that contains all the necessary commands. This file can then be submitted for execution in the *Batch* mode.

- **Embedded Mode:** Pig commands can also be submitted through Java methods. A Java library is provided which allows the integration of Pig scripts inside Java code. In this mode, Pig Latin programs can be generated dynamically or dynamic control flows can be constructed.

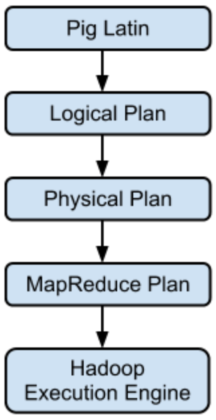


Figure 3.1: Pig System Overview

3.2 Pig Latin

Pig Latin is the high-level dataflow language of the Pig system. Pig Latin scripts consist of statements which can contain variables of different types, as well as built-in or user-defined functions.

3.2.1 Statements

Pig Latin statements cover most of the usual data analysis operations, such as:

- **Loading and Storing:** LOAD, STORE, DUMP
  These commands allow Pig to read and write data from the distributed file system or other sources and can be combined with user-defined functions that specify data characteristics, such as schema information.
- **Filtering:** FILTER, DISTINCT, FOREACH...GENERATE, STREAM
  These statements receive a single dataset as input and can operate on it in records, in order to perform transformations and produce a new dataset as output.
- **Grouping and Joining:** GROUP, COGROUP, JOIN, CROSS
  These are commands that operate on one or multiple datasets. They can be used to group records based on one or more fields, join records on user-specified keys or produce the cartesian product of all records.
- **Sorting:** ORDER, LIMIT
  Pig Latin also includes commands for ordering the input one or more fields or limiting the number of output records.

- **Combining and Splitting:** UNION, SPLIT
  Datasets can also be combined in one larger dataset, using UNION, or split in smaller sets based on some attribute, using SPLIT.

3.2.2 Data Types

Pig Latin offers a wide type system, covering the most common numeric, text, binary and complex types. All scalar Pig types are represented in interfaces by java.lang classes, which makes them easy to work with in user defined functions.

- **Numeric:** Int, Long, Float, Double
  The size and precision of these types are identical to the ones of the java.lang package.
- **Text:** Chararray
  A string or array of characters. Constant chararrays can be are expressed as string literals with single quotes.
- **Binary:** Bytearray
  An array on bytes.
- **Complex:**

  Tuple: A tuple is an ordered sequence of fields of any type. Each field contains one data element and can be referred to by name or position. Elements do not all need to be of the same type and schema information is optional. A tuple constant can be written enclosing the fields inside parentheses and using commas to separate the fields. For example, ('foo', 42) describes a tuple constant with two fields.

  Bag: A bag is an unordered set of tuples. Tuples inside bags cannot be referenced by position and a schema information is again optional. A constant bag can be constructed using curly brackets to enclose the tuples and tuples are separated by commas. For example, ('foo', 42), ('bar', 88) describes a bag with two tuples, each of which has two fields. Bags are often used in Pig in order to create type sets, by using one-field tuples. Bag is the only type in Pig that is not required to fit into memory. As bags can be quite large, Pig has a mechanism to detect when bags do not fit in memory and can spill parts of them to disk.

  Map: A map in Pig is a set of key-value pairs, where the value can be of any type, including complex types. The key is of type chararray and is used to find the value in the map. Map constants can be constructed using brackets where keys and values are separated by a hash, and key value pairs are separated by commas. For example, ['name'#'bob', 'age'#55] will create a map with two keys, “name“ and “age“. Note that it is allowed to have values of different types, chararray and int in this case.

3.2.3 Functions

The language has a rich set of built-in functions, but also allows users to define and use their own functions. The built-in functions can be divided in the following categories:

- **Eval**: AVG, CONCAT, COUNT, isEmpty, MAX, MIN, etc.
- **Load/Store**: TextLoader, PigStorage
- **Math**: ABS, LOG, ROUND, SQRT, etc.
- **String**: INDEXOF, LOWER, REPLACE, SUBSTRING, UPPER, etc.
- **Bag and Tuple**: TOTUPLE, TOBAG

A complete guide on built-in Pig functions can be found in (6).

3.2.4 An Example: WordCount

We present here the most popular and simple example of data analysis applications, the WordCount. In this example, a text file is given as input and the number of times each word appears in the file is expected as output. Let’s assume that we have a text file with the following content [1]:

I got an aching head
I have been sleeping too long
In this broken bed
What can I do to excite you
What can I do to lie still

This application can be easily written in Pig Latin in the following way:

```
wordInput = LOAD 'input' USING TextLoader();
words = FOREACH wordInput GENERATE FLATTEN((TOKENIZE($0)))
AS word;
grouped = GROUP words BY word;
result = FOREACH grouped GENERATE group AS key, COUNT(words)
AS count;
STORE result into 'output';
```

[1]Lyrics from the Song "Mercy on the Street" by Midnight Choir

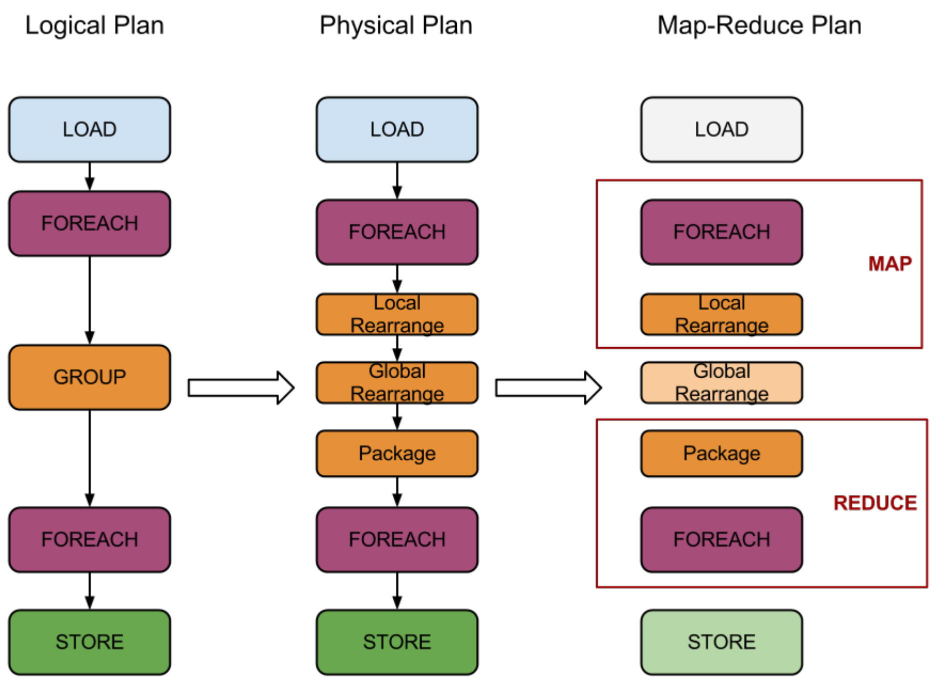


Figure 3.2: Pig Plan Compilation for the WordCount example

In the above script, the contents of the file are loaded in a set called wordInput and are then split into words and saved in the words set. Next, the words set is grouped using each word as the key and saved in the set grouped. For each element of the grouped set, the occurrences of each word are counted and the key-value pairs result are generated. The generated plans are shown in Figure 3.2. In this example, FOREACH and Local Rearrange can be performed inside the Mapper, while the Package and the next FOREACH can be performed inside the Reducer. The Global Rearrange and LOAD/STORE operations are taken care by the Hadoop framework automatically. After the execution, an output file will be created with the following content:

(I,4), (In,1), (an,1), (do,2), (to,2), (bed,1), (can,2), (got,1), (lie,1), (too,1), (you,1), (What,2), (been,1), (have,1), (head,1), (long,1), (this,1), (still,1), (aching,1), (broken,1), (excite,1), (sleeping,1)

3.2.5 Comparison with SQL

Pig Latin has a lot of similarities with SQL and people used to write SQL queries often find it easy to get started with it. However, there are a few very fundamental differences that might create confusion in the beginning. while SQL is a query language and describes a question to be answered, Pig Latin is used to describe how to answer it, by defining exactly how to process the input data. Another important difference is that an SQL query is written in order to answer one single question. If users need to do several data operations, they must write separate queries, often storing the intermediate data into temporary tables. Another solution is using subqueries, which might be tricky to write correctly. Pig, on the other hand, is designed to support several data operations in the same script. As a result, there is no need to write the data pipeline in the form of subqueries or to store data in temporary files. Moreover, while SQL is designed to operate on data that fulfill certain constraints and have well-defined schemas, Pig is able to operate on data with unknown schemas or even inconsistent. Data need not be organized in tables and can be processed as soon as they are loaded in HDFS.

3.3 Implementation

In this section, we will discuss in more detail the internals of the Pig system. We first explain how plans are generated and how MapReduce jobs are created. Then, we briefly describe how the Hadoop Launcher is implemented. Next, we describe Pig's Logical Optimizer and give a few examples of common optimizations. We then explain a set of common operators and discuss in detail their compilation to MapReduce. Finally, we provide a brief comparison between Pig and MapReduce.

3.3.1 Plan Generation

The Pig execution engine is divided into a front end and a back end. The front end takes care of all compilation and transformation from one Plan to another. First, the parser transforms a Pig Latin script into a Logical Plan. Semantic checks (such as type checking) and some optimizations (such as determining which fields in the data need to be read to satisfy the script) are done on this Logical Plan. We discuss these optimizations in the following section. The Logical Plan is then transformed into a PhysicalPlan. This Physical Plan contains the operators that will be applied to the data. Each Logical operator is compiled down to one or more Physical Operators. The PhysicalPlan is then passed to the MRCompiler. This is the compiler that transforms the PhysicalPlan into a DAG of MapReduce operators. It uses a predecessor depth-first traversal of the PhysicalPlan to generate the compiled graph of operators. When compiling an operator, the goal is first trying to merge it in the existing MapReduce operators, in order to keep the generated number of jobs as small as possible. Physical operators are divided into blocking and non-blocking. Blocking are these operators that require a shuffling phase and will therefore force the creation of a reduce phase. A new MapReduce operator is introduced only

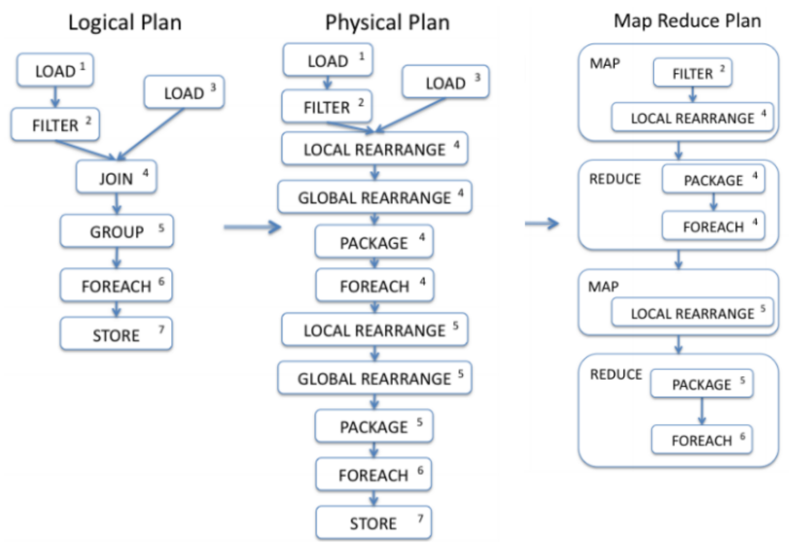


Figure 3.3: Logical to Physical to MapReduce Plan Generation

for blocking operators and splits. The two operators are then connected using a store-load combination. The output of the MRCompiler is an MapReduce Plan. This plan is then optimized by using the Combiner where possible or by combining jobs that scan the same input data. The final set of of MapReduce jobs is generated by the JobControlCompiler. It takes an MapReduce Plan and converts it into a JobControl object with the relevant dependency info maintained. The JobControl Object is made up of Jobs each of which has a JobConf. The conversion is done by compiling all jobs that have no dependencies and removing them from the plan. The generated jobs are then submitted to Hadoop and monitored by Pig. In the back end, Pig provide generic Map, Combine and Reduce classes which use the pipeline of physical operators constructed in the front end to load, process, and store the data. The plan generation for the following script can be seen in Figure 3.3.

```
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A BY y>0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE group, COUNT(D);
STORE F INTO 'output';
```

3.3.1.1 Pig's Hadoop Launcher

In this section we shortly present functionality of the main class that launches Pig for Hadoop MapReduce. The class has a simple interface to:

- reset the state of the system after launch
- launch Pig (in cluster or local mode)
- explain how the generated Pig plan will be executed in the underlying infrastructure

Other methods provided are related to gathering runtime statistics and retrieving job status information. The most important methods of MapReduceLauncher are *compile()* and *launchPig()*. The compile method gets a Physical Plan and compiles it down to a Map-Reduce Plan. It is the point where all MapReduce optimizations take place. A total of eleven different optimizations are possible in this stage, including combiner optimizations, secondary sort key optimizations, join operations optimizations etc. The launchPig method receives the Physical Plan to be compiled and executed as a parameter and returns statistics collected during the execution. In short, it consists of the following simplified steps:

- Retrieves the optimized Map-Reduce Plan.
- Retrieves the Execution Engine. The JobClient class provides the primary interface for the user-code to interact with Hadoop's JobTracker. It allows submitting jobs and tracking their progress, accessing logs and status information. Usually, a user creates a JobConf object with the configuration information and then uses the JobClient to submit the job and monitor its progress.
- Creates a JobClient Object.
- Creates a JobControlCompiler Object. The JobControlCompiler compiles the Map-Reduce Plan into a JobControl object. The JobControl object encapsulates a set of MapReduce jobs and their dependencies. It tracks the state of each job and has a separate thread that submits the jobs when they become ready, monitors them and updates their states. This is shown in Figures 3.4 and 3.5.
- Repeatedly calls the JobControlCompiler's compile method until all jobs in the Map-Reduce Plan are exhausted
- While there are still jobs in the plan, it retrieves the JobTracker URL, launches the jobs and periodically checks their status, updating the progress and statistics information.
- When all jobs in the Plan have been consumed, it checks for native Map-Reduce jobs and runs them.
- Finally, it aggregates statistics, checks for exceptions, decides the execution outcome and logs it.

