Rewriting and Code Generation for Dataflow Programs

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Wieland Hoffmann*
IBM Deutschland Research & Development GmbH
whoffman@de.ibm.com

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Nowadays, several data processing engines to analyze and query big data exist. In most of the cases, if users want to perform queries using these engines, the complete program has to be implemented in a supported programming language by the users themselves. This requires them to understanding both the programming language as well as the API of the platform and also learning how to control or even enable parallelism and concurrency. Especially with this tight integration into programming languages, the internal rewriting of queries to optimize the flow and order of the data and operators is another big challenge since the query optimization techniques are difficult to apply. In this paper, we want to address these problems by utilizing the dataflow model and a code generator and compiler for various platforms based on it, namely Piglet. The focus of this paper lies on stream processing platforms and, therefore, the associated challenges, especially for two exemplary engines, are described. Moving on from there, we introduce our internal procedure for rewriting dataflow graphs and finish with the presentation of our own DSL approach to even support user-defined rewriting rules.

Keywords

Query Optimization, Stream Processing, Dataflow, Dynamic Data, Data Analysis

1. INTRODUCTION

During the last years, the abstraction level of query languages was subject to some fluctuations. It all started with a very high level of abstraction by the provision of declarative query languages such as SQL or XQuery. These come with a lot of advantages like a standardized language, ease of use even for non-technical users, and automatic optimization possible through the well-known relations between the provided operators. However, this type of data retrieval is only limited extensible. By introducing MapReduce for Big Data challenges, a lot of new possibilities for data querying and processing were made available. For this, as a drawback, one has to deal with a greatly increased level of complexity, manual optimization, and, in general, a low level of abstraction. Hence, the current research tries to combine these worlds to provide platforms with programming language integrated APIs to raise this level again. For example, Apache Spark and Apache Flink use Scala (beside Java and Python) as a kind of dataflow language. However, to get along with these domain-specific languages (DSLs) the data analyst has to know and to understand the corresponding language and the API for the chosen platform. Furthermore, the user has to write a lot of code beside the actual query, build and deploy the code for the backend, and for a great part manually optimize the program. Thus, as already mentioned in [14], the idea is to provide a high-level declarative interface for streaming queries and at the same time a scalable cluster-based stream processing platform. From this idea, Piglet[1], a language extension to Pig Latin and dataflow compiler for multiple batch and streaming backends, recently originated to meet these requirements. The provision of such a multi-platform compiler brings some challenges. First of all, there are the different requirements and conditions for batch and stream processing. For instance, whereas in batch processing one deals with a finite amount of stored data tuples, working with streams means to get along with an open-ended continuous data stream. This is accompanied by fault-tolerance and elasticity tasks. In addition, there are not only differences between batch and streaming, but also platforms with the same processing type show different behaviors. This does not mean that some of the backends do something wrong, but rather arises from the different processing approaches (i.e., Spark Streaming via micro-batches, Flink Streaming and Storm via tuple-by-tuple). Another big challenge is also the automatic optimization of dataflows consisting of user-defined operators since they can normally only be treated as black boxes by the compiler.

With this paper, we try to wipe out the drawbacks of both the high-level declarative approach (e.g., extensibility) and scalable cluster-based data processing platforms (e.g., high coding effort) by combining these two worlds and describe the challenges associated with it. Here, the focus is mainly on streaming platforms. The paper consists of two main contributions:

*The work presented here was written as a Master's thesis at the TU Ilmenau [10].

[1] https://github.com/ksattler/piglet

28^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 24.05.2016 - 27.05.2016, Nörten-Hardenberg, Germany.

