PonIC: Using Stratosphere to Speed Up Pig Analytics

Vasiliki Kalavri[1], Vladimir Vlassov[1], and Per Brand[2]

[1] KTH Royal Institute of Technology
{kalavri,vladv}@kth.se
[2] Swedish Institute of Computer Science
Stockholm, Sweden
perbrand@sics.se

Abstract. Pig, a high-level dataflow system built on top of Hadoop MapReduce, has greatly facilitated the implementation of data-intensive applications. Pig successfully manages to conceal Hadoop's one input and two-stage inflexible pipeline limitations, by translating scripts into MapReduce jobs. However, these limitations are still present in the backend, often resulting in inefficient execution.

Stratosphere, a data-parallel computing framework consisting of PACT, an extension to the MapReduce programming model and the Nephele execution engine, overcomes several limitations of Hadoop MapReduce. In this paper, we argue that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness.

We have ported Pig on top of Stratosphere and we present a process for translating Pig Latin scripts into PACT programs. Our evaluation shows that Pig Latin scripts can execute on our prototype up to 8 times faster for a certain class of applications.

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, is in urgent need of data analytics systems, capable of scaling up to petabytes of data. Such systems need to efficiently analyze text, web data, log files and scientific data. Most of the recent approaches use massive parallelism and are deployed on large clusters of hundreds or even thousands of commodity hardware.

MapReduce [1], proposed by Google, is the most popular framework for large-data processing; its open-source implementation, Hadoop[1], is nowadays widely used. However, it has several limitations, including the limitation on the number of input datasets (only one input set) and the limitation on a structure of a program that must follow a static fixed pipeline pattern of the form split-map-shuffle-sort-reduce. This pipeline is suitable for simple applications, such as log-file analysis, but severely complicates the implementation of relational queries or

[1] http://hadoop.apache.org/

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 279–290, 2013.

graph algorithms. These limitations have led researchers to develop more general-purpose systems, inspired by MapReduce [2–6]. One of them is Stratosphere [6], which consists of a programming model, PACT (Parallelization Contracts), and the Nephele execution engine. The system is essentially a generalization of MapReduce and aims to overcome the limitations mentioned above.

Both models, MapReduce and PACT, require significant programming ability and in-depth understanding of the systems' architectures. Applications usually lead to complex branching dataflows which are low-level and inflexible. In order to save development time and make application code easier to maintain, several high-level languages have been proposed for these systems. Currently, high-level platforms on top of Hadoop include JAQL [7], Hive [8] and Pig [9]. Pig Latin, which is the language of the Pig platform [10], offers the simplicity and declarativeness of SQL, while maintaining the functionality of MapReduce. Pig compiles Pig Latin into MapReduce jobs which are executed in Hadoop. Pig hides Hadoop's one-input and two-stage dataflow limitations from the programmer and provides built-in functions for common operations, such as filtering, join and projection. It also directly benefits from Hadoop's 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. The translation of relational operators for the static pipeline of Hadoop produces an inefficient execution plan since data have to be materialized and replicated after every MapReduce step.

The goal of Pig was to make MapReduce accessible to non-experts and relieve the programmer from the burden of repeatedly coding standard operations, like joins. Another goal was to make Pig independent of any particular backend execution engine. However, Pig was developed on top of Hadoop, ended up solving specific Hadoop problems and became highly coupled with its execution engine. The Stratosphere data-parallel computing framework offers a superset of MapReduce functionality, while overcoming some of the major weaknesses of the MapReduce programming model. 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, the PACT programming model of Stratosphere supports multiple inputs.

In this paper, we present PonIC (Pig on Input Contracts), an integration of the of Pig System with Stratosphere. We have analyzed the internal structure of Pig and have designed a suitable integration strategy. In order to evaluate the benefits of the integration, we have developed a prototype implementation. The current prototype supports a subset of the most common Pig operations and it can be easily extended to support the complete set of Pig Latin statements. Thus, we show that it is possible to plug a different execution engine into the Pig system and we identify the parts of Pig that can be reused. With our Pig to PACT translation algorithm and our prototype, we show that Stratosphere has desirable properties that significantly simplify the plan generation. We have developed a set of basic scripts and their native MapReduce and PACT equivalents and we provide a comparison of PonIC with Pig, as well as the corresponding native programs. We observe that Stratosphere's relational operators are much

more efficient than their MapReduce equivalents. As a result, PonIC has a great advantage over Pig on Hadoop and often executes faster than native Hadoop MapReduce. The main contributions of this paper are as follows.