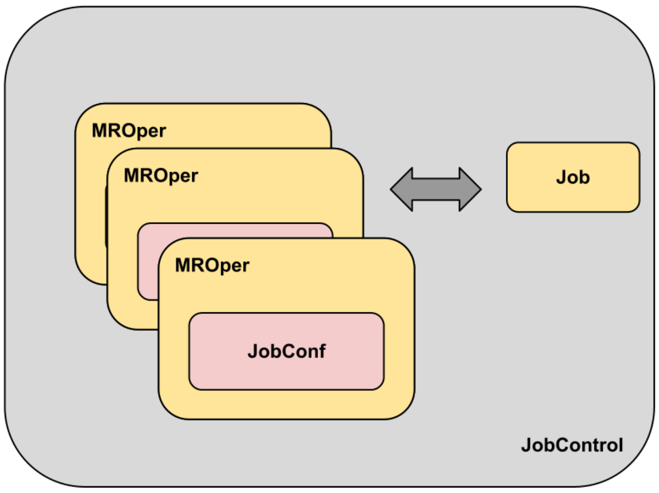


Figure 3.4: Inside a JobControl Object

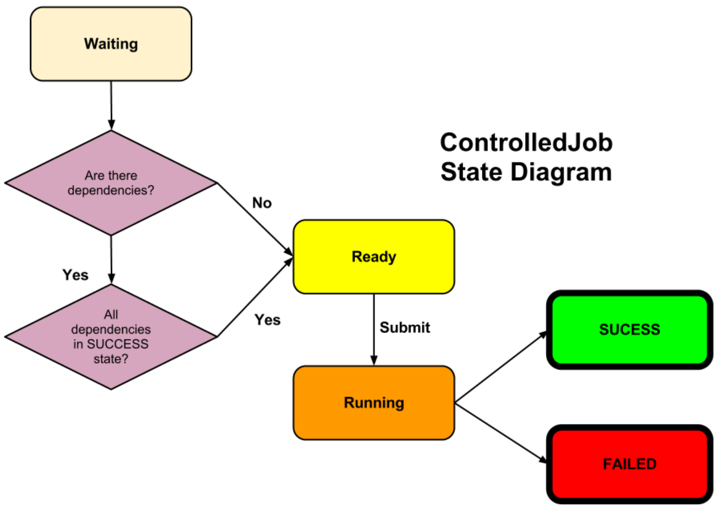


Figure 3.5: ControlledJob State Diagram

3.3.2 Plan Optimization

In this section we describe Pig's Logical Plan Optimizer. As we have already discussed in section 3.3.1, the initial Logical Plan is created by an one-to-one mapping of the Pig Latin statements to Logical Operators. The structure of this Plan is of course totally dependent on the scripting skills of the user and can result in highly inefficient execution. Pig performs a set of transformations on this plan before it compiles it to a Physical one. Most of them are trivial and have been long used in database systems and other high-level languages. However, is is still interesting to discuss them in the "Pig context".

3.3.2.1 Rules, RuleSets, Patterns and Transformers

The base optimizer class is designed to accept a list of RuleSets, i.e. sets of rules. Each RuleSet contains rules that can be applied together without conflicting with each other. Pig applies each rule in a set repeatedly, until no rule is longer applicable or it has reached a maximum number of iterations. It then moves to the next set and never returns to a previous set. Each rule has a pattern and an associated transformer. A pattern is essentially a sub-plan with specific node types. The optimizer will try to find this pattern inside the Logical Plan and if it exists, we have a match. When a match is found, the optimizer will then have to look more in depth into the matched pattern and decide whether the rule fulfills some additional requirements. If it does, then the rule is applied and the transformer is responsible for making the corresponding changes to the plan. Some extra caution is needed in two places. The current pattern matching logic assumes that all the leaves in the pattern are siblings. This assumption creates no problems with the existing rules. However, when new rules are designed, it should be kept in mind that the pattern matching logic might need to be changed. Another point that needs highlighting concerns the actual Java implementation. When searching for a matching pattern, the *match()* method will return a list of all matched sub-plans. Each one of them is a subset of the original plan and the operators returned are the same objects as in the original plan.

3.3.2.2 Optimization Examples

We present here a small subset of optimizations performed by Pig.

- **ColumnMapKeyPrune**
  This rules prunes columns and map keys that are not needed. More specifically, removes a column if it mentioned in a script but never used and a map key if it never mentioned in the script.

- **FilterAboveForeach**
  This rule pushes Filter operators above Foreach operators. However, it checks if the field that Filter works on is present in the predecessor of Foreach:

  ```
  A = LOAD 'file' AS (a, b, c);
  B = FOREACH A GENERATE a+2, b;
  C = FILTER B BY b>0;
  STORE C INTO 'output';
  ```

- **MergeFilter**
  This rule merges two consecutive Filter operators, adding the condition of the second Filter to the condition of the first Filter with an AND operator. The following script :

  ```
  B = FILTER A BY a>0;
  C = FILTER B BY b>0;
  ```

  will become:

  ```
  B = FILTER A BY (a>0 AND b>0);
  ```

- **MergeForeach**
  This rule merges Foreach operators, but it's not as simple as it sounds. There are a few additional requirements that need to be met. For example, if the first Foreach operator has a Flatten in its internal plan, the rule cannot be applied. Flatten is an advanced Pig operations that removed nesting. The optimizer also checks how many times the outputs of the first Foreach are used by the second Foreach. The assumption is that if an output is referred to more than once, the overhead of multiple expression calculation might even out the benefits from the application of this rule. For example, the optimization will not be performed in the following script:

  ```
  A = LOAD 'file' AS (x, y, z);
  B = FOREACH A GENERATE 2*x+y AS b;
  C = FOREACH B GENERATE b, b+z;
  ```

3.3.3 Compilation of Basic Pig Operators

In this post I will present some of the basic and most common and useful Pig operators. I will explain how they operate on data and what results they produce, but also how they are internally translated into Map-Reduce jobs and executed on the Hadoop execution engine.

- **FOREACH**
  FOREACH takes as input a record and generates a new one by applying a set of expressions to it. It is essentially a projection operator. It selects fields

from a record, applies some transformations on them and outputs a new record. FOREACH is a non-blocking operator, meaning that it can be included inside the current Map-Reduce operator.

- **FILTER**
  FILTER selects those records from dataset for which a predicate is true. Predicates can contain equality expressions, regular expressions, boolean operators and user-defined functions. FILTER is also non-blocking and can be merged in the current Map or Reduce plan.

- **GROUP BY**
  GROUP collects all records with the same key inside a bag. GROUP generates records with two fields: the corresponding key which is assigned the alias "group" and a bag with the collected records for this key. We can group on multiple keys and we can also GROUP "all". GROUP all will use the literal "all" as a key and will generate one and only record with all the data in it. This can be useful if we would like to use some kind of aggregation function on all our records, e.g. COUNT. GROUP is a blocking operator and it compiles down to three new operators in the Physical Plan: Local Rearrange, Global Rearrange and Package. It requires repartitioning and shuffling, which will force a Reduce phase to be created in the MapReduce plan. If we are currently inside a Map phase, then this is no big problem. However, if we are currently inside a Reduce phase, a GROUP will cause the pipeline to go through Map-Shuffle-Reduce.

- **ORDER BY**
  The ORDER BY operator orders records by one or more keys, in ascending or descending order. However, ORDER is not implemented as simply as Sorting-Shuffle-Reduce. Instead ,it forces the creation of two MapReduce jobs. The reason is that datasets often suffer from skew. That means that most of the values are concentrated around a few keys, while other keys have much less corresponding values. This phenomenon will cause only a few of the reducers to be assigned most of the workload, slowing down the overall execution. The first MapReduce job that Pig creates is used to perform a fast random sampling of the keys in the dataset. This job will figure out the key distribution and balance the load among reducers in the second job. However, this technique breaks the Map-Reduce convention that all records with the same key will be processed by the same reducer.

- **JOIN**
  No matter how common and trivial, join operations have always been a headache to MapReduce users. The problem originates from MapReduce's Map-Shuffle-Sort-Reduce static pipeline and single input second-order functions. The challenge is finding the most effective way to "fit" the join operation into this programming model. The most common strategies are two and both consist of one Map-Reduce job:

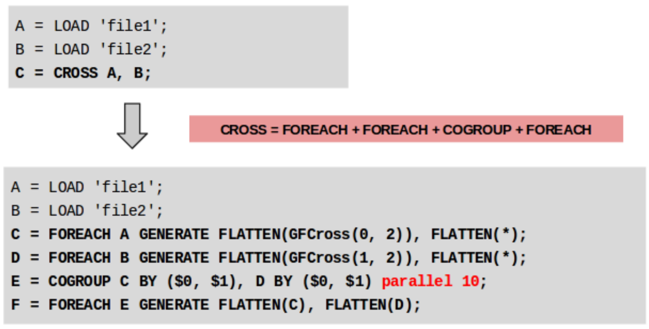
Reducer-side join: In this strategy, the map phase serves as the preparation phase. The mapper reads records from both inputs and tags each record with a label based on the origin of the record. It then emits records setting as key the join key. Each reducer then receives all records that share the same key, checks the origin of each record and generates the cross product.

Mapper-side join: The alternative comes from the introduction of Hadoop's distributed cache. This facility can be used to broadcast one of the inputs to all mappers and perform the join in the map phase. However, it is quite obvious that this technique only makes sense in the case where one of the inputs is small enough to fit in the distributed cache. Fortunately, Pig users do not need to program the join operations themselves. Pig's default join is the Reducer-side join described above. However, Pig users can use the JOIN operator in pair with the USING keyword in order to select more advanced join execution strategies. Pig's advanced join strategies and their compilation to MapReduce can be found in (21).

- **COGROUP**
  COGROUP is a generalization of the GROUP operator, as it can group more than one inputs based on a key. Of course, it is a blocking operator and is compiled in a way similar to that of GROUP.

- **UNION**
  UNION is an operator that concatenates two or more inputs without joining them. It does not require a separate Reduce phase to be created. An interesting point about UNION in Pig is that it does not require the input records to share the same schema. If they do, then the output will also have this schema. If the schemas are different, then the output will have no schema and different records will have different fields. Also, it does not eliminate duplicates.

- **CROSS**
  CROSS will receive two or more inputs and will output the cartesian product of their records. This means that it will match each record from one input with every record of all other inputs. If we have an input of size n records and an input of size m records, CROSS will generate an output with n*m records. The output of CROSS usually results in very large datasets and it should be used with care. CROSS is implemented in a quite complicated way. A CROSS logical operator is in reality equivalent to four operators as shown in Figure 3.6. The GFCross function is an internal Pig function and its behavior depends on the number of inputs, as well as the number of reducers available (specified by the "parallel 10" in the script). It generates artificial keys and tags the records of each input in a way that only one match of the keys is guaranteed and all records of one input will match all records of the other.



```
A = LOAD 'file1';
B = LOAD 'file2';
C = CROSS A, B;
```

```
A = LOAD 'file1';
B = LOAD 'file2';
C = FOREACH A GENERATE FLATTEN(GFCross(0, 2)), FLATTEN(*);
D = FOREACH B GENERATE FLATTEN(GFCross(1, 2)), FLATTEN(*);
E = COGROUP C BY ($0, $1), D BY ($0, $1) parallel 10;
F = FOREACH E GENERATE FLATTEN(C), FLATTEN(D);
```

Figure 3.6: CROSS operator compilation in MapReduce

3.4 Comparison with MapReduce

In the previous chapter, we saw that MapReduce is a very simple yet powerful framework for analyzing large datasets in parallel. Pig's goal is to make parallel processing even easier. In this section, we present a brief comparison of the two systems.
Clearly, Pig's main advantage over MapReduce lies in its data-flow language, Pig Latin, and its built-in common data processing operations, such as join, filter, group by, order by, union, etc. Group by is directly provided by MapReduce during the shuffle and reduce phases and order by is also provided for free as a side-effect of the way grouping is implemented. The complexity of implementing the rest of the operations ranges from trivial (filter, projection) to complicated (join). Pig's implementations of these operations are quite sophisticated and use advanced engineering techniques in order to make them more efficient (e.g. sampling in order by).
Another important advantage of Pig over MapReduce is its ability to perform optimizations on the user code. When writing a MapReduce program, the framework has no information on the user code. On the other hand, Pig creates the logical plan from the a Pig Latin script, which can then be analyzed and reveal possible transformations that could be applied on the dataflow. As a result, it also can perform early error checking and optimizations.
Another difference is that MapReduce has no type system. This is a choice that provides the users with great flexibility in defining their own types and serialization methods. However, the absence of a type system essentially prevents the framework from being able to perform any kind of code checking both before and during runtime.
In conclusion, writing and maintaining Pig Latin code poses a clearly lower cost to the developers. Yet, there exist algorithms that would be much more difficult to develop with Pig than with MapReduce. An additional downside is that, when using Pig, the developer has less control over the program behavior. This is true for any kind of high-level system. Not needing to have a deep understanding oh how the underlying implementation works is convenient, but it can also lead to highly inefficient code. It is true that, given enough time, it is always possible to write a program in a

low-level system that will out-perform any generic system. Consequently, native MapReduce should be preferred over Pig for performance sensitive applications. The universal rule that developers should choose the right tool for the job applies here as well.

Chapter 4
4 Stratosphere

Stratosphere is a data-processing framework, under research by TU Berlin. It provides a programming model for writing parallel data analysis applications, PACT, and an execution engine, Nephele, able to execute dataflow graphs in parallel. Stratosphere can be seen as a generalization of Hadoop Map-Reduce. In this section, both Nephele and PACT are addressed in more detail.