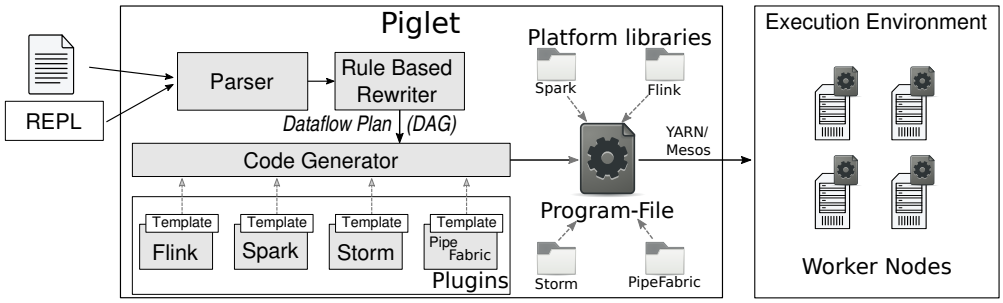

Figure 1: Piglet's internal architecture: The code generator uses the backend plugins to create platform specific programs.

- We provide a high-level abstraction for querying data in the form of a unified model of dataflows and realize the mapping to various platform-specific concepts.
- With the developed framework for rewriting dataflow graphs, we offer an easy way of implementing and integrating user-defined operators together with their rewriting rules into the dataflow.

Although the implementation was done within our own dataflow compiler Piglet, we think the ideas are not limited to this system.

2. THE DATAFLOW MODEL

The dataflow model is nothing really new as it dates back to the 1960's and 1970's [11, 16] having the motivation to automatically exploit parallelism. Thereby programs are represented internally as directed acyclic graphs (DAGs) with its nodes being mutually independent operation blocks, which are connected by input and output pipes to represent the dataflow. This creates a segmentation of the operations having no data dependency between each other and thus can be executed in parallel. The recognition of data dependencies can be accomplished automatically by the compiler and normally needs no action taken by the user.

These and further advantageous characteristics of the dataflow model are meant to be leveraged in Piglet, which uses such a model as internal representation. The goal of the project is to provide a language extension to Pig Latin and a compiler to generate code for several modern data processing platforms. In the course of this, the code is also built and deployed to the specified execution environment without any necessary intervention of the user.

In figure 1 the internal architecture of Piglet is presented. As a first step, the Pig Latin input is parsed into a dataflow plan which contains a list of objects whose types implement the `PigOperator` trait. One type exists for each operator of Pig and the additional ones provided by Piglet. The input can be provided either as pre-written files or by interactively entering statements via a Read-Eval-Print-Loop (REPL). In the next step, a rule based rewriter automatically optimizes (see section 5) and possibly adapts the dataflow plan depending inter alia on the target platform (see section 4). Subsequently, the rewritten plan is used together with the backend plugins to generate the target program. It is then zipped with the necessary platform library into a *jar* file and lastly deployed to the execution environment, which then takes care of the possibly parallel execution.

The purpose of the platform library is to hide some implementation details from the generated code, mainly used for source and sink operators, for example, schema extracting loading methods, objects for deployment or aggregate helper functions. Since backends are treated as plugins, one can extend the compiler with other platforms by creating a plugin, which should implement specific interfaces as well as a template file as input for the code generator to create the corresponding target code.

Due to the kind of data processing for streams, a lot of characteristics and challenges arise, which were not tackled in the batch implementation. The main challenges among them, which we pick up in the following two sections, are:

- windows,
- blocking operators, and
- different behavior of various streaming backends.

3. RELATED WORK

A similar goal of providing a unified model for defining and processing dataflows for different platforms is intended by the Google Dataflow model [1] and the corresponding Apache incubator project Beam[2]. In contrast to Piglet, Apache Beam comes with a Java SDK unifying and simplifying the creation of dataflow programs. As a drawback, this also means that the data analyst has to be familiar with Java and has to pick up the new API. Piglet, on the other hand, provides a high-level dataflow language allowing the user a much better abstraction and enabling many more optimization opportunities.

The challenges in terms of data stream processing mentioned above are also partly discussed in [12], where Krämer and Seeger define a logical algebra with precise query semantics for data streams based on the relational algebra. In contrast to batch processing, many standard operators such as join, aggregate or duplicate elimination (i.e., blocking operators) can only refer to a subset of the stream defined by windows. Instead of integrating windows directly into these standard operators, they decided to separate these functionalities as it was also done in Piglet. Thus, the redundant definition of window constructs within the operators is avoided and it also allows the user to apply multiple operations to one window specification. On top of that, the well-known semantics of the relational algebra is preserved as much as possible.