- Our integration is entirely transparent to Pig's end-users and existing Pig Latin applications can be executed on PonIC without any modification. The syntax and the semantics are completely unchanged.
- We show that Pig can be harnessed to alternative execution engines and present a way of integration.
- We identify the features of Pig that negatively impact execution time.
- We show that Pig can be integrated with Stratosphere and gain performance.
- We propose a complete translation process of Pig Logical Plans into Stratosphere Physical Plans and we present and evaluate PonIC.

The rest of this paper is structured as follows. In Section 2, we provide the necessary background on the Pig and Stratosphere systems. Section 3 discusses the restrictions that MapReduce poses on Pig's current implementation and presents our Pig-to-Stratosphere translation process. In Section 4, we discuss our prototype implementation in detail. Section 5 contains the evaluation of PonIC against Pig on Hadoop, native Hadoop MapReduce and native PACT Stratosphere. In Section 6, we comment on related work, while we provide our conclusions, open issues and vision for the future in Section 7.

2 Background

In this section, we provide the essential background. We briefly discuss the MapReduce programming model, the Pig system and the Stratosphere system.

2.1 The MapReduce Programming Model

MapReduce is a data-parallel programming model. Its architecture is inspired by functional programming and consists of two second-order functions, Map and Reduce, which form a static pipeline. Data are read from an underlying distributed file system and are transformed into key-value pairs, which are grouped into subsets and processed by user-defined functions in parallel. Data distribution, parallelization and communication are handled by the framework, while the user only has to write the first-order functions wrapped by the Map and Reduce functions. However, this abstraction 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. MapReduce was initially proposed by Google and its open-source implementation, Hadoop and HDFS [11] are nowadays widely used.

2.2 Pig

Pig consists of a declarative scripting language, Pig Latin, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. The Pig System takes a Pig Latin program as input and produces a series of MapReduce jobs to be executed on the Hadoop engine. Compilation happens in several steps. First, the parser transforms a Pig Latin script into a Logical Plan. Each Logical operator is compiled down to one or more Physical Operators. The Physical Plan is then passed to the compiler that transforms it into a DAG of MapReduce operators. MapReduce operators are topologically sorted and connected between them using a store-load combination, producing the MapReduce Plan as output. The generated jobs are finally submitted to Hadoop and monitored by Pig.

2.3 Stratosphere

Stratosphere is a parallel data-processing framework, which consists of a programming model, PACT (Parallelization Contracts), and an execution engine, Nephele, capable of executing dataflow graphs in parallel. Nephele is an execution engine designed to execute DAG-based data flow programs. It manages task scheduling and setting up communication channels between nodes. Moreover, it supports dynamic allocation of resources during execution and fault-tolerance mechanisms. The PACT programming model is a generalization of the MapReduce programming model. 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. The following three more Contracts are defined in PACT:

- 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.
- 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.
- The *CoGroup* Contract creates independent subsets by combining all pairs that share the same key.

3 Plan Compilation

As explained in the previous section, a Pig Latin script is parsed and transformed into a graph of logical operators, each corresponding to one command. This graph, the *Logical Plan*, is then translated into a *Physical Plan*, a graph of physical operators, which defines how the logical operations will be executed. Multiple strategies can be used to map logical operators to physical ones and it's the system's compiler job to choose a strategy, depending on the underlying execution engine's capabilities, dataset characteristics, hints provided by the developer, etc. The translation process in Pig is briefly explained next.

3.1 Plan Compilation in Pig

Pig's compiler translates logical to physical operators, with the additional restriction that each physical operator needs to be expressed in terms of MapReduce steps or parts thereof. The compiler keeps track of the current phase during translation and knows if it is a map or a reduce step. For each operator, it checks if it can be merged into the current phase. If communication is required, the current phase is finalized and a new phase is started in order to compile the operator. If the current phase is a map, a reduce phase will be initiated; otherwise, a new MapReduce job needs to be created and store-load combination is required to chain the jobs. We explain the translation process using an example from a slightly modified query of the PigMix benchmark[2] shown below:

Example Query 1

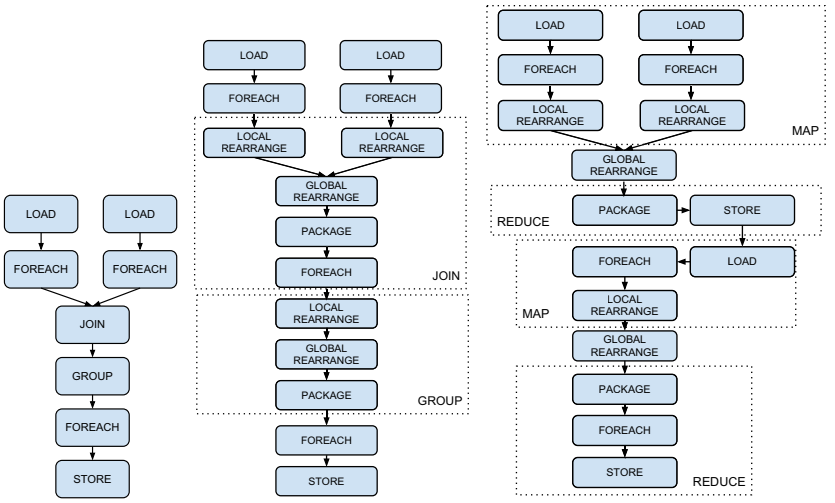
```
A = load 'page_views' as (user, timestamp, revenue);
B = foreach A generate user, revenue;
alpha = load 'users' as (name, phone, address, city);
beta = foreach alpha generate name;
C = join beta by name, B by user;
D = group C by $0;
E = foreach D generate group, SUM(C.revenue);
store E into 'out';
```

The simple Example Query 1 loads two datasets, performs a join on a common attribute to find the set of users who have visited some webpages, groups the resulting dataset and generates the estimated revenue for each user. Figure 1(a) shows the simplified Logical Plan for the above script, whereas Figure 1(b) shows the generated Physical Plan. Note that the join operator is replaced by four new operators and the group operator is translated into three physical operators similarly. The Physical Plan is then translated into a MapReduce Plan, as shown in Figure 1(c). First, a map phase is created and as the Physical Plan is traversed, operators are added to it. When the global rearrange operator is reached, shuffling is required, therefore the map phase is finalized and a reduce phase is initiated. When a new MapReduce job is created, a store-load pair is added in between to set the output of the first as the input of the second.

Our example shows that even for a small script, generated plans can be long and cumbersome. If the generated Logical Plan does not fit well the MapReduce static pipeline, performance might degrade. Adding store-load combinations and materialization of results in between jobs is also a source of inefficiency.

In contrast to MapReduce, using Stratosphere as the backend for Pig significantly simplifies the translation process. Input Contracts can be greatly exploited to generate shorter and more efficient plans, without any extra effort from the programmer. We present our translation algorithm next.

[2] `http://cwiki.apache.org/PIG/pigmix.html`



(a) Logical Plan (b) Physical Plan (c) MapReduce Plan

Fig. 1. Pig Plans for Example Query 1

3.2 Pig to PACT Plan Translation

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 the most common and useful ones and we describe here how they are be translated into PACT operators. A more detailed description of the translation process we followed can be found in [12].

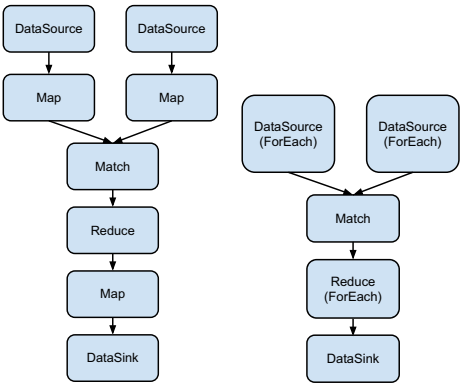
Input/Output. Pig provides the LOAD and the STORE commands for data input and output. These two logical operators can be mapped directly to the *GenericDataSource* and the *GenericDataSink* Input Contracts of Stratosphere. In our implementation, we only support input and output from and to files, so we have based our implementation on the more appropriate Contracts, *FileDataSource* and *FileDataSink*. The generic Contracts can be easily extended to support other kinds of input and output sources.