4.1 The Nephele Execution Engine

Nephele is an execution engine designed to execute DAG-based data flow programs. It takes care of task scheduling and setting up communication channels between nodes. Moreover, it supports dynamic allocation of resources during execution and fault-tolerance mechanisms.
The programming model provided with Stratosphere is PACT. However, it is possible to submit jobs directly to the Nephele engine, in the form of Directed Acyclic Graphs (DAGs). Each vertex of the graph represents a task of the job. There are three types of vertices: Input, Output and Task. The edges of the graph correspond to the communication channels between tasks.

4.1.1 Nephele Advantages

Nephele allows parametrization in a high degree, in order to achieve several optimizations. For example, one can set the degree of data parallelism per task or explicitly specify the type of communication channels between nodes. Another important advantage of Nephele compared to other engines is the possibility for dynamic resource allocation. Nephele is capable of allocating resources from a Cloud environment, depending on the workload. On the contrary, MapReduce and Dryad have been designed to work in cluster environments only.

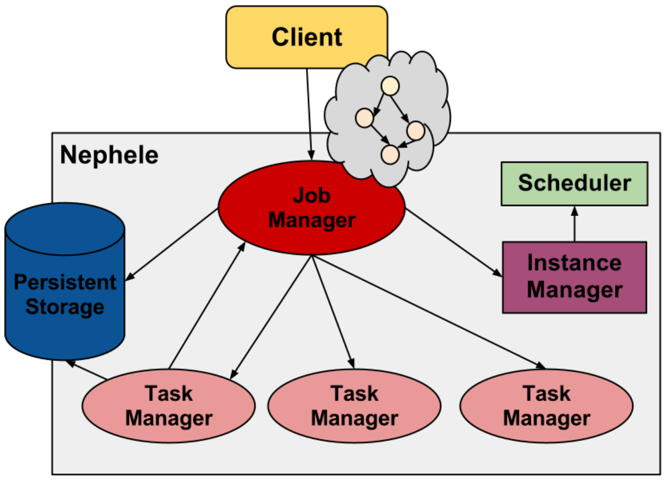


Figure 4.1: Nephele Execution Engine Architecture

4.1.2 Nephele Architecture

The architecture of the Nephele execution engine is presented in Figure 4.1. In order to submit a job to the Nephele engine, a Client has to communicate with the the Job Manager. The Job Manager is unique in the system and is responsible for scheduling the jobs it receives and coordinating their execution. The resources required for job execution are managed by the Instance Manager. The Instance Manager allocates or deallocates virtual machines depending on the workload of the current execution phase. The jobs are executed in parallel by instances, each of which is controlled by a Task Manager. The Task Manager communicates with the Job Manager and is assigned jobs for execution. During execution, each Task Manager sends information about changes in the execution state of the job (completed, failed, etc.). Task Managers also periodically send heartbeats to the Job Manager, which are then propagated to the Instance Manager. This way, the Instance Manager is keeping track of the availability running instances. If a Task Manager has not sent a heartbeat in the given heartbeat interval, the host is assumed to be dead. The Instance Manager then removes the respective Task Manager from the set of compute resources and calls the scheduler to take appropriate actions.
When the Job Manager receives a job graph from the Client, it decides how many and what types of instances need to be launched. Once all virtual machines have booted up, execution is triggered. Persistent storage, accessible from both the Job and Task Managers, is needed to store the jobs' input and output data.

4.1.3 Nephele Jobs

Jobs in Nephele are defined as Directed Acyclic Graphs (DAGs). Each graph vertex represents one task and each edge indicates communication flow between tasks. Three types of vertices can be defined: Task vertex, Input vertex and Output vertex. The Input and Output vertices define how data is read or written to disk. The Task vertices are where the actual user code is executed.
Nephele defines a default strategy for setting up the execution of a job. However, there is a set of parameters that the user can tune in order to make execution more efficient. These parameters include the number of parallel subtasks, the number of subtasks per instance, how instances should be shared between tasks, the types of communication channels and the instance types that fulfill the hardware requirements of a specific job.
Nephele offers three types of communication channels that can be defined between tasks. A Network Channel establishes a TCP connection between two vertices and allows pipelined processing. This means that records emitted from one task can be consumed by the following task immediately, without being persistently stored. Tasks connected with this type of channel are allowed to reside in different instances. Network channels are the default type of communication channel chosen by the Nephele, if the user does not specify a type. Subtasks scheduled to run on the same instance can be connected by an In-Memory Channel. This is the most effective type of communication and is performed using the instance's main memory, also allowing data pipelining. The third type of communication is through File Channels. Tasks that are connected through this type of channel use the local file system to communicate. The output of the first task is written to an intermediate file, which the serves as the input of the second task.

4.2 The PACT Programming Model

The PACT programming model is a generalization of the MapReduce programming model and aims to overcome its limitations. It extends the idea of the Map and Reduce second-order functions, introducing the Input Contracts. An Input Contract is a secondary function that accepts a first-order user-defined function and one or more data sets as inputs. Input Contracts do not have to form any specific type of pipeline and can be used in any order that respects their input specifications. In the context of the PACT programming model, Map and Reduce are Input Contracts. Apart from these two, three more Contracts are defined:

- The *Cross* Input Contract accepts multiple inputs of key value pairs and produces subsets of all possible combinations among them, building a Cartesian product over the input. Each combination becomes then an independent subset.

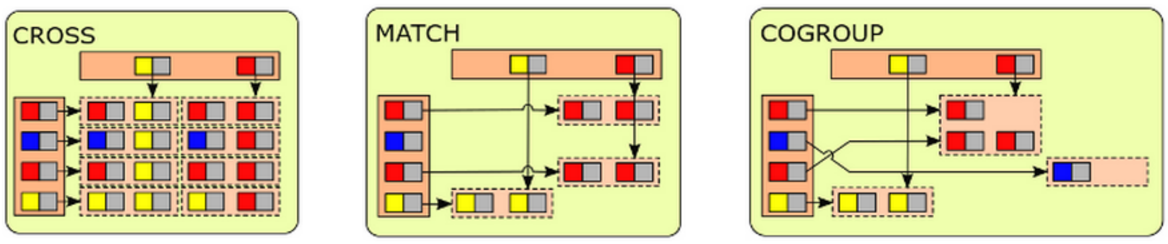


Figure 4.2: The Cross, Match and CoGroup Input Contracts. Same colour represents equal keys.

- The *Match* Contract operates on two inputs and matches each pair of the first input with one pair of the second input that has the same key value. This contract naturally maps to an inner join, in terms of database semantics.
- Finally, the *CoGroup* function creates independent subsets by combining all pairs that share the same key.

Figure 4.2 illustrates the functionality of the three Contracts mentioned above.

4.2.1 Output Contracts

In addition to Input Contracts, PACT also offers, Output Contracts. These are optional compiler hints that can be used for further optimizations, under certain circumstances. More specifically, Output Contracts assert certain properties of a PACT's output data. These assertions can then be exploited by the compiler in order to create a more effective partitioning strategy and execution plan. Currently, the following Output Contracts are supported in Stratosphere:

- **Same-Key**
  In order for this Output Contract to be used, each output key-value pair that is generated by the user function needs to have the same key as the key-value pair(s) it was generated from. In other words, the function preserves the partitioning and order on the keys.
- **Super-Key**
  A Super-Key Output Contract can be used when the user function generates key-value pairs whose keys are superkeys of the input key-value pair(s). This function will preserve the partitioning and partial oder on the keys.
- **Unique-Key**
  For this Output Contract to be used, it must be guaranteed that the user function used will generate a unique key for each output key-value pair across all parallel instances.

Although Output Contracts are an important feature of the Stratosphere system, they were not used for our Pig on Stratosphere implementation, due to time limitations. However, we believe that it would be beneficial to extend the Pig Latin language in order to include special keywords, which could be translated into Stratosphere Output Contracts.

4.3 Relational Operations With PACTs

Expressing common relational operations with PACTs is, in most cases, quite straightforward. This is not true for MapReduce, whose static pipeline often complicates the implementation and makes applications hard to write, understand and maintain. In this section, we present how Stratosphere's Input Contracts can be used in order to implement a set of the most typical relational operations.

- **Inner-Join**
  An inner join naturally maps to the *Match* Input Contract. *Match* guarantees that all pairs from the two (or more) inputs that share the same key, will be processed individually and by exactly one instance. The user function used with *Match* will only have to perform the concatenation of the values and emit the new pair.

- **Outer-Join**
  An outer join can be implemented in Stratosphere using *CoGroup*. The user function can be the identity function. If for some key, the set of values is empty in one of the inputs, the other input's pairs will be concatenated with null values.

- **Anti-Join**
  CoGroup can also be used to implement an anti-join. In this case, the user function will have to ignore all the records from one input if there are records with the same key in the other input.

- **Theta-Join**
  A theta-join can be realized in Stratosphere using the *Cross* Contract. The user function will then have to check the specific conditions of the join and decide if the record should be emitted or not.

Apart from the different possible join operations, the cartesian product and the the co-grouping by more than one key can be directly implemented using the *Cross* and the *CoGroup* Input Contracts respectively.

4.4 Comparison of MapReduce and PACT

The MapReduce programming model was designed with the aim to simplify the parallelization of simple analytical tasks over large amounts of data and serves this purpose very well. However, for more complex tasks, PACT makes application developing easier. Tasks that would require a series of MapReduce jobs can be naturally implemented using only a few Input Contracts. This is partially achieved by the design decision of PACT to accept more than one data set as input, in contrast to MapReduce. Moreover, PACT overcomes the limitation of a static pipeline, introducing dynamic execution plans and on-the-fly creation of data-flow graphs.

Chapter 5

5 Pig and Stratosphere Integration

In this chapter, we describe how we designed and realized the integration between Pig and Stratosphere. We first present our logic behind the integration strategy chosen. We discuss difficulties and challenges faced and we provide solutions. We then present our Pig to PACT translation algorithm and provide an illustrated example. Next, we provide details on out implementation, describing the structure of the project, the main packages and classes implemented and the changes we made to the Pig codebase.

5.1 Integration Alternatives

In this section, we present some alternative design choices concerning the actual implementation of the project, i.e. the integration of Pig and Stratosphere systems. The main goal is to have a working system, such that Pig Latin scripts can be executed on top of the Nephele execution engine. However, performance is also an issue, and of course, we would not like to construct a system that would be less effective than the current implementation. The architectures of the two systems are shown side by side in Figure 5.1. Looking at the architecture flows, it is obvious that several solutions exist and the integration can be achieved in different ways and on different levels. Several feasible solutions were evaluated and are presented in this section.

- **Translate MapReduce programs into PACT programs**
  This is the naive and most straight-forward way of solving the problem, shown in Figure 5.2. PACT already supports Map and Reduce Input Contracts, which can be used for the transformation of the Map-Reduce Plan into a one-to-one PACT Plan. The Logical and Physical Plans that are generated by Pig can be re-used without modification. However, it is obvious that this solution would not provide any gains compared to the existing implementation. In fact, it should be slower, since it adds one more layer to the system architecture. This approach seems to be the simplest and it was the starting point for our

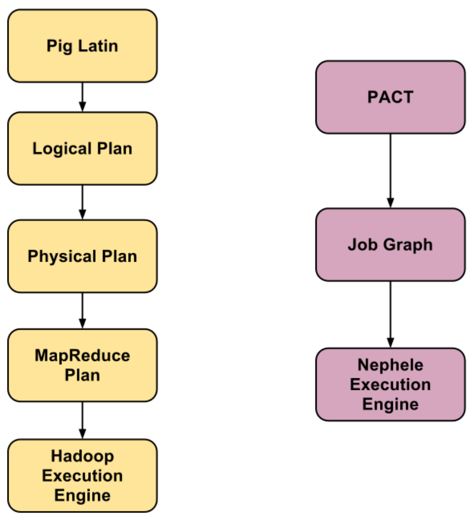


Figure 5.1: Pig and Stratosphere Architectures

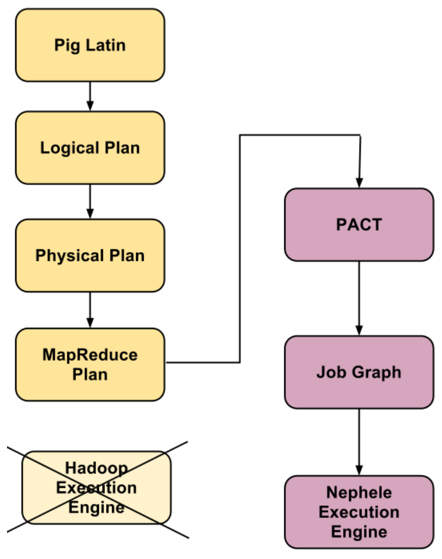


Figure 5.2: Direct MapReduce to PACT integration solution

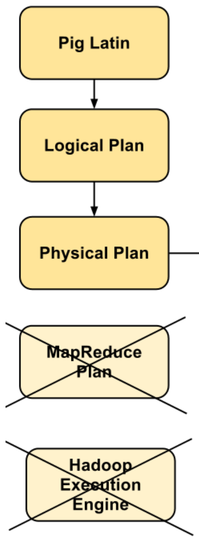


Figure 5.3: Physical to PACT plan integration solution

implementation. However, it proved to be problematic. Although one might think that the implementation would be a simple task, the dependencies of Pig with Hadoop on that level made it almost impossible to proceed. On the MapReduce plan level, all physical operators have already been grouped into map and reduce phases and Hadoop-specific classes and interfaces are used extensively in this layer of the Pig codebase. In order to transform the generated MapReduce plan into a PACT plan, one has to find mapping for these classes in the Stratosphere context, while also preserving the interfaces to the rest of the codebase. For example, it is necessary to express the functionality of a Hadoop Job or a JobControl object in the Stratosphere context. Unfortunately, there is no such class in Stratosphere that provides the same functionality. The information that was encapsulated inside these objects could not be easily mapped to Stratosphere objects. According to our experience, working on this level is not the correct approach.