As implied by the third challenge above, there exist many data streaming systems, which propose several processing models especially for handling windows. Since there are no standards for querying streaming data, they all come up

[2] `http://incubator.apache.org/projects/beam.html`

with their own syntax and semantics. Because of that, the engines process queries in different ways although the user would expect the same behavior. On the other hand, several systems sometimes also express common capabilities in different ways. First introduced in [4] and later described in greater detail in [5], the SECRET model was proposed to provide a descriptive model for analyzing and comparing the execution semantics of stream processing engines. The four dimensions of the SECRET model are:

- ScopE - gives information about the window intervals.
- Content - maps the intervals to window contents.
- REport - reveals the conditions for window contents to become visible.
- Tick - states when the engine will trigger an action on an input stream.

With these dimensions, it is possible to compare and explain differences in behavior of the various systems. In their experiments, they reveal the variances in window construction, window reporting and in triggering. It was shown that even for the same engine sometimes it happens that the results are different. The reason for that is the different mapping of the content to scopes, for example due to the assignment of time values to tuples based on the system time. It was also shown that different approaches for reporting might also lead to different results for the same query. Thus, it can be seen that it is important to understand the window semantics behind a stream processing engine before writing queries for it. More explicitly for Piglet, this means, for instance, that the same query can produce different results for the various backends.

Beside the dataflow model and streaming challenges, there is also the problem of optimization of dataflow programs, which was already deeply discussed and implemented in the past. A famous system, for example, is EXODUS [8], which can be used to integrate generated application-specific optimizers into databases. For the generation of the target C-code, EXODUS needs the set of operators and methods, rules for transforming the query trees as well as cost functions for each operator as input. In the running system, the incoming queries are transformed into trees and optimized by repeatedly applying the passed rules. Thereby, it maintains information about all resulting alternative trees and the change in cost for each transformation. The successor framework of EXODUS is Cascade [7], which was introduced to improve and enrich its predecessor by, for example, extensibility, dynamic programming, and more flexibility. Especially the improved handling of predicates is highlighted, for example, by detaching them from a logical join operator to transform it into a physical nested loop operation and pushing it into the selection operator for the inner input tree.

There also exist optimization approaches for MapReduce programs. For instance, Spark SQL's Catalyst [2, 3] that is an optimizer integrated into Spark and used for Spark SQL and the corresponding data abstraction called `DataFrames`. Similar to the rewriter in Piglet, it uses many features of the programming language Scala for a seamless integration into the system's code. Advantages through that are, for instance, the ease of adding and connecting new rules, techniques, and features as well as the possibility for developers to define extensions. Roughly taken, Catalyst works with trees and applies manipulating rules to them. They use it in four phases, namely analysis, logical optimization, physical planning, and the final code generation. The analysis phase resolves, among other things, the relation and attribute names within the given query in the first place. On the resulting logical plan rule-based optimizations, such as reordering, are applied. The subsequent physical planning then maps physical operators from the Spark engine to the logical plan and selects the optimal plan based on a cost model (e.g., choosing the best fitting join implementation). As a final step, code is generated to run on each machine based on the selected physical plan.

4. FLINK AND SPARK STREAMING

As described in detail in [6], Flink[3] with its streaming API is one of the stream processing platforms integrated into Piglet. Furthermore, also Spark Streaming[4], Apache Storm[5] as well as PipeFabric [13] were added to the supported target streaming platforms of Piglet. The biggest challenge here is the presence of different semantics and behaviors of the backends as already described in general in section 2 and 3. In the following, some of the differences of Flink and Spark Streaming as well as general challenges with stream processing are sketched, which became conspicuous during the integration into the code generator.

Since basic Pig Latin is made for batch processing, it was necessary to enhance Piglet's input language by streaming operators such as `Socket_Read`, `Socket_Write` and `Window` as well as underlying loading functions like `PigStream` in the first place. Beside the language extensions, the compiler has also been equipped with corresponding mappings from the dataflow operators to the streaming APIs. It was found that specific operators supported in batch processing are not logical or sometimes not even possible in its original form for streams. That is why these operators were either completely omitted (e.g., `Limit`) or only supported inside of windows (e.g., `Order By` and `Distinct`).