Relational Operators. PACTs support most of the common relational operations. The FILTER and FOREACH statements correspond to a Map Contract. 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. 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. Pig realizes the ORDER BY operation by creating two MapReduce jobs. With PACTs, the same functionality can be offered in a much simpler way using the *GenericDataSink* Contract.

Advanced Operators. From the set of the advanced Pig operators, we choose CROSS and UNION. The CROSS operator can be directly mapped to the *Cross* Input Contract, while the *Map* Input Contract can be used to realize UNION. The Map Contract (Stratosphere 0.2) offers a method, which provides the functionality we need to implement UNION.

Our translation consists of two stages. At the first stage the Logical Plan is translated into a plan of PACT operators. This PACT Plan is the equivalent of Pig's Physical Plan. The second stage translates the PACT Plan into actual Input Contracts and submits the PACT Plan to the Nephele execution engine.

The Plan generation for the Example Query 1 is shown in Figure 2(a). There is an one-to-one mapping of logical operators to PACT operators and consequently Input Contracts, which makes the graph and the translation process much simpler. The resulting graph can be further optimized, by merging filter and foreach operators into the preceding Contracts, as shown in Figure 2(b).



(a) Initial Plan (b) Optimized Plan

Fig. 2. PACT Plans for Example Query 1

3.3 Discussion

Even though we have considered only a subset of Pig operators, it is important to stress that the completeness of our proposal is guaranteed. The PACT programming model is a generalization of the MapReduce programming model. Since every Pig Latin program and Logical Plan can be translated into a MapReduce Plan, it can therefore also be translated into a PACT Plan.

Using Stratosphere and Input Contracts as the backend results into a more straightforward translation process. The one-to-one Pig-to-PACT mapping requires less communication, due to less shuffling. Data is pipelined between Input Contracts, eliminating the need for frequent materialization. Also, the execution plan benefits from optimizations of the Logical Plan by Pig's Logical Plan optimizer and of the PACT Plan by Stratosphere's optimizer[3].

[3] `http://stratosphere.eu/wiki/doku.php/wiki:pactcompiler`

4 Implementation

PonIC has been implemented as an extension to the Pig system and reuses Pig functionality where possible. Pig classes or wrappers are used in order to make them compatible with the new features. The source code is publicly available[4].

We have identified the parts of the Pig software stack that are not tightly coupled to the Hadoop execution engine, namely the parser and the Logical Plan layer. The underlying layers have been replaced with our compilation layer that tranforms the Logical Plan into a Stratosphere execution plan.

Pig's Logical Plan is traversed in a depth-first fashion. The traversal starts from the plan's roots and a *visit()* method is responsible for recognizing the operator type and creating the appropriate PACT operator, according to the mappings of Table 1. 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 predecessors. This way, a graph of PACT operators is gradually constructed. When the PACT Plan has been created, it is submitted to Nephele for execution. Table 1 summarizes the Pig to PACT translation mappings for the subset of the Pig operators considered in this study.

Table 1. Pig to PACT operators mapping (for the chosen subset of Pig operators)

Pig Operator	Input Contract
LOAD	FileDataSource
STORE	FileDataSink
GROUP	Reduce
INNER JOIN	Match
OUTER JOIN / COGROUP	CoGroup
UNION	Map
FILTER / FOREACH	Map
ORDER	FileDataSink

The most significant extensions made to the Pig codebase are:

- An additional execution mode to allow starting Pig in Stratosphere execution mode with the command `pig -x strato`.
- An extension of Pig's `HExecutionEngine` class as an engine for Stratosphere.
- A re-implementation os the relational and expression operators to support the new APIs.
- A `LogToPactTranslationVisitor` class, based on Pig's `LogToPhyTranslationVisitor` class, as the first-level compiler.
- A package of PACT operators, based on Pig's physical operators.
- A `PactCompiler` class, as the second-level compiler.
- Stratosphere-specific load and store functions.
- A `contractsLayer` and a stubsLayer packages, which contain wrapper classes of Stratosphere's Input Contracts and Stub classes.

[4] `http://github.com/PonIC/PonIC`

5 Evaluation

We conducted our experiments on an OpenStack cluster, using 10 ubuntu Virtual Machines (VMs), each having 4 VCPUs, 90GB of disk space and 8GB of RAM. We deployed Hadoop version 1.0.0, Pig version 0.10.0 and Stratosphere version 0.2. Hadoop's NameNode and JobTracker, as well as Stratosphere's JobManager run on a dedicated VM, while the remaining 9 VMs serve as slave nodes. Default parameters were used for HDFS block size and replication factor.