- **Translate the Physical Plan into a PACT Plan**
  A second approach would be moving one level up in the Pig architecture flow and try to translate the Physical plan into a PACT plan, as illustrated in Figure 5.3. Our analysis showed that the Physical Plan is also very dependent on the MapReduce framework and does not reflect the correct level for integration with another execution engine. When the Physical Plan is created, the logical operators have already been translated in a way specific to the underlying execution engine and it was not possible to use the additional Input Contracts, such as Match, Cross and CoGroup. For example, the (CO)GROUP operation is compiled down to three new operators and the CROSS operation is compiled

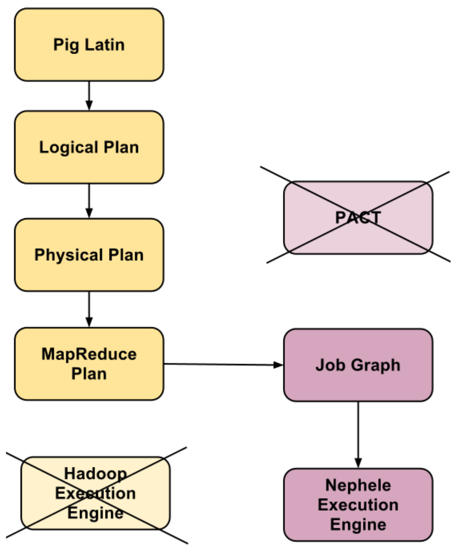


Figure 5.4: MapReduce/Physical plan to Nephele Job integration solution

down to four. However, in the case of Stratosphere, they can be mapped directly to the CoGroup and Cross Input Contracts respectively. On this level, one would be obliged to re-implement relational operations using only map and reduce operations. This would lead, of course, into loss of performance and would not provide any advantages over the current solution.

- **Translate the MapReduce / Physical Plan into a Nephele Job**
  Looking at the two system architectures, one might consider logical that the more layers we remove the faster the resulting system would be. For example, one could argue that getting rid of both the high-level programming frameworks, Map-Reduce and PACT, would speed up things. This alternative is shown in Figure 5.4. However, merging at that point, would include re-implementing a job already done, i.e. compiling down to code that can be understood by an execution engine, such as Nephele (or Hadoop). A speedup in this case is quite improbable to happen and it should mean that there are inefficiencies or design flaws in the PACT compiler. We had no reason to believe that this would be true, so this alternative solution was not investigated further.

- **Translate the Logical Plan into a PACT Plan**
  The evaluation of the solutions discussed above made it clear that we had to move even higher in the Pig architecture in order to achieve our goal. Such a decision would mean less dependencies with the Hadoop execution engine, more implementation freedom, but also a much larger amount of code to be written and functionality to be implemented. Translating the Logical Plan into a set of Pact Operators eventually proved to be a reasonable approach. The

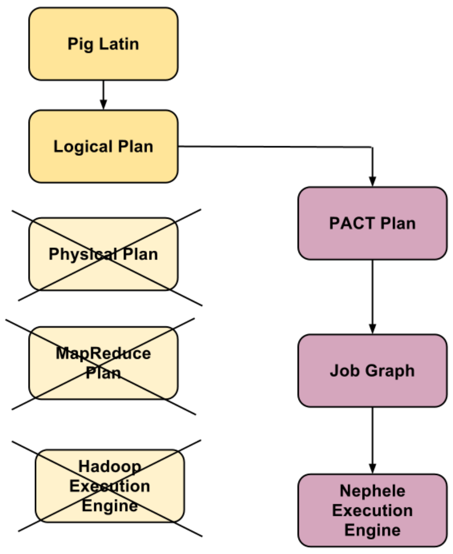


Figure 5.5: Logical to PACT Plan integration solution

solution is shown in Figure 5.5. The translation algorithm used is presented in detail in the following section.

The solutions discussed here are not the only ones possible. One could think of and propose several variations on different levels. We chose the last solution after having evaluated several other and also having in mid that when any kind of abstraction is made, and this applies as well for high-level languages, there is always an overhead you have to pay in exchange for simplicity. The underlying system, of which the details the abstraction aims to hide, will be designed to take several decisions that would often differ from those an experienced low-level programmer would take. However, the abstraction only has value, provided that the frustration imposed to the user by the slow-down of accomplishing their job, is lower than the satisfaction they get by being able to accomplish this job in a simpler way.

5.2 Pig to PACT Plan Translation Algorithm

Pig Latin offers a large set of commands that are used for input and output, relational operations, advanced operations and the declaration of user-defined functions. We chose a subset of these statements, which, according to our judgment, are the most common and useful ones and we describe here how they could be translated into PACT operators.

- **Input / Output**
  Pig provides the LOAD and the STORE commands for data input and output.

These two logical operators can be mapped directly to the GenericDataSource and the GenericDataSink Input Contracts of Stratosphere. In our implementation, we only support input and output from and to files, so we have used the more appropriate Contracts, FileDataSource and FileDataSink. Similar to a Pig script, a PACT plan can have one or more FileDataSource Contracts, where the user can specify a file path and several parameters related to the input data format.

- **Relational Operators**
  As we described in the previous chapter, PACTs offer a natural way to express most of the common relational operations. A FILTER or a FOREACH statements do not require the creation of a new Input Contract and could be easily merged in the user-function of the current one. The GROUP logical operator naturally maps to the Reduce Input Contract, while INNER and OUTER JOIN operations can be implemented using the Match and CoGroup Input Contracts as explained in the previous chapter. Pig's ORDER BY operator can sort the input records in ascending or descending order, specifying one or more record fields as the sorting key. As explained in Chapter 3, Pig realizes the ORDER BY operation by creating 2 MapReduce jobs: During the first job, it performs a sampling of the input to estimate the key distribution. It then builds a custom partitioner and uses it to balance the keys among the reducers. With PACTs, the same functionality can be offered in a much simpler way. The Stratosphere version 0.2, the GenericDataSink Contract will provide a *setGlobalOrder()* method, which will allow the user to specify parameters similar to the ones Pig offers, i.e. sorting keys and ascending / descending order.

- **Advanced Operators**
  From the set of the advanced Pig operators, we chose to implement the CROSS and the UNION. The CROSS operator can be mapped to the Cross Input Contract, while the Map Input Contract can be used to realize the UNION. In the Stratosphere version 0.2, the Map Contract will offer an *addInput()* method, which could be called once for each input of the UNION operator.

Our Pig to PACT translation algorithm mappings are illustrated in Figure 5.6.

Pig's logical plan is traversed in a depth-first fashion and a visitor is attached to each node. Depending on the type of the node operator, a specific *visit()* method will be invoked that will create the necessary PACT operators. The traversal starts from the plan's roots as following:

```
roots <- LogicalPlan.getRoots()
FOREACH r in roots
    visit(r)
```

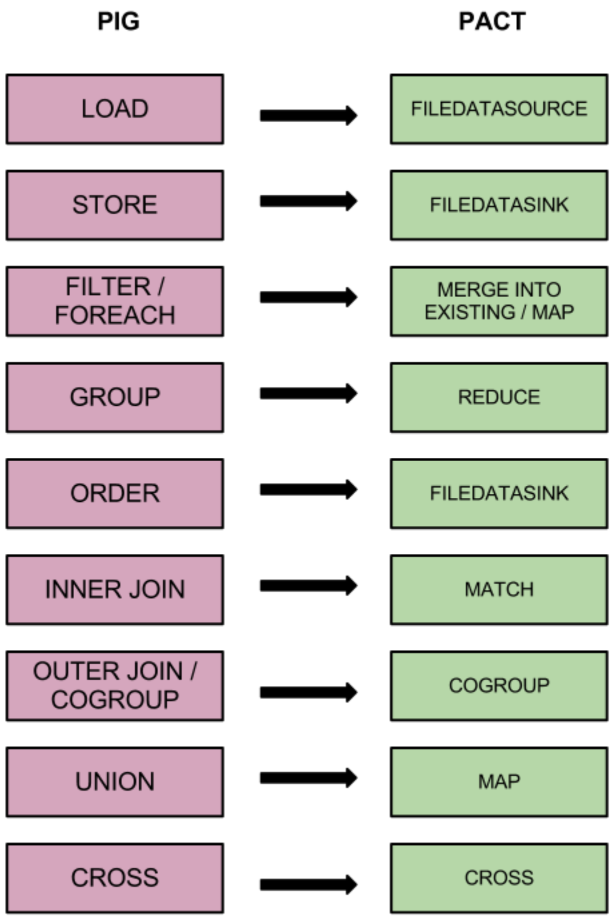


Figure 5.6: Pig to PACT translation algorithm operator mappings

The *visit()* method is responsible for recognizing the operator type and creating the appropriate PACT operator, according to the mappings of Figure 5.6. It is also responsible for setting the correct parameters, such as data types, operator alias, result types, as well as connecting the newly created operator to its correct predecessors. This way, a graph of PACT operators is being gradually constructed. Finally, the *visit()* method is also responsible for saving the mapping between the logical and the PACT operators in a HashMap object. For example, the *visit()* that would be invoked in the case of a LOAD operator would perform the actions shown below:

```
visit(loLoad){
        FileDataSourceOperator fds = new FileDataSourceOperator();
        fds.setAlias();
        fds.setLFile();
        fds.setResultType();
        fds.setSignature();
        currentPlan.add(fds);
        logToPactMap.put(loLoad, fds);

        List<Operator> op = loLoad.getPlan().getPredecessors(loLoad);
        PactOperator from;

        if(op != null) {
             from = logToPactMap.get(op.get(0));
             currentPlan.connect(from, fds);
        }
}
```

5.2.1 Example

We present here a representative example aiming to clarify the translation algorithm and demonstrate how much PACT simplifies the translation process. The example chosen is a script taken from the original Pig paper (22). We provide here the Pig Latin code for the script:

```
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A BY y>0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE group, COUNT(D);
STORE F INTO 'out';
```

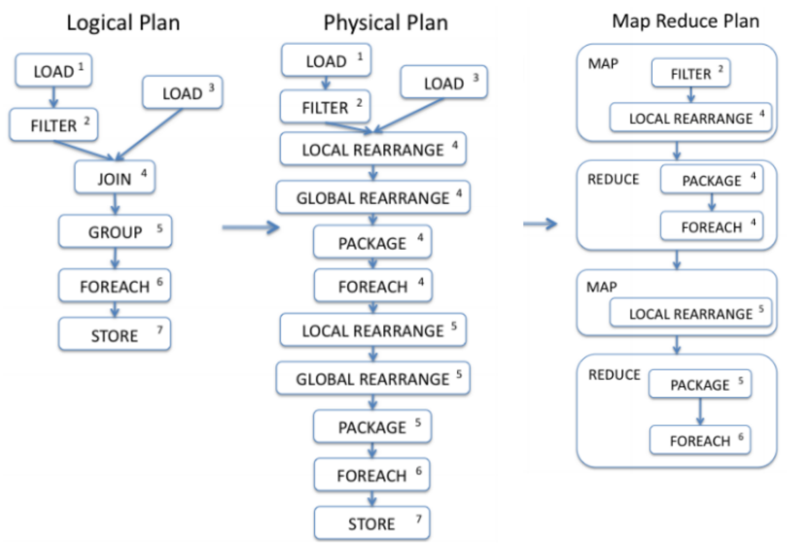


Figure 5.7: Plan generation in Pig

This simple script loads data from two different files into the sets A and B. It then filters input A and performs a JOIN operation between the two inputs. Next, a GROUP is performed on the output of the JOIN, a COUNT is calculated and the final output is stored into a file. The Logical, Physical and MapReduce plans generated by Pig for this script are shown in Figure 5.7. The Logical Plan is created directly from the parsed representation of the Pig Latin script and contains one node for each Pig Latin statement. In order for the Physical Plan to be created, the Logical Plan is traversed and operators are translated into their Physical equivalents. In Figure 5.7, the numbers inside the graph nodes are used in order to show the mappings from Logical to Physical operators. For example, a LOAD Logical operator is translated into a LOAD Physical operator, while a JOIN Logical Operator will be translated into a series of four Physical operators: LOCAL REARRANGE, GLOBAL REARRANGE, PACKAGE and FOREACH. After the Physical Plan has been created, it is traversed in the same way as the Logical Plan, in order to generate the MapReduce Plan. In the beginning, a Map phase is created and physical operators are pushed into it until a blocking operator is found. This is the case for the GLOBAL REARRANGE in our example. At this point, the GLOBAL REARRANGE will be removed from the plan, as it is already provided by the MapReduce framework through the shuffling phase. In its place, a Reduce phase is created and all subsequent operators are pushed into this phase, until the next blocking operator is found. The result of this process is a MapReduce Plan consisting of two MapReduce jobs.
Figure 5.8 shows how the same script would be compiled using our Pig to PACT algorithm. Each one of the LOAD operators become a FileDataSource operator and the FILTER can be merged in one of them. Next, the JOIN is simply translated into a Match Operator, while the GROUP becomes a Reduce Operator. The subsequent

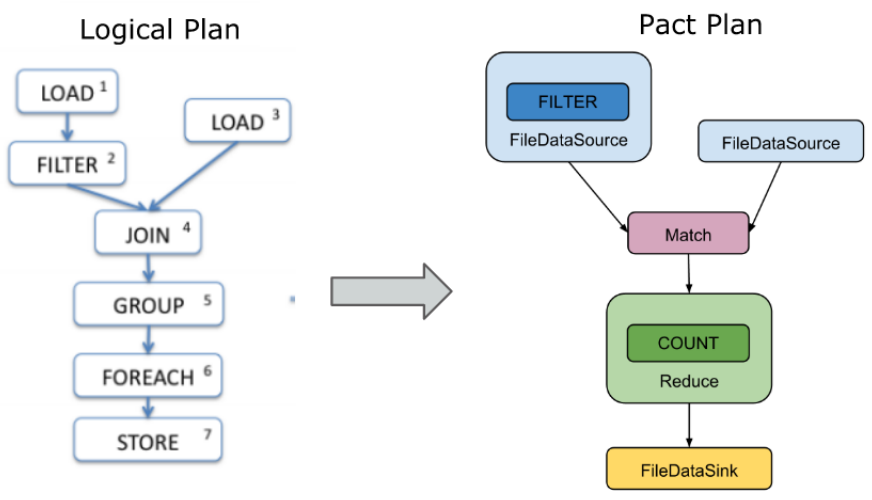


Figure 5.8: Plan generation using PACTs

FOREACH with its user function COUNT can then be merged inside the Reduce Operator. Finally, the STORE will be mapped into a FileDataSink Operator. The resulting Plan is much smaller and also reflects much more clearly the script semantics. Compared to Pig's logical-to-physical translation algorithm, our algorithm is much simpler. Most of the operators can be mapped to exactly one PACT and there is no need for categorizing operators into blocking and non-blocking or keeping track of phases. In contrast to the MapReduce static pipeline, PACTs can be used in any order and can be linked to each other creating a directed acyclic graph very similar to Pig's logical plan. Consequently, each operator can be compiled individually and no state needs to be maintained between operator compilation. The only case when the algorithm needs to be aware of the current operator is this of a FILTER or a FOREACH statement. However, the algorithm could be simplified even more and create a Map Input Contract for these operators. In this case, the overhead of creating a new contract would be equal only to the serialization and de-serialization overhead of the inputs.