To make windows and joins work for Flink Streaming the dataflow plan must be rewritten, that is, defining the window scope as well as rewriting `Join` and `Cross` operators. The former means to decide which operators need to be applied to the window and from when the stream is flattened again. For that `Foreach`, `Join`, `Cross` and sink nodes can represent the terminator of a window scope. At this point, a corresponding `WindowApply` node is inserted into the dataflow, which is taken up in the code generation step. The later window rewriting process searches for `Join` and `Cross` operators to assign a window definition to them derived from the input nodes. This is necessary because the Flink Streaming API only allows to join two data streams on a common time-based window, but not two windowed streams or a dataset with a windowed stream like in Spark Streaming. Thus, three requirements must be fulfilled beforehand:

- the direct input nodes must be windows,
- the windows must be time-based, and
- all input nodes require having the same window and sliding size.

If these requirements are met the dataflow plan is rewritten in a way as shown in figure 2 with the associated Piglet query:

[3] `https://flink.apache.org/`
[4] `https://spark.apache.org/`
[5] `https://storm.apache.org/`

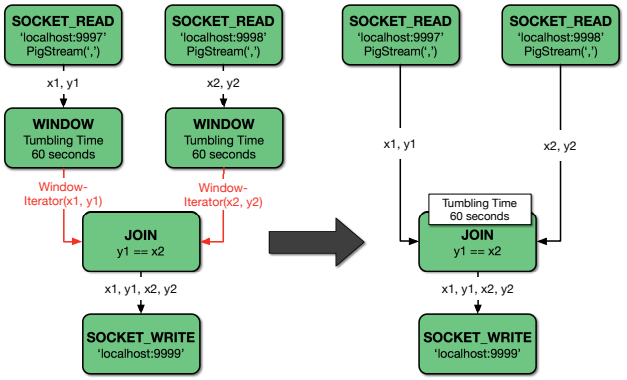

Figure 2: Rewriting of a `Join` operator in Piglet for Flink Streaming

```
in1 = SOCKET_READ 'localhost:9997' USING PigStream
    (',') AS (x1: int, y1: int);
in2 = SOCKET_READ 'localhost:9998' USING PigStream
    (',') AS (x2: int, y2: chararray);
w1 = WINDOW in1 RANGE 60 SECONDS;
w2 = WINDOW in2 RANGE 60 SECONDS;
joined = JOIN w1 BY y1, w2 BY x2;
SOCKET_WRITE joined TO 'localhost:9999';
```

Internally the inputs of the window nodes are collected and used as new input for the `Join` operator. That is, the window definition is extracted and inserted into the unifying operator, the pipes are redirected, and finally unnecessary operators are removed.

For Spark Streaming, the window rewriting step is not necessary since windowed streams can be joined directly. The only addition is a transformation of the tuples to key-value pairs before joining the streaming subsets. This task is done directly in the code generation phase and thus is not visible in the dataflow plan.

Beside the `Join`- and `Cross`-specific rewritings, the general window scope definition is partly implemented within the rewriter, too. Currently, the aforementioned `WindowApply` node is simply inserted into the dataflow as an extra path from the window start to the end of the scope. The code generator then takes care of integrating the operations into the apply method. An example dataflow written in our Pig language extension could be the following:

```
in = LOAD 'file.csv' USING PigStream(',') AS (x: Int,
     y: Int);
win = WINDOW in RANGE 20 SECONDS;
dis = DISTINCT win;
grp = GROUP dis BY win.x;
cnt = FOREACH grp GENERATE group, COUNT(grp);
STORE cnt INTO 'amount.csv';
```

Here, the read tuples are collected in a tumbling window of 20 seconds. The scope of this window is terminated by the `Foreach` statement as explained above. Thus, the duplicate elimination, grouping and aggregation operation are put into the apply method. For Spark Streaming, the dataflow is unchanged since the window operation just adjusts the micro-batch contents. The structure of the resulting codes for Spark and Flink Streaming is visualized in figure 3.

