Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 1, 69–79
DOI: 10.2478/amcs-2019-0005

amcs

PARALLELIZING USER–DEFINED FUNCTIONS IN THE ETL WORKFLOW USING ORCHESTRATION STYLE SHEETS

SYED MUHAMMAD FAWAD ALI [a,b,*], JOHANNES MEY [c], MAIK THIELE [c]

[a]Faculty of Computing
Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland

[b]Data Engineering
trivago N.V. Leipzig, Bosestrasse 4, 04109, Leipzig, Germany
e-mail: fawadali.ali@gmail.com

[c]Faculty of Computer Science
Technical University of Dresden, Helmholtzstrasse 10, 01069, Dresden, Germany

Today's ETL tools provide capabilities to develop custom code as user-defined functions (UDFs) to extend the expressiveness of the standard ETL operators. However, while this allows us to easily add new functionalities, it also comes with the risk that the custom code is not intended to be optimized, e.g., by parallelism, and for this reason, it performs poorly for data-intensive ETL workflows. In this paper we present a novel framework, which allows the ETL developer to choose a design pattern in order to write parallelizable code and generates a configuration for the UDFs to be executed in a distributed environment. This enables ETL developers with minimum expertise in distributed and parallel computing to develop UDFs without taking care of parallelization configurations and complexities. We perform experiments on large-scale datasets based on TPC-DS and BigBench. The results show that our approach significantly reduces the effort of ETL developers and at the same time generates efficient parallel configurations to support complex and data-intensive ETL tasks.

Keywords: ETL workflow, parallel ETL operators, parallel algorithmic skeletons, user-defined functions.

1. Introduction

As the volume, variety, and velocity (3Vs) of data are growing at a record rate, extract-transform-load (ETL) processes and data analysis workflows are becoming more and more sophisticated. Especially the variety of today's data but also the wide range of different analytical use-cases increasingly exceed the expressiveness of standard ETL operators. As an example from the data cleansing perspective, the messy and noisy nature of big data demands new types of cleansing operators, such as outlier detection or de-duplication, that specifically fit the ever-changing characteristics of the data. The same applies to the data analytics perspective, where we find a zoo of algorithms such as classification, regression, clustering, collaborative filtering, and many more.

To overcome the limited expressive power provided by the standard ETL operators, most ETL tools offer the functionality to write custom code as user-defined functions (UDFs). A UDF is a software program written in any programming, scripting, or procedural language. These UDFs allow the ETL developer to extend the functionality of an ETL tool that is outside a scope of the already provided built-in ETL operators. For example, a UDF can be used to perform aggregations or any kind of run-time intensive computations on a data set that may be necessary before loading into a data warehouse (DW). It may be written by the ETL developer who is developing a data pipeline (ETL) or by any third-party. Therefore, it may be more prone to errors and less efficient, which may result in a performance bottleneck due to its poor code and high computational complexity. In an ETL workflow, a UDF is normally considered a black-box activity, i.e., it is difficult to assess the run-time and space complexity of a UDF. However, if the UDF is already optimized, or is configurable to be optimized by an ETL framework without changing its code, it may help the ETL

*Corresponding author

developer to optimize the blocking UDF activities in an ETL workflow.

We carried out a thorough research on existing techniques for optimizing ETL workflows in an ETL framework (Ali and Wrembel, 2017), and one of the conclusions of our research was to incorporate in an ETL framework the functionality to allow ETL developers to write parallelizable user-defined functions that tackle the range of different analytical use-cases.

In this paper, we present a novel approach to incorporate the functionality in an ETL framework which assists the ETL developer in writing parallelizable UDFs to be executed in a distributed environment, e.g., Hadoop, Flink, etc. To achieve our goal, we leverage the so-called *orchestration style sheet (OSS) processor* that encapsulates and separates the parallelization concern from the development of UDFs. This processor generates parallelizable code and a set of configurations to execute a generated UDF in a distributed environment.

In particular, our contributions are the following:

- We provide a software application for the integration of any open source ETL framework to facilitate the ETL developer in writing a UDF by separating parallelization concerns from the code, thus reducing potential error sources in the otherwise manual and cumbersome parallelization process. The choice of the open source ETL tool was made to use it as a sandbox for our approach, because it provides more flexibility and control to the developers in order to develop custom ETL operators and alter currently existing ones. For example, in this paper, we used Pentaho Data Integrator, which allows the developers to program user-defined functions and generate them as any other built-in ETL operator.

- We provide code skeletons or design patterns to be used by our application to reduce the amount of effort required by the ETL developer in writing complex and efficient programs.

- We present experiments on a large-scale dataset based on BigBench (Ghazal *et al*., 2013), proving that optimizing the computing-intensive user-defined activity improves the overall performance of an ETL workflow.

Related work is presented in Section 2. In Section 3 we describe the use case scenario of our running example, which we will use throughout the paper. We then introduce the orchestration style sheet (OSS) processor in Section 4. Our proposed framework and approach towards generating parallelizable UDFs using the OSS are described in Section 5. Experimental evaluation for the feasibility of our approach is discussed in Section 6. Conclusions are included in Section 7.

2. Related work

There has been a lot of research for the past decade on the optimization of ETL workflows due to a critical requirement on execution time. The research by Simitsis *et al.* (2005; 2010), Tziovara *et al*. (2007), Kumar and Kumar (2010), Karagiannis *et al*. (2013), or Vassiliadis *et al*. (2009) highlights the problem of ill-timed completion of an ETL workflow and discusses the methods to improve its execution performance.