5.3 Implementation

This section discusses some technical details about the implementation of our Pig on Stratosphere prototype. As both Pig and Stratosphere are written in Java, we also chose the same implementation language. We started implementing the prototype as a separate project, which we later merged into the Pig codebase. Since the first phase of this project included an extended study of the Pig codebase, we were already familiar with the programming style, conventions and structure of the Pig project

and we tried to follow the same style in our own implementation. Pig classes were reused wherever possible or wrappers were created in order to make them compatible with the new features. As discussed in the beginning of this Chapter, the only parts of the Pig project that were not tightly coupled to the Hadoop execution engine were the parser and the Logical Plan. This was the point were we drew the line of separation and where we started re-implementating the rest of the stack according to Stratosphere standards.
Following Pig's style, our implementation can be divided in two stages after the Logical Plan. The first stage implements the translation of the Logical Plan into a Plan of PACT operators. This PACT Plan can be seen as the equivalent of Pig's Physical Plan. The second stage includes the translation of this PACT Plan into actual Input Contracts and the submission of the PACT plan to the Nephele execution engine.
In total, 10 Java packages and 66 new classes were created. Below, we briefly present the functionality included in the most important of them.

- **thesis.operators**
  This is the package that contains the implementations of the PACT operators, substitutes for Pig's Physical Operators. The abstract class *PactOperator* is the main superclass of all operators and defines their functionality. It extends Pig's org.apache.pig.impl.plan.Operator class. PactOperators are the objects that are members of the PactPlan. A Logical Plan is translated into a PactPlan, which is a dataflow graph of PactOperators. Each PactOperator corresponds to an InputContract, a FileDataSource or FileDataSink. All supported PACT operators inherit from *PactOperator*. Currently, the implemented classes include *CoGroupOperator*, *CrossOperator*, *FileDataSinkOperator*, *FileDataSourceOperator*, *MapOperator*, *MatchOperator* and *ReduceOperator*. *PactOperator* and its subclasses offer methods to attach and detach inputs, to fetch the next data tuple, to perform an operation on a tuple and return a result tuple. A brief UML diagram of the main classes in this package is shown in Figure 5.9.

- **thesis.pactLayer**
  This is the package that contains the classes necessary for the translation of the Logical Plan into a PACT Plan. This package contains the *PactPlan* and the *LogicalToPactCompiler* classes. The *PactPlan* class describes the compiled graph whose nodes are PactOperator objects corresponding to PACTs. It is a subclass of Pig's *org.apache.pig.impl.plan.OperatorPlan*. *PactPlan* offers methods to add, remove and replace a PactOperator from the plan, to connect PactOperator objects, to write a visual representation of the Pact Plan into the given output stream and to check if the PactPlan is empty. The *LogicalToPactCompiler* class is the main Compiler class that translates a Pig Logical Plan into a Plan of Pact Operators. It is implemented as a subclass of Pig's *org.apache.pig.newplan.logical.relational.LogicalRelationalNodesVisitor* and overrides the visit methods for the limited set of the supported operations

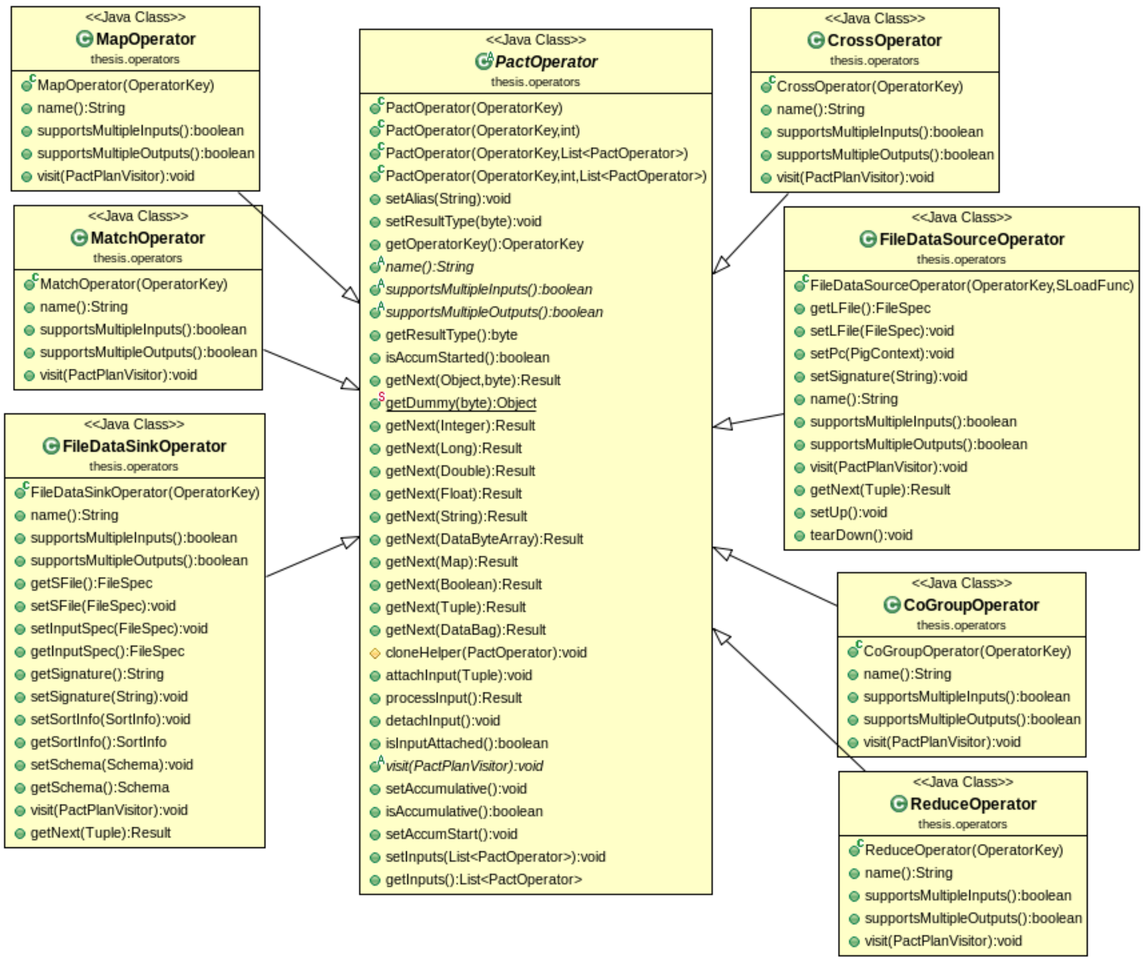
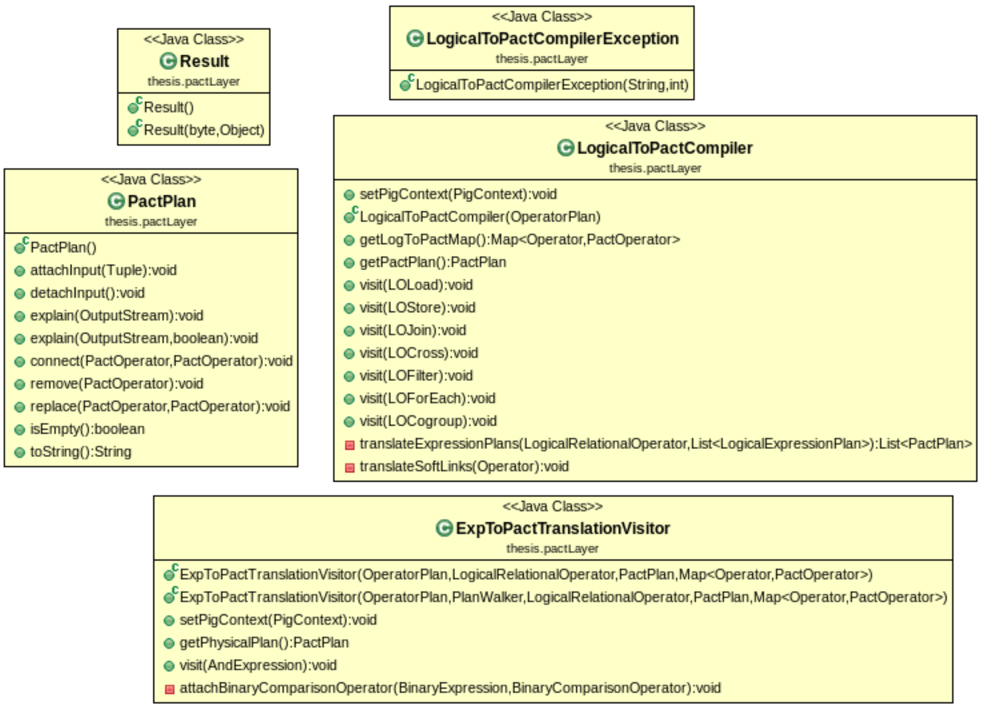

<<Java Class>>
MapOperator
thesis.operators
MapOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void
<<Java Class>>
MatchOperator
thesis.operators
MatchOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void
<<Java Class>>
FileDataSinkOperator
thesis.operators
FileDataSinkOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
getSFile():FileSpec
setSFile(FileSpec):void
setInputSpec(FileSpec):void
getInputSpec():FileSpec
getSignature():String
setSignature(String):void
setSortInfo(SortInfo):void
getSortInfo():SortInfo
setSchema(Schema):void
getSchema():Schema
visit(PactPlanVisitor):void
getNext(Tuple):Result
<<Java Class>>
PactOperator
thesis.operators
PactOperator(OperatorKey)
PactOperator(OperatorKey,int)
PactOperator(OperatorKey,List<PactOperator>)
PactOperator(OperatorKey,int,List<PactOperator>)
setAlias(String):void
setResultType(byte):void
getOperatorKey():OperatorKey
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
getResultType():byte
isAccumStarted():boolean
getNext(Object,byte):Result
getDummy(byte):Object
getNext(Integer):Result
getNext(Long):Result
getNext(Double):Result
getNext(Float):Result
getNext(String):Result
getNext(DataByteArray):Result
getNext(Map):Result
getNext(Boolean):Result
getNext(Tuple):Result
getNext(DataBag):Result
cloneHelper(PactOperator):void
attachInput(Tuple):void
processInput():Result
detachInput():void
isInputAttached():boolean
visit(PactPlanVisitor):void
setAccumulative():void
isAccumulative():boolean
setAccumStart():void
setInputs(List<PactOperator>):void
getInputs():List<PactOperator>
<<Java Class>>
CrossOperator
thesis.operators
CrossOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void
<<Java Class>>
FileDataSourceOperator
thesis.operators
FileDataSourceOperator(OperatorKey,SLoadFunc)
getLFile():FileSpec
setLFile(FileSpec):void
setPc(PigContext):void
setSignature(String):void
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void
getNext(Tuple):Result
setUp():void
tearDown():void
<<Java Class>>
CoGroupOperator
thesis.operators
CoGroupOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void
<<Java Class>>
ReduceOperator
thesis.operators
ReduceOperator(OperatorKey)
name():String
supportsMultipleInputs():boolean
supportsMultipleOutputs():boolean
visit(PactPlanVisitor):void