Another challenge, as mentioned in section 2, are blocking operators, which in batch processing are applied to the complete input. In the previous examples one could see such operations, namely `Join` and `Count` (in general: aggregates). Since a data stream can be potentially endless, one cannot just collect the data until the last tuple arrives. Even if the

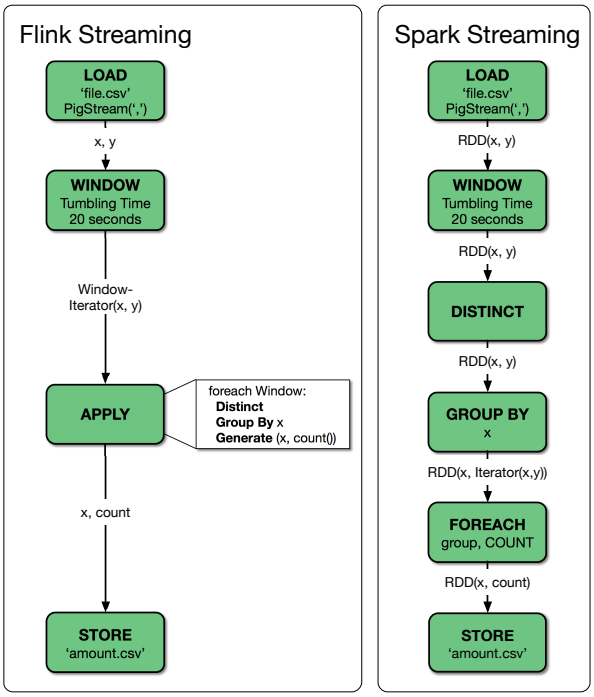

Figure 3: Structure of the generated code for Flink and Spark Streaming

stream eventually ends, it is not guaranteed that all the tuples fit in memory. Nevertheless, to support such operators there are two alternative approaches. On the one hand, one could apply them only to a subset like it was done in the examples with the help of windows. On the other hand, the results could be updated for every incoming tuple by using stateful operators. In the current state, we also support the second variant for aggregates, which are also called rolling aggregates or accumulations. Since Spark and Flink Streaming support stateful operators, we directly use them for this task. To achieve this behavior one just has to omit the window statement.

5. QUERY PLAN PROCESSING

As shown in Figure 1 and described in detail in [10], Piglet performs a rewriting step that transforms the output of the parser by repeatedly applying rules to the dataflow plan before passing it on to the code generator. Compared to other frameworks like Cascades [7] and EXODUS [8], the rewriting process in Piglet is fully integrated into Piglet itself and, therefore, can use features of the host language. However, it would not be easy to integrate it into another system.

The rewriting step serves a few different purposes. The first is optimization by modifying the dataflow plan to exploit relationships between different operators of the same or different types. This allows, for example, to rewrite the order of `Filter` and `Order By` operators such that the `Filter` operator is always executed first, potentially shrinking (but never enlarging) the data set flowing to the `Order By` operator.

The second goal is the support of Pig operators that cannot be mapped 1:1 to Spark or Flink operations, but can be rewritten to existing functions (for example Pig's `Split Into` operator is rewritten to multiple `Filter` operators).

The same principles can be applied to support new operators beyond those offered by Pig whose syntax better fit a certain problem domain. We chose to implement most of the rewriting rules of [9] as examples for this goal.

Piglet also allows the user to embed user-defined functions for use with Pig's `Foreach` operator directly into the input script like in the following example:

```
<% def myFunc(i: Int): Int = i + 42 %>
out = FOREACH in GENERATE myFunc(f1);
```

Despite those usually being small functions, they still might interact with other types of operators, so being able to supply rewriting rules in addition to the user-defined functions themselves was another goal.