We used the PigMix data generator to create a `page_views` dataset of 10 million rows (approximately 15GB) and the corresponding `users` table. We developed five scripts for evaluation, namely a Load/Store operation, a Filter script which filters out 50% of the input, a Group operation, a Join of the `page_views` and the `users` dataset and a Mixed query, corresponding to the Example Query 1, containing a combination of Load, Group, Join and Store operators. Each test was executed 5 times and the results presented here have a standard deviation of less than 1% in all cases. The test applications were developed in Pig Latin (executed both on Pig and PonIC), native MapReduce and PACT.

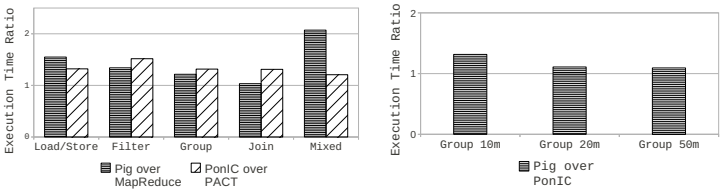
5.1 Implementation Overhead

Whenever using high-level languages, there is an overhead users have to pay in exchange for the abstraction offered. This overhead is one of the factors defining the value of the abstraction. Figure 3(a) shows the performance overhead for the Pig system over the corresponding native Hadoop MapReduce implementations and for PonIC over PACT. For Pig, this overhead includes setup, compiling, data conversion and plan optimization time. The results for Pig confirm already published results [9]; Pig is around 1.2 to 2 times slower than a native MapReduce application. Figure 3(a) also shows that PonIC's overhead is significantly lower and smaller than 1.6 in all cases. We believe that the smaller overhead is mainly due to the more efficient translation process. Since PonIC only supports a subset of Pig's features, the overhead could increase in a complete implementation. However, as we described in Section 3.3, in the worst case, an operator could be translated into PACT, using only the Map and Reduce Contracts. Such a naive translation would result into an overhead comparable to Pig's overhead.

In order to have a better idea on how the overhead changes depending on the dataset size, we ran the Group query for three different sizes of the `page_views` dataset. The results in Figure 3(b) show that the overhead caused by setup and compilation time has a heavier influence on smaller datasets.

5.2 Comparison with Pig and Hadoop MapReduce

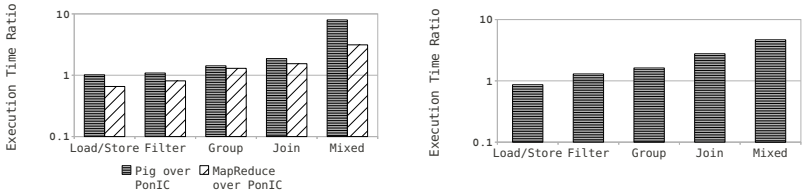
Figure 4(a) shows the execution time ratio of Pig and native Hadoop MapReduce over PonIC. Y axis is in logarithmic scale. PonIC matches Pig's execution time for the Load/Store and the Filter queries, while it is significantly faster in the rest of the cases. When compared to native MapReduce, PonIC is also faster, except from the Load/Store and Filter operations, for which setup and data conversion



(a) PonIC vs. Pig/MapReduce (b) Varying Data Sizes

Fig. 3. Evaluation Results: Overhead

times are dominant. In the case of Mixed query, PonIC is 8 times faster than Pig. The MapReduce Plan that Pig creates for this query contains two MapReduce jobs in order to implement the join and the group operations, involving a materialization step in between them. On the other hand, PonIC can execute faster, exploiting Stratosphere's data pipelining between Input Contracts. The main reason why PonIC is generally faster than Pig is demonstrated in Figure 4(b), which is a comparison between the execution time of native MapReduce and PACT implementations. It shows that, in all the cases except Load/Store, Stratosphere is faster than native MapReduce.