Figure 5.9: UML diagram for package thesis.operators



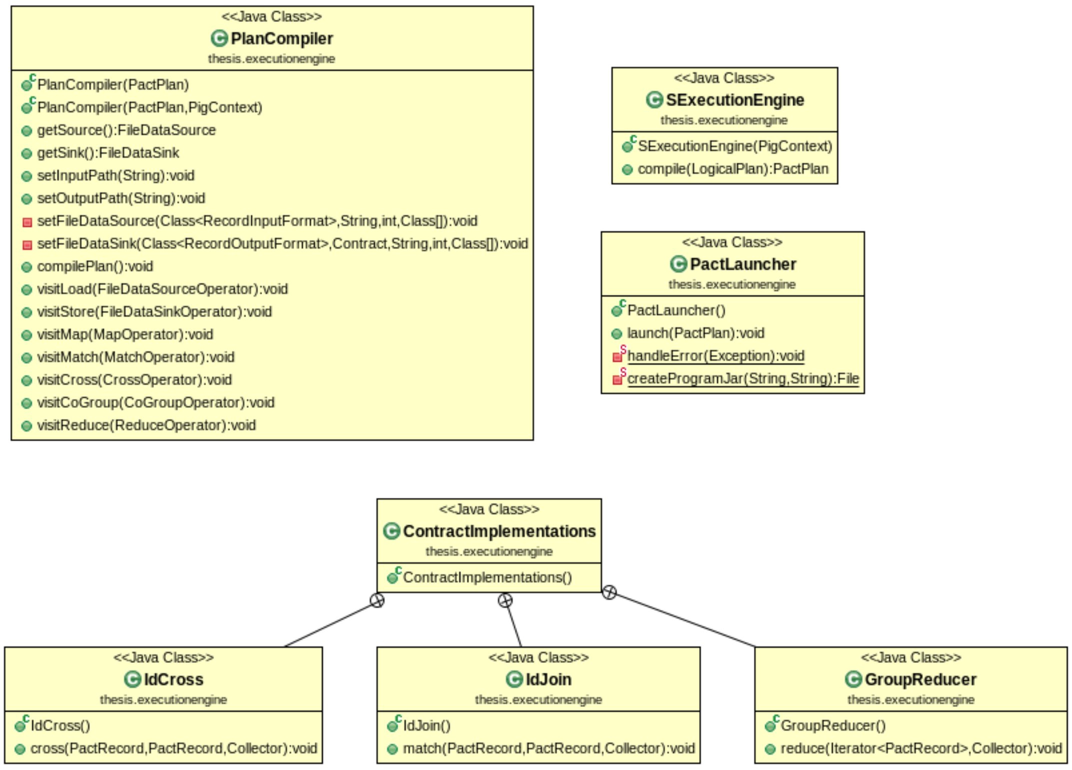
<<Java Class>>
Result
thesis.pactLayer
Result()
Result(byte,Object)
<<Java Class>>
LogicalToPactCompilerException
thesis.pactLayer
LogicalToPactCompilerException(String,int)
<<Java Class>>
PactPlan
thesis.pactLayer
PactPlan()
attachInput(Tuple):void
detachInput():void
explain(OutputStream):void
explain(OutputStream,boolean):void
connect(PactOperator,PactOperator):void
remove(PactOperator):void
replace(PactOperator,PactOperator):void
isEmpty():boolean
toString():String
<<Java Class>>
LogicalToPactCompiler
thesis.pactLayer
setPigContext(PigContext):void
LogicalToPactCompiler(OperatorPlan)
getLogToPactMap():Map<Operator,PactOperator>
getPactPlan():PactPlan
visit(LOLoad):void
visit(LOStore):void
visit(LOJoin):void
visit(LOCross):void
visit(LOFilter):void
visit(LOForEach):void
visit(LOCogroup):void
translateExpressionPlans(LogicalRelationalOperator,List<LogicalExpressionPlan>):List<PactPlan>
translateSoftLinks(Operator):void
<<Java Class>>
ExpToPactTranslationVisitor
thesis.pactLayer
ExpToPactTranslationVisitor(OperatorPlan,LogicalRelationalOperator,PactPlan,Map<Operator,PactOperator>)
ExpToPactTranslationVisitor(OperatorPlan,PlanWalker,LogicalRelationalOperator,PactPlan,Map<Operator,PactOperator>)
setPigContext(PigContext):void
getPhysicalPlan():PactPlan
visit(AndExpression):void
attachBinaryComparisonOperator(BinaryExpression,BinaryComparisonOperator):void

Figure 5.10: UML diagram for package thesis.pactLayer

of our prototype. The main classes of this package are depicted in the UML diagram of Figure 5.10.

- **thesis.pactLayer.plans**
  This package contains the *PactPlanVisitor* class. It is a visitor for navigating and operating on a plan of PactOperators objects. This class only contains the logic to traverse the plan, while the actual implementation of visiting individual nodes is left to the *PlanCompiler* class.

- **thesis.executionengine**
  This package contains the classes that enable Pig to work with Stratosphere. First of all, it contains the second system compiler, the *PlanCompiler* class, which is a subclass of the *PactPlanVisitor* class and responsible for translating a PACT plan into a set of Input Contracts. It generates the Stratosphere plan and implements the specific visit methods for all PactOperator objects supported. The *ContractImplementations* class contains static implementations of the Stratosphere stubs supported. The *SExecutionEngine* class is the Stratosphere execution engine class and contains the public method that a user calls to compile the Logical Plan into a PACT Plan. Finally, the class *PactLauncher* of this package is the class used to create and launch a Stratosphere program. It contains the functionality to create the necessary jar, the *PactProgram* object



<<Java Class>>
PlanCompiler
thesis.executionengine
PlanCompiler(PactPlan)
PlanCompiler(PactPlan,PigContext)
getSource():FileDataSource
getSink():FileDataSink
setInputPath(String):void
setOutputPath(String):void
setFileDataSource(Class<RecordInputFormat>,String,int,Class[]):void
setFileDataSink(Class<RecordOutputFormat>,Contract,String,int,Class[]):void
compilePlan():void
visitLoad(FileDataSourceOperator):void
visitStore(FileDataSinkOperator):void
visitMap(MapOperator):void
visitMatch(MatchOperator):void
visitCross(CrossOperator):void
visitCoGroup(CoGroupOperator):void
visitReduce(ReduceOperator):void
<<Java Class>>
SExecutionEngine
thesis.executionengine
SExecutionEngine(PigContext)
compile(LogicalPlan):PactPlan
<<Java Class>>
PactLauncher
thesis.executionengine
PactLauncher()
launch(PactPlan):void
handleError(Exception):void
createProgramJar(String,String):File
<<Java Class>>
ContractImplementations
thesis.executionengine
ContractImplementations()
<<Java Class>>
IdCross
thesis.executionengine
IdCross()
cross(PactRecord,PactRecord,Collector):void
<<Java Class>>
IdJoin
thesis.executionengine
IdJoin()
match(PactRecord,PactRecord,Collector):void
<<Java Class>>
GroupReducer
thesis.executionengine
GroupReducer()
reduce(Iterator<PactRecord>,Collector):void

Figure 5.11: UML diagram for package thesis.executionengine

and the *Client* to execute it. It exports the launch method which is called from the *org.apache.PigServer* class. Find the main classes of this package and their relations in more detail in Figure 5.11.

- **thesis.io**
  This package contains classes that are necessary in order to allow Pig to exchange data with Stratosphere. The *SLoadFunc* class is essentially a class for Stratosphere based on *org.apache.pig.LoadFunc* and it provides the interface to load data into Pig. It can read from an HDFS file or other source. Similar to *org.apache.pig.LoadFunc*, *SLoadFunc* is tightly coupled to Stratosphere's eu.stratosphere.pact.common.io.RecordInputFormat. It basically sits atop a *RecordInputFormat* and translates from *PactRecords* to Pig's tuples. It contains the basic features needed by the majority of load functions. The corresponding functionality for storing data is included inside the *SStoreFuncInterface* interface.

Our aim was to make as few changes to the Pig codebase as possible. The necessary changes made in the Pig codebase in order to support the limited functionality of our prototype can be found in detail in Appendix Changes in Pig. Also, find in Appendix Dependencies with Stratosphere all the dependencies with the Stratosphere project that were introduced because of the integration.

5.4 Discussion

Evaluating integration alternatives has been a very useful and interesting experience. We gained a deep understanding of the Pig internals, as well as familiarity with the codebase and the programming style used in the project. According to our judgment, the most suitable point for integration of Pig with an alternative execution engine proved to be the Logical Plan level.
For the implementation, we followed a strategy similar to the one already used by Pig, i.e. we created a two-stage compiler. During the first stage, an intermediate Physical Plan, which we called PACT Plan, is created. This Plan os PACT operators is then translated into the final graph using the actual Input Contracts, during the second stage. A lot of functionality of Pig was re-used, such as graph implementation and node visitor classes and methods.
Finally, it is important to remark that even if the implementation of our prototype is not complete, due to time and resource constraints, the completeness of our proposal is guaranteed. The PACT programming model is a generalization of the MapReduce programming model that also offers bigger flexibility in defining dataflows. Since every Pig Latin program and Logical Plan can be translated into a MapReduce Plan, it is obvious that is can also be translated into a PACT plan. Moreover, the translation process into a PACT Plan is more straight-forward and it is also highly benefited by the fact that PACTs can be combined in order to form arbitrary complex dataflow graphs (12).

Chapter 6

Evaluation

6.1 Cluster Configuration

For the evaluation of the systems, we had access to a 7-node Ubuntu Linux cluster at SICS (Swedish Institute of Computer Science). Each cluster node had 30GB of RAM and 12 six-core AMD Opteron(tm) processors. The nodes had Ubuntu 11.04 - Linux Kernel 2.6.38-12-server as Operating System and Java(TM) SE Runtime Environment version 1.6.0. We deployed Hadoop version 0.20.2, Stratosphere development version 0.2 and Pig version 0.9, as well as our prototype on the cluster. The following common configuration was used for Hadoop and Stratosphere:

- 5GB of RAM maximum memory per node
- Maximum 7 concurrent tasks per node
- 64MB HDFS block size

6.2 Test Data

For our experiments, we generated random data according to the Pig Performance page on the Pig wiki (7). Using a simple Python script, we created two datasets of tab delimited data with identical schema: name - string, age - integer, gpa - float. The first dataset consisted of 200 million rows (4384624709 bytes) and was used for all our experiments. The second set contained 10 thousand rows (219190 bytes) and was used as the second set in the join and cross experiments. Since the cluster was shared by multiple users, we ran each experiment multiple times, on different dates and times of day and calculated the average execution time and standard deviation of each measurement. The results presented here are based on measurements with standard deviation no larger that 0.1.

6.3 Test Cases

For the evaluation of our prototype, we developed six simple scripts that test basic functionality. The Pig Latin scripts used are shown below:

- Load and Store

  ```
  A = LOAD '200M.in';
  STORE A INTO 'load_store.out';
  ```

- Filter that removes 10% of data

  ```
  A = LOAD '200M.in' AS (name, age, gpo);
  B = FILTER A BY age > 90;
  STORE B INTO 'filter10.out';
  ```

- Filter that removes 90% of data

  ```
  A = LOAD '200M.in' AS (name, age, gpo);
  B = FILTER A BY age < 90;
  STORE B INTO 'filter90.out';
  ```

- Group

  ```
  A = LOAD '200M.in' AS (name, age, gpo);
  B = GROUP A BY age;
  STORE B INTO 'group.out';
  ```

- Join

  ```
  A = LOAD '200M.in' AS (name, age, gpo);
  B = LOAD '10K.in' AS (sname, sage, sgpo);
  C = JOIN A BY age, B BY sage;
  STORE C INTO 'join.out';
  ```

- Cross

  ```
  A = LOAD '200M.in' AS (name, age, gpo);
  B = LOAD '10K.in' AS (sname, sage, sgpo);
  C = CROSS A , B;
  STORE C INTO 'cross.out';
  ```

Apart from the Pig Latin scripts, we also developed equivalent native MapReduce and PACT applications for each one of the test cases. In the case of MapReduce, we used the Reducer-side algorithm for implementing the Join and Pig's artificial keys algorithm for the Cross. Both algorithms are described in 3.3.3. Note that we did not make use of the distributed cache facility, since we wanted the native MapReduce program to be as close as possible to the automatically generated job produced by Pig. Pig's default join strategy is identical to the one we used. If the user wants to make use of the distributed cache in Pig, they need to declare it explicitly, by using the 'REPLICATED' keyword in the script. Pig (or Hadoop) has no internal mechanism on detecting if one of the inputs is small enough so that it can fit in memory. On the other hand, when using the Match Input Contract in Stratosphere, the PACT compiler will evaluate all possible execution strategies and will choose the most effective one. Our results for the Join execution confirm this advantage of the PACT compiler over Pig/Hadoop.

6.4 Results

In this section we present the results of our experimental evaluation. All following diagrams illustrate comparisons among the four types of applications we developed for the test cases of 6.3, i.e. Pig on Hadoop, Native MapReduce, Pig on Stratosphere and Native PACT applications. In order to make the comparison more clear, all diagrams have been designed to have a darker area background color below the barrier of one.

6.4.1 Pig on Hadoop vs. Native MapReduce

We consider native MapReduce applications execution time as the baseline for our experiments, in order to reveal the value offered by Pig's abstraction. The same approach is taken by the Pig paper authors in their evaluation section (22). As a starting point, we compared the execution time of the Pig on Hadoop scripts with the one of native MapReduce programs. Our results are shown in Figure 6.1. The ratio of the execution time of Pig over the native MapReduce execution is depicted on the vertical axis, in logarithmic scale, while there is one bar for each of the test cases described in 6.3. The results confirm the ones already published by the Pig community. That is, Pig is around 1,2 to 1,6 times slower than a native MapReduce application. This overhead includes setup, compiling, data conversion and plan optimization time.

6.4.2 Pig on Stratosphere vs. Native PACTs

In order to have a similar comparison and evaluation of our prototype's compiler, we drew on Figure 6.2 the ratio of the execution time of Pig on Stratosphere scripts over native PACT programs. The vertical axis is in logarithmic scale. We observe in this diagram that the overhead imposed by our system on top of the native PACT applications is higher compared to the one of Figure 6.1 for the first two scripts. For the rest of the scripts, it appears that the overhead is significantly lower and

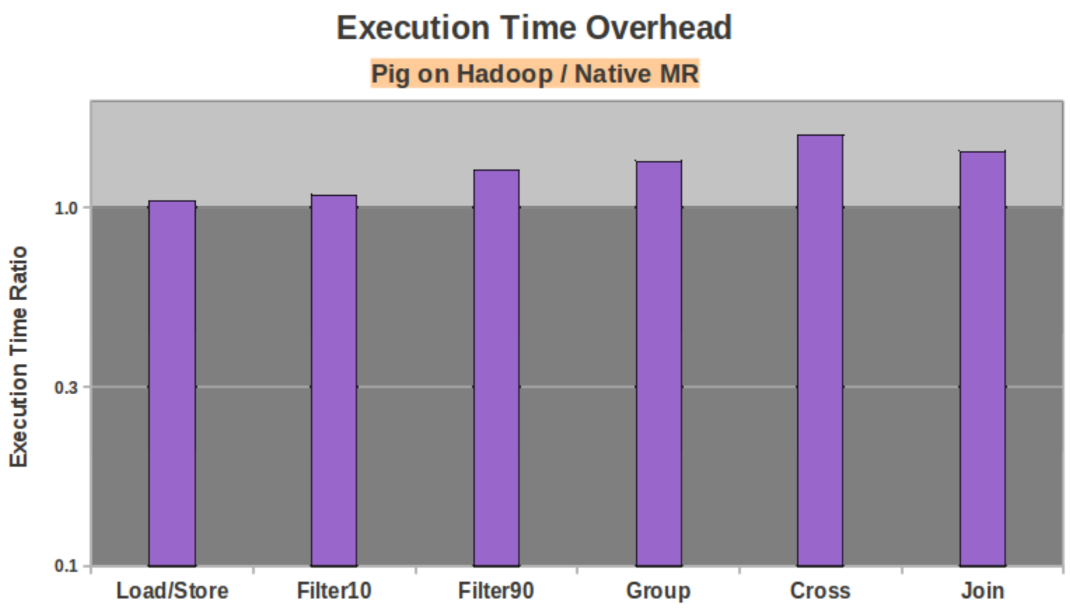


Figure 6.1: Pig on Hadoop vs. Native MapReduce implementations

smaller than 1,5 in all cases. One could argue that this result suggests that our compiler is more efficient than Pig's. However, one should keep in mind that our system only supports a small subset of Pig's features and the translation process has been significantly simplified. We believe that the overhead would be much higher if our system would have been implemented with full functionality. On the other hand, the result is still promising and shows that there is a margin big enough for creating an efficient and complete Pig on Stratosphere compiler.