At a very high level, the rewriting process consists of two parts. The first of those is activating available rules at startup, depending on the settings with which Piglet is run, and applying the rules to a dataflow plan after the parser has been executed. Several objects called Rulesets, one for each backend and language feature, exist, each providing a method called `registerRules` which registers zero or more functions with an object called `Rewriter`. Rules are represented as functions of the type

```
Function1[Any, Option[Any]]
```

mapping each input object to either `Some(newobject)`, indicating that the object has been changed and is to be replaced by `newobject`, or `None`, indicating that the function did not change the object. As soon as the options have been read, the registration method of each applicable ruleset is executed. After the input script has been read and transformed into Scala objects by the Parser, the method `processPlan` of the rewriter object will be called, passing the dataflow plan as the argument. This method will then repeatedly apply all registered rules to all operators in the plan in a top-down fashion, starting with the source nodes, until no rule leads to a change anymore. For traversing the dataflow plan we use the library Kiama [15]. Not only does it provide a data type `Strategy` that abstracts rewriting operations (this type extends the function type mentioned previously), it also includes methods for combining several separate operations of that type (`and`, `ior`, ...), extending them with repetition (`repeat`, `repeat`, ...), and traversal (`bottomup`, `topdown`, ...) behaviour, or any combination of those. This means that, unless otherwise necessary, a rule only needs to handle a single operator as its input, being applied to all possible operators is taken care of by the Piglet system. The traversal operations work with every type that implements the `Rewritable` trait supplied by Kiama, which the operator objects in Piglet do.

6. USER-DEFINED REWRITING

While implementing a set of rules it became apparent that our first approach writing them (as ordinary functions) is quite cumbersome. It involved first checking the input type of the object (because the type of the argument is `Any`, but most rules only apply to a specific type of operator), checking additional conditions on the operator object (for example the function called by a `Foreach` operator), usually through a series of calls of the form `if (...) {return None}` and only then changing the object.

We, therefore, make extensive use of features offered by the Scala programming language to ease the implementation and registration of rules. Its local type inference and

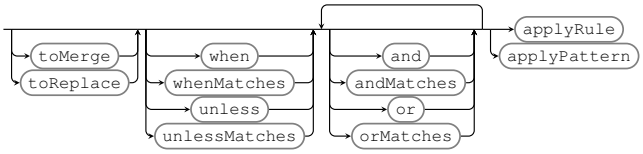

Figure 4: Syntax diagram for the rewriting DSL

`ClassTag`s, for example, are used to great effect for making it easier to implement rules that are only applied to specific types of operators by wrapping the rule at runtime and registering the wrapper, which is now able to only call the wrapped function if the input object is of a type that it accepts, thereby working around the type erasure that happens at compile time for languages targeting the JVM.

Scala also makes it easy to develop embedded DSLs by allowing to replace the dot before method calls and the parentheses around the argument with spaces if the method is of arity one (it takes only one argument). This is called `infix notation` and is used by, for example, the ScalaTest framework to produce code similar to english sentences.

Instead of forcing each rule to implement the steps outlined above anew, we decided to use the infix notation to provide a DSL of our own, which takes care of creating the wrapper function mentioned earlier automatically (given appropriate type signatures) and moves the checks for additional conditions out of the rule itself into zero or more anonymous functions.

Figure 4 shows the syntax for this DSL. The `applyRule` and `applyPattern` methods can be used to register a single function as a rewriting rule. The methods `when`, `unless`, `and`, `or` and their counterparts ending in `Matches` can be used to supply additional functions for checking preconditions on the input object. The only difference between both types is that functions whose name ends in `Matches` accept partial functions as their argument. Lastly, `toMerge` and `toReplace` will activate special behaviour for the two cases of merging operators (passing not one, but two operators to the other functions) and replacing one operator with exactly one other. Internally the builder pattern is used to keep track of all the preconditions and to wrap the function given to `applyRule` in them before registering it. The following is an example of a rule utilizing the DSL:

```
def groupedSchemaJoinEarlyAbort(op: BGPFilter)
      Boolean

Rewriter unless groupedSchemaJoinEarlyAbort
  and { op => RDF.isPathJoin(op.patterns) }
  applyRule J4
```

In the type signature of `groupedSchemaJoinEarlyAbort` the type of the operator is restricted, so the anonymous function passed to `and` can already use that information to access the `patterns` attribute that only `BGPFilter` objects have. The same operations can be used for implementing rewriting rules for user-defined functions embedded in the script. Piglet's parser has been extended to treat everything in a section of embedded code that follows the keyword `rules:` as Scala code containing the implementation of rewriting rules as in the following example:

```
<% def myFunc(i: Int): Int = i + 42
   rules:
   // Rules can be written here
 %>
```

A few imports are automatically added to the code before it is executed inside the same JVM Piglet is running in. To

further ease the implementation of rules for embedded code, which only apply to `Foreach` operators that usually only call a single function, we provide an additional extractor object that returns the operator object and the name of the function that is getting called if the operator matches that pattern. Other extractor objects for providing quick access to predecessors and successors of operators are provided as well. All of them return a tuple whose first element is the object that is matched, so the patterns can be nested.