(a) PonIC vs. Pig/MapReduce (b) Native MapReduce vs. PACT

Fig. 4. Evaluation Results: Execution Time Comparison

6 Related Work

Among the supported high-level languages for MapReduce, Hive is probably the most popular and has been used in work similar to ours. Hive has been integrated with the ASTERIX system [5]. ASTERIX provides a data-agnostic algebra layer, which allows Hive to run on top of the Hyracks runtime. Hive execution plans are translated to ASTERIX algebra plans and better performance is achieved without any changes in the HiveQL queries. To our knowledge, no published evaluation measurements exist to support this claim.

The Shark system [13] allows HiveQL queries to execute on top of Spark [4], in an analogous way to ours with Pig and Stratosphere. However, Shark's goal is to provide a unified system where both SQL queries and iterative analytics applications can co-exist and execute efficiently. Our work and the Shark project share some discoveries regarding the limitations of the MapReduce-based

execution engines, which result in inefficient execution, namely the expensive data materialization and inflexibility of static pipelines over general DAGs.

There has been recent work in integrating JAQL with the Stratosphere system [14], which led to the creation of Meteor [15]. Meteor is a high-level language inspired by JAQL and lies on top of a relational algebra layer, Sopremo. Meteor programs are translated into Sopremo operators, which are then compiled into Input Contracts, in a way similar to our work. However, Meteor, like JAQL, only supports the JSON data model and no performance measurements are yet available, as far as we know. With our work, we benefit both Pig and Stratosphere users. Pig developers can gain improved performance without changing their applications, while Stratosphere users can now exploit the expressiveness of the Pig Latin language to develop applications faster and execute them on the Nephele execution engine, with only minimal compilation overhead.

7 Conclusions and Future Work

Existing programming models for Big Data analytics, such as MapReduce and PACT, have been a great contribution and are widely used. 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, who are used to work with higher-level languages. Therefore, improving the performance of systems like Pig is of great importance.

In this paper, we examined the feasibility of integrating Pig with Stratosphere. We show that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness. We concluded that, even though Pig is tightly coupled to the Hadoop execution engine, integration is possible by replacing the stack below the Logical Plan layer. The translation algorithm and prototype integration of Pig with Stratosphere allows execution of Pig Latin scripts in the Stratosphere execution engine, without modifying the scripts, while offering improved performance.

Several issues remain unexplored and are interesting for further investigation. 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. For example, Pig Latin could be extended to include keywords corresponding to Output Contracts or PACT's compiler hints. Since Stratosphere now offers its own high-level language, Meteor, it would also be very interesting to compare its expressiveness, usability and performance against Pig.

Acknowledgements. This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043. The authors would also like to thank the Stratosphere team for their help throughout this work.

References

1. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation. USENIX Association (2004)
2. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev (2007)
3. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers. ACM, New York (2009)
4. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud 2010 (2010)
5. Alsubaiee, S., Behm, A., Grover, R., Vernica, R., Borkar, V., Carey, M.J., Li, C.: Asterix: scalable warehouse-style web data integration. In: Proceedings of the Ninth International Workshop on Information Integration on the Web, IIWeb 2012. ACM (2012)
6. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In: Proceedings of the 1st ACM symposium on Cloud computing, SOCC 2010, pp. 119–130. ACM, New York (2010)
7. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C.C., Özcan, F., Shekita, E.J.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. PVLDB 4, 1272–1283 (2011)
8. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. 2(2), 1626–1629 (2009)
9. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of map-reduce: the pig experience. Proc. VLDB Endow. 2(2), 1414–1425 (2009)
10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1099–1110. ACM, New York (2008)
11. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE Computer Society, Washington, DC (2010)
12. Kalavri, V.: Integrating pig and stratosphere. Master's thesis, KTH, School of Information and Communication Technology, ICT (2012)
13. Engle, C., Lupher, A., Xin, R., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: fast data analysis using coarse-grained distributed memory. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, New York (2012)
14. Lawerentz, C., Nagel, C., Berezowski, J., Guether, M., Ringwald, M., Kaufmann, M., Vu, N.T., Lobach, S., Pieper, S., Bodner, T., Wurtz, C.: Project jaql on the cloud. Final report, TU Berlin (2011)
15. Heise, A., Rheinlaender, A., Leich, M., Leser, U., Naumann, F.: Meteor/sopremo: An extensible query language and operator model. In: Proceedings of the International Workshop on End-to-end Management of Big Data (BigData) in conjunction with VLDB 2012, Istanbul, Turkey (2012)