6.4.3 Pig on Hadoop vs. Pig on Stratosphere

The next comparison, in Figure 6.3, shows how many times faster or slower each of the scripts is executed in our prototype, to the of Pig on Hadoop execution time. The upper half corresponds to slower execution, while the lower half depicts advantage of Pig on Stratosphere. Pig on Stratosphere performs worse for the load/store and the filtering scripts, while it demonstrates an outstanding performance gain over Pig on Hadoop for the other three scripts. It runs 1,4 times faster in the case of grouping and 2,5 times faster in the case of cross. For the join script, it outperforms Pig on Hadoop almost 4,5 times. This outcome is mainly due to two facts. First, it seems from our measurements that native PACT programs perform worse than native MapReduce programs in the case of the first three scripts. Consequently, Pig on Stratosphere is also expected to perform worse than Pig on Hadoop. Second, as we already mentioned, we have implemented join using the default Reducer-side algorithm that does not exploit the distributed cache. On the other hand, just by

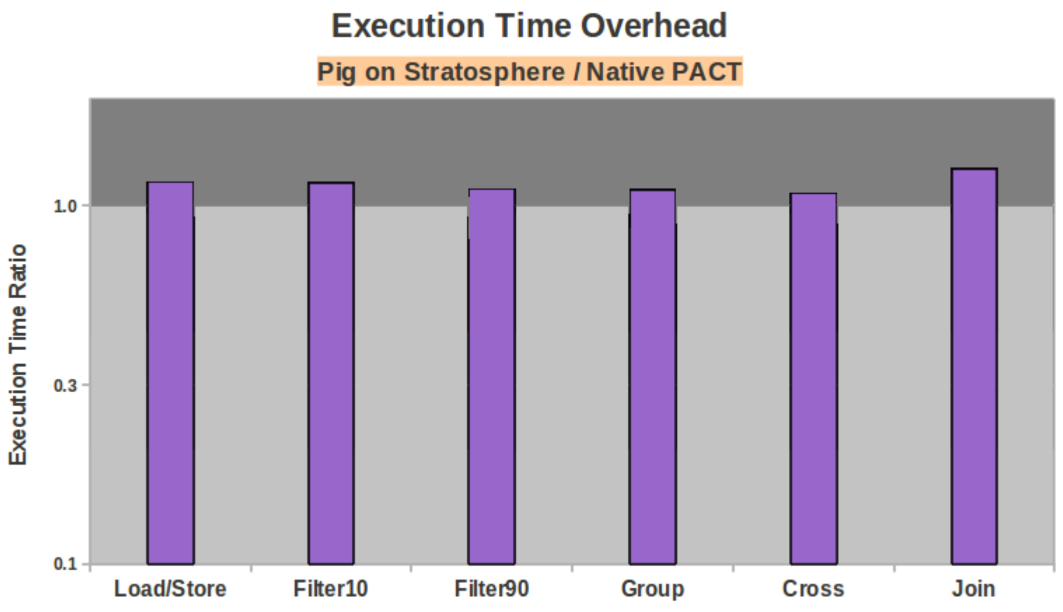


Figure 6.2: Pig on Stratosphere vs. Native PACT implementations

using the Match input contract of Stratosphere, the PACT compiler realizes that the second input can fit into memory and chooses the broadcasting execution strategy. This strategy has a very big advantage over the partitioning and shuffling one. As a result, Pig on Stratosphere benefits from this fact for free. Of course, it would be interesting to investigate a case where none of the two inputs fit into memory, but, unfortunately, we have no experimental results for this case in the present thesis. However, it is our intention to research this issue further in our future work.

6.4.4 Pig on Stratosphere vs. Native MapReduce

The last comparison is that of our Pig on Stratosphere implementation over the baseline native MapReduce, shown in Figure 6.4. The result is not surprising and is a direct aftermath of the combination of the previous results. Pig on Stratosphere performs worse than Pig on Hadoop for the first three test cases and since Pig on Hadoop is slower than native MapReduce, this implies that also Pig on Stratosphere will be slower than native MapReduce for these cases. For the group, cross and join Pig on Stratosphere is faster than Pig on Hadoop. The performance gain in these cases is greater than the performance loss of Pig on Hadoop over native MapReduce, which results in Pig on Stratosphere outperforming native MapReduce for these cases. The main reason behind this outcome is native Stratosphere's advantage over native MapReduce for this type of scripts.

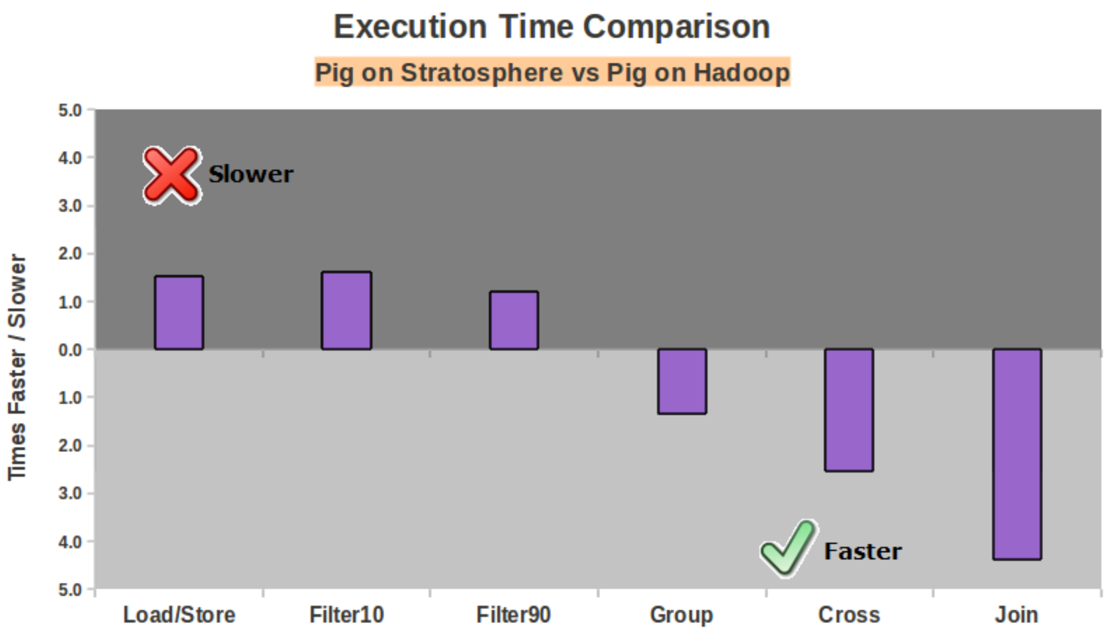


Figure 6.3: Pig on Hadoop vs. Pig on Stratosphere

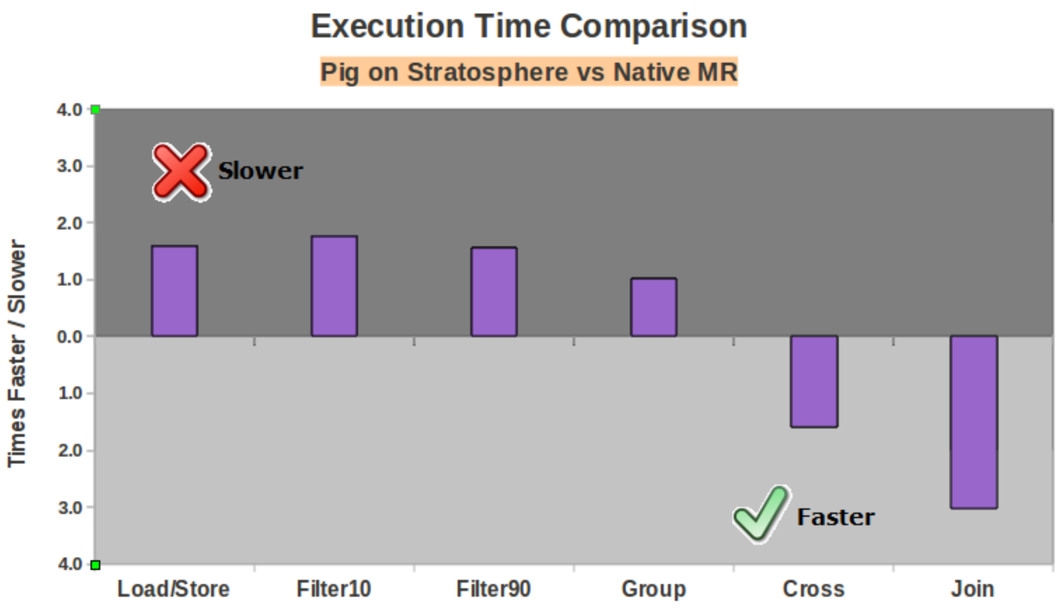


Figure 6.4: Pig on Stratosphere vs. Native MapReduce implementations

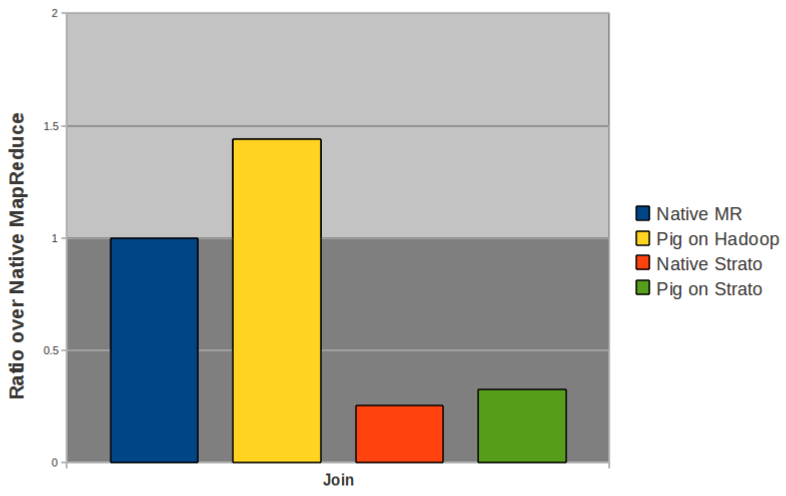


Figure 6.5: Join execution times compared to native MapReduce implementation

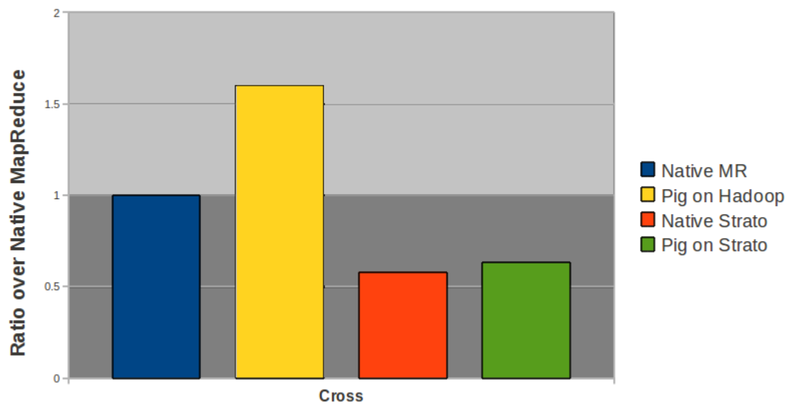


Figure 6.6: Cross execution times compared to native MapReduce implementation

6.4.5 Global Comparison

In order to provide a global comparison, we chose the join and cross scripts and drew Figures 6.5 and 6.6. In these diagrams, we provide a visual representation of the ratios of all implementations over the baseline native MapReduce, which is depicted on the first bar (blue). The second bar (yellow) shows the ratio of the Pig on Hadoop execution time, the third (orange) shows the ratio of the native PACT applications and the last bar (green) shows the ratio of Pig on Stratosphere. In both these diagrams, our argument on native PACT being more efficient than native MapReduce becomes obvious. It is clear that the margin is large enough for covering the performance loss that a complete abstraction on top of Stratosphere would cause.