7. CONCLUSION AND FUTURE WORK

We have presented our approaches of code generation and rewriting for dataflow programs integrated into Piglet with the major concentration on data streaming. As mentioned above, we think the concepts can be applied to other systems, too.

As a future perspective, we also want to further extend the presented ideas. Relating to streaming features, there is still a lot to do for supporting a comprehensive range of application scenarios. First of all, there exist many widespread connectors for source and sink nodes, such as Apache Kafka, Twitter's Streaming API or Flume, which should be made available in Piglet. Another point in our agenda are operator alternatives, for example, `Join`, since at the current state only equijoins are supported. It was also considered to support states for the `Sample` (e.g., reservoir sampling) or `Order By` (e.g., keeping all seen tuples and reporting the new order) operator. Obviously, one must always keep in mind the required memory for such an approach. On top of that, because the window operator is one of the main concepts in data streaming, we think about adding more variants like delta-based windows. As mentioned in section 4 a `WindowApply` node is simply inserted to define the window scope. Another approach would be to put all operations within the window scope directly into this node in the form of a sub-plan already during the rewriting step. Thereby, the code being produced is more transparent in the internal representation and this is also a much cleaner solution.

Furthermore, the rewriting system can be extended to make it easier to assign specific properties to operator types that can automatically be used by general rewriting rules that are not tailored to one type of operator. Another possible extension is the support for rewriting by utilizing algebraic properties of operators such as associativity and distributivity. Using these properties, not only one, but a set of new plans can be generated from an initial plan. If a Pig script is run multiple times, information gathered during execution could be used to choose the best of those plans.

Apart from embedded functions, the parser and rewriting rules associated with them, adding new types of operators and rules has to happen in Piglet's code itself. Extending this to support loading user-defined operator types at runtime, possibly via the `ClassLoader` system, is also an option for the future.

8. REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[2] M. Armbrust, Y. Huai, C. Liang, R. Xin, and M. Zaharia. Deep Dive into Spark SQL's Catalyst Optimizer. `https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html`, 2015.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM, 2015.

[4] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[5] N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, and I. Botan. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal*, 22(4):421–446, 2013.

[6] P. Götze. Code Generation for Dataflow Programs using the example of Flink Streaming. Master Thesis, Technische Universität Ilmenau, October 2015.

[7] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[8] G. Graefe and D. J. DeWitt. *The EXODUS Optimizer Generator*, volume 16. ACM, 1987.

[9] S. Hagedorn, K. Hose, and K.-U. Sattler. SPARQling Pig - Processing Linked Data with Pig Latin. In *BTW*, March 2015.

[10] W. Hoffmann. Regelbasierte Transformation von Datenflussgraphen. Master Thesis, Technische Universität Ilmenau, January 2016.

[11] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. In *ACM Computing Surveys*, volume 36, pages 1–34, 2004.

[12] J. Krämer and B. Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM Trans. Database Syst.*, 34(1):4:1–4:49, Apr. 2009.

[13] O. Saleh and K.-U. Sattler. The Pipeflow Approach: Write Once, Run in Different Stream-processing Engines. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 368–371. ACM, 2015.

[14] K.-U. Sattler and F. Beier. Towards Elastic Stream Processing: Patterns and Infrastructure. *First International Workshop on Big Dynamic Distributed Data (BD3)*, 2013.

[15] T. Sloane. Experiences with Domain-specific Language Embedding in Scala. In *Domain-Specific Program Development*, page 7, 2008.

[16] T. B. Sousa. Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium on Informatics Engineering*, 2012.