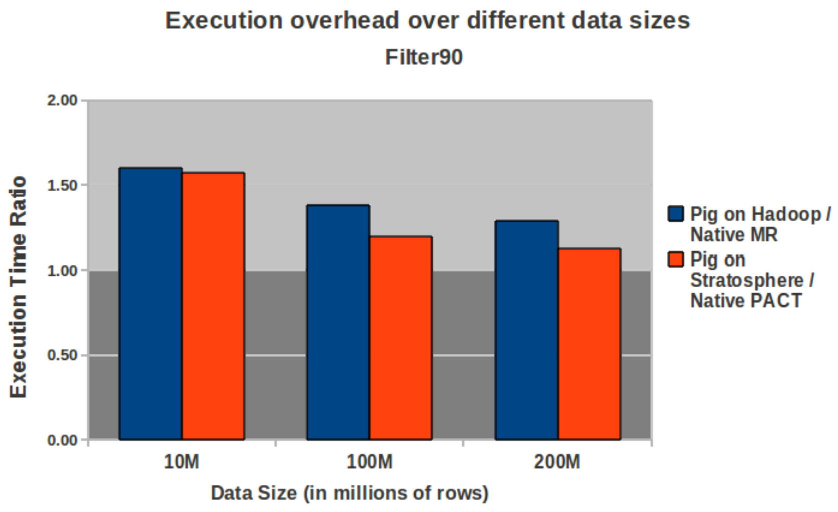


Figure 6.7: Execution overhead for different dataset sizes - Filter90

6.4.6 Dataset Size Impact

The overhead introduced by the abstraction in either of the cases (Pig on Hadoop and Pig on Stratosphere) is a function of several parameters, which cannot be easily defined, especially if evaluation is performed on one single dataset. This overhead includes setup time, compilation, plan optimization and transformation time, Pig to execution engine communication time (e.g. statistics and job monitoring), but also datatype conversion time. Some of these parameters are highly dependent on the size of the input data. In order to have a better idea on how the overhead changes depending on the dataset size, we ran two of the scripts with varying input size. The results are shown in Figure 6.7 and 6.8, for the script of filtering 90% of the input data and the grouping script respectively. The datasets chosen are 10 million rows, 100 million rows and 200 million rows, while the vertical axis corresponds to execution time ratio.

As expected, it seems that the setup and compilation time has a heavier influence for smaller datasets. However, profound investigation is necessary in order to more accurately determine the parameter characteristics.

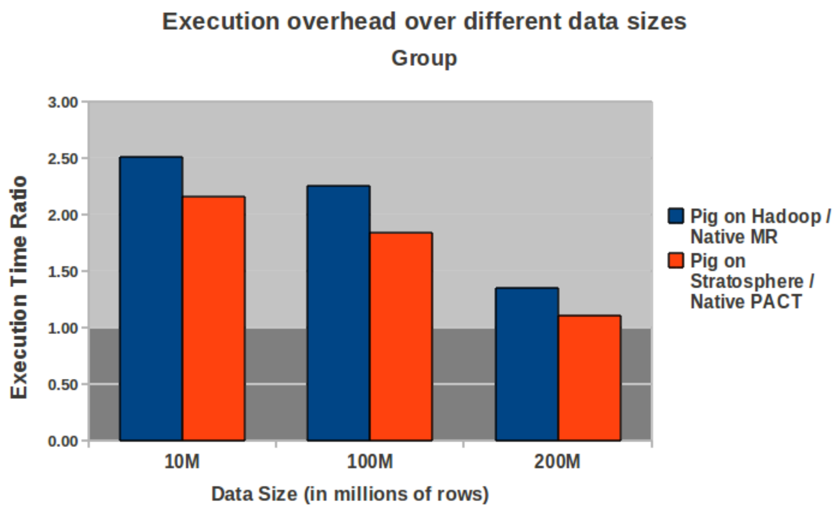


Figure 6.8: Execution overhead for different dataset sizes - Group

Chapter 7
Conclusions

Big data analytics and management is one of the biggest trends and challenges of our era and the requirement for efficient tools is becoming more and more urgent. Existing programming models like MapReduce and PACT have been a great contribution. However, in order to fully exploit the possibilities provided by the increasing amounts of data in business and scientific applications, data analysis should become accessible to non-experts, users that are used to work with SQL or higher-level languages. Systems like Pig have made this possible and improving their performance is of great importance.
In this thesis, we studied Pig's architecture and examined the possibility of integrating it with Stratosphere. We concluded that even though Pig is tightly coupled to the Hadoop execution engine, integration is possible. According to our judgment, the Pig stack can be separated from the Hadoop execution engine, no lower than the Logical Plan level. As a consequence, integration with Stratosphere, or any other execution engine, requires re-implementation of the rest of the stack.
In the case of Stratosphere, the translation of Pig's Logical Plan into a PACT plan turned out to be much simpler than the translation to MapReduce jobs. The reason for this is that Stratosphere is a more general framework than MapReduce, offers higher flexibility and provides an easy way of expressing common relational operations through Input Contracts. Moreover, the first results of the performance evaluation of our prototype reveal Stratosphere's advantage over MapReduce for relational operations and allow us to be optimistic that Pig on Stratosphere could perform even better than native MapReduce.

7.1 Open Issues

Several issues remain unexplored due the limited scope of this project and time restrictions. First of all, not all Pig operators have been implemented in our prototype. More specifically, only the narrow set of operators, on which we carried out our experiments, is supported. Thus, more advanced queries containing nested operators, SPLIT, DISTINCT, LIMIT etc. cannot be used with our prototype. Also, all operators currently support maximum two inputs. However, this could be addressed by either extending Stratosphere in adding Multi-Input Contracts, or by creating a pipeline or even binary trees of consequtive Dual-Input Contracts. Secondly, we acknowledge that a more extensive evaluation is needed in order to draw safe conclusions. Experiments should be performed on varying sizes of datasets, in order to define the setup and overhead constants. Also, scalability should be addressed, by varying the number of available instances for computation.

7.2 Future Work

We certainly believe that creating a system that fully supports Pig Latin and generates Stratosphere jobs is not the limit of this research. Several optimizations can now be added to Pig because of the underlying Nephele execution engine. Pig Latin could be extended to include keywords that would correspond to Output Contracts or PACT's compiler hints. Moreover, we believe that it would be of great interest to extend the PigMix (8) benchmark suite with native PACT applications and evaluate our system with it, offering a more fair comparison to the existing Pig system. Based on the results of a wider and more extensive evaluation, we could then identify the type of applications for which Pig on Hadoop performs better and the type of applications for which Pig on Stratosphere has an advantage. With this information, it would be possible to build a "smarter" Pig compiler that could ideally choose the best execution engine depending on the application type.

Appendix

Changes in Pig

- **org.apache.pig.Main** : replaced all the occurrences of the class org.apache.pig.LoadFunc to thesis.io.SLoadFunc
- **org.apache.pig.PigServer** : replaced all the occurrences of the class org.apache.pig.builtin.PigStorage, changed the method lanchPlan() to create a thesis.executionengine.PactLauncher object and call its launch() method, changed the compilePp() to create a thesis.executionengine.SExecutionEngine and return a thesis.pactLayer.PactPlan.
- **org.apache.pig.impl.io.FileSpec** : replaced all the occurrences of the class org.apache.pig.builtin.PigStorage to thesis.io.builtin.SPigStorage
- **org.apache.pig.newplan.logical.relational.LOLoad** : replaced all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc
- **org.apache.pig.newplan.logical.relational.LOStore** : replaced all occurrences of org.apache.pig.StoreFuncInterface to thesis.io.SStoreFuncInterface
- **org.apache.pig.newplan.logical.rules.ColumnPruneVisitor** : replaced all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc
- **org.apache.pig.newplan.logical.rules.InputOutputFileValidator** : replaced all occurrences of org.apache.pig.StoreFuncInterface to thesis.io.SStoreFuncInterface
- **org.apache.pig.newplan.logical.rules.PartitionFilterOptimizer** : replaced all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc
- **org.apache.pig.parser.FunctionType** : replaced all occurrences of org.apache.pig.StoreFuncInterface to thesis.io.SStoreFuncInterface and all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc
- **org.apache.pig.parser.LogicalPlanBuilder** : replaced all occurrences of org.apache.pig.StoreFuncInterface to thesis.io.SStoreFuncInterface, all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc and all occurrences of org.apache.pig.builtin.PigStorage to thesis.io.builtin.SPigStorage

- **org.apache.pig.parser.QueryParserUtils** : replaced all occurrences of org.apache.pig.StoreFuncInterface to thesis.io.SStoreFuncInterface, all occurrences of org.apache.pig.LoadFunc to thesis.io.SLoadFunc and all occurrences of org.apache.pig.builtin.PigStorage to thesis.io.builtin.SPigStorage

Dependencies with Stratosphere

The following Stratosphere packages had to be exported as jar and included in Pig's build path:

- eu.stratosphere.nephele.io
- eu.stratosphere.nephele.ipc
- eu.stratosphere.nephele.jobgraph
- eu.stratosphere.nephele.event
- eu.stratosphere.nephele.instance
- eu.stratosphere.nephele.managementgraph
- eu.stratosphere.nephele.protocols
- eu.stratosphere.nephele.topology
- eu.stratosphere.nephele.util
- eu.stratosphere.nephele.net
- eu.stratosphere.nephele.template
- eu.stratosphere.nephele.types
- eu.stratosphere.pact.client.nephele.api
- eu.stratosphere.pact.common.contract
- eu.stratosphere.pact.common.io
- eu.stratosphere.pact.common.io.stratistics
- eu.stratosphere.pact.common.plan
- eu.stratosphere.pact.common.stubs
- eu.stratosphere.pact.common.types
- eu.stratosphere.pact.common.type.base
- eu.stratosphere.pact.compiler
- eu.stratosphere.pact.runtime
- eu.stratosphere.pact.common.type.base.parser

Bibliography

[1] *Cascading*, `http://www.cascading.org/`, [Online; Last accessed 2012].

[2] *Facebook*, `http://facebook.com`, [Online; Last accessed 2012].

[3] *Hadoop: Open-Source implementation of MapReduce*, `http://hadoop.apache.org`, [Online; Last accessed 2012].

[4] *Introduction to Apache Pig*, `http://www.cloudera.com/?resource=introduction-to-apache-pig`, [Online; Last accessed 2012].

[5] *LINQ*, `http://msdn.microsoft.com/library/bb397910.aspx`, [Online; Last accessed 2012].

[6] *Pig Functions Reference*, `http://pig.apache.org/docs/r0.10.0/func.html`, [Online; Last accessed 2012].

[7] *Pig Performance Wiki Page*, `http://wiki.apache.org/pig/PigPerformance`, [Online; Last accessed 2012].

[8] *PigMix*, `https://cwiki.apache.org/confluence/display/PIG/PigMix`, [Online; Last accessed 2012].

[9] *Scalding*, `https://dev.twitter.com/blog/scalding`, [Online; Last accessed 2012].

[10] *Stratosphere*, `http://www.stratosphere.eu/`, [Online; Last accessed 2012].

[11] *Twitter*, `http://twitter.com`, [Online; Last accessed 2012].

[12] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke, *Mapreduce and pact - comparing data parallel programming models.*, BTW (Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz, eds.), LNI, vol. 180, GI, 2011, pp. 25–44.

[13] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, *System r: relational approach to database management*, ACM Trans. Database Syst. **1** (1976), no. 2, 97–137.

[14] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke, *Nephele/pacts: a programming model and execution framework for web-scale analytical processing*, Proceedings of the 1st ACM symposium on Cloud computing (New York, NY, USA), SoCC '10, ACM, 2010, pp. 119–130.

[15] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl C. Kanne, Fatma Özcan, and Eugene J. Shekita, *Jaql: A Scripting Language for Large Scale Semistructured Data Analysis*, PVLDB **4** (2011), no. 12, 1272–1283.

[16] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum, *Flumejava: easy, efficient data-parallel pipelines*, Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), PLDI '10, ACM, 2010, pp. 363–375.

[17] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong, *Tenzing a sql implementation on the mapreduce framework*, 2011, pp. 1318–1327.

[18] Surajit Chaudhuri, *An overview of query optimization in relational systems*, Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (New York, NY, USA), PODS '98, ACM, 1998, pp. 34–43.

[19] Christoph Lawerentz Christoph Nagel Jakub Berezowski Martin Guether Matthias Ringwald Moritz Kaufmann Ngoc Tiep Vu Stefan Lobach Stephan Pieper Thomas Bodner Christian Wurtz, *Project jaql on the cloud*, Final report, TU Berlin, April 2011.

[20] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, Commun. ACM **51** (2008), no. 1, 107–113.

[21] Alan F. Gates, *Programming pig*, first ed., O'Reilly, 2011.

[22] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava, *Building a high-level dataflow system on top of map-reduce: the pig experience*, Proc. VLDB Endow. **2** (2009), no. 2, 1414–1425.

[23] Paulo Ricardo Motta Gomes and Zhenhua Xu, *Comparing distributed data processing paradigms: Mapreduce vs pact*, Final report, KTH, The Royal Institute of Technology, January 2012.

[24] G. Graefe, *Volcano: An extensible and parallel query evaluation system*, IEEE Trans. on Knowl. and Data Eng. **6** (1994), no. 1, 120–135.

[25] Goetz Graefe, *The cascades framework for query optimization*, IEEE Data Eng. Bull. **18** (1995), no. 3, 19–29.

[26] Goetz Graefe and David J. DeWitt, *The exodus optimizer generator*, Proceedings of the 1987 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD ’87, ACM, 1987, pp. 160–172.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks*, SIGOPS Oper. Syst. Rev. **41** (2007), no. 3, 59–72.

[28] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins, *Pig latin: a not-so-foreign language for data processing*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD ’08, ACM, 2008, pp. 1099–1110.

[29] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, *Interpreting the data: Parallel analysis with sawzall*, Sci. Program. **13** (2005), no. 4, 277–298.

[30] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, *The hadoop distributed file system*, Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (Washington, DC, USA), MSST ’10, IEEE Computer Society, 2010, pp. 1–10.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy, *Hive: a warehousing solution over a map-reduce framework*, Proc. VLDB Endow. **2** (2009), no. 2, 1626–1629.

[32] Daniel Warneke and Odej Kao, *Nephele: efficient parallel data processing in the cloud*, Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (New York, NY, USA), MTAGS ’09, ACM, 2009, pp. 8:1–8:10.

[33] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey, *Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language*, Proceedings of the 8th USENIX conference on Operating systems design and implementation (Berkeley, CA, USA), OSDI’08, USENIX Association, 2008, pp. 1–14.

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, 25 June 2012

. . . .

Vasiliki Kalavri

TRITA-ICT-EX-2012:146