The work presented by Simitsis *et al*. (2005) proposes to optimize the ETL workflow by re-ordering the ETL activities in a directed acyclic graph (DAG) by pushing the highly-selective activities at the beginning of the flow. Simitsis *et al*. (2010) use the same approach as previously (Simitsis *et al*., 2005), but now it is more focused on generating an optimal ETL workflow in terms of performance, fault-tolerance, and freshness. Similarly to the aforementioned approaches in terms of ETL workflow design, Tziovara *et al*. (2007) propose to change the order of input tuples to improve the execution cost of an ETL activity, which results in the optimal overall cost of execution flow. Kumar and Kumar (2010) propose a slightly different approach for logical optimization of ETL workflows by using the dependency graph and efficient heuristics. The above approaches can be categorized as *graph-based* ones. Their pros and cons are summarized as follows:

Pros: The approach to optimize the ETL workflow is semi-autonomous, i.e., algorithms are used to transform the given input graph-based ETL workflow to the more efficient but semantically equivalent workflow.

Cons: However, the optimization algorithm may take a long time to generate the efficient ETL workflow in the case of large and complex input ETL workflows. Furthermore, the aforementioned approaches do not support UDFs.

Karagiannis *et al*. (2013) discuss the scheduling strategies to optimize the performance of an ETL workflow in terms of execution time and memory consumption without a loss of data.

Pros: The aforementioned scheduling strategy is also semi-autonomous, i.e., it takes the ETL workflow as user input and applies an algorithm on it based on different policies in order to generate an efficient workflow. On top of this, monitoring ETL workflows is in place for scheduling ETL activities in a workflow.

Cons: There is a possibility of losing data during scheduling. Therefore, the approach is not applicable to most of the traditional ETL processing. Moreover, the approach does not support UDFs in an ETL workflow.

Parallel programming is a popular strategy to improve the performance of data and computation of intensive workflows. Parallelism can be achieved either by partitioning the data into N subsets and process each

subset in N parallel sub-flows (data parallelism), or by using pipeline parallelism (task parallelism).

Currently, in the literature, research work on parallelizing ETL workflows is not attracting much attention and most of the work is related to the traditional data-flow parallelism. MapReduce (Dean and Ghemawat, 2008) is one of the popular programming models focusing on automatic data-flow parallelism. It is a popular choice to perform big data analysis with data mining algorithms in a parallel distributed computing environment (Weinberg and Last, 2017). The MapReduce programming model has proven a significant decrease in the execution time of computing-intensive workflows or processes when executing in a distributed parallel environment, e.g., Hadoop (González-Vélez and Kontagora, 2011).

Parallelization Contract (PACT) (Battré *et al.*, 2010) is another programming model for parallel processing of a massive data set and is based on the MapReduce model. Another approach towards parallelizing the data flow is Structured Computations Optimized for Parallel Execution (SCOPE) (Chaiken *et al.*, 2008). It is a declarative and extensible scripting language similar to SQL to simplify data analysis of a massive amount of data residing on clusters of hundreds or thousands of machines.

Liu *et al.* (2013) as well as Thomsen and Pedersen (2011) propose approaches towards incorporating parallelism in the ETL workflow. Liu *et al.* (2013) present a parallel dimensional ETL framework based on MapReduce called ETLMR. Thomsen and Pedersen (2011) propose a method to exploit parallelism in the ETL workflow at a code level by introducing both *task parallel* and *data parallel* strategies to attain the maximum performance. It provides different constructs that allow the ETL developer to convert the linear ETL workflow into its corresponding parallel flow. These constructs can provide significant improvement in performance if applied carefully at the parts of the ETL workflow which do not prevent parallelization. For the aforementioned parallelism based approaches, the pros and cons are summarized as follows:

Pros: The proposed approaches provide a detailed account of parallelization techniques to improve the execution and cost performance of ETL workflows. They also partially support UDFs in ETL workflows, although do not support the optimization of UDFs.

Cons: The proposed approaches do not provide any cost model to identify the required degree of parallelism. Hence, the ETL developer has to either perform a trial-and-error method or to execute ETL transformations using test data to figure out the required degree of parallelism.

Most of the discussed approaches are limited to relational operators, whereas it is important to optimize both relational operators as well as UDFs in order to optimize complex ETL workflows. In our paper, we address the optimization of the ETL workflow through parallelizing UDFs. Similarly to Thomsen and Pedersen (2011), we provide constructs (style-sheets) to write efficient parallel UDFs without concerning parallelism. Furthermore, we are additionally focusing on the UDFs that support semi-structured and unstructured data sets to pre-compute computing-intensive analytic queries.

Große *et al.* (2014) acknowledge the importance of optimizing both relational operators and UDFs to optimize the execution performance. This work leverages the use of annotations in order to parallelize UDFs. However, the approach is limited to the UDFs written as table functions. Also, it requires the developer to annotate the custom code using annotations supported by the optimizer to generate the optimized parallel execution plan. This requires the developer to predict which parts of the code needs to be parallelized in order to maximize the execution performance, whereas we provide the ETL developer with style-sheets as Java code templates. Our solution is programming language-independent, extensible, and capable of generating parallel programs in different procedural and declarative programming languages. Hence, in this paper we address parallelization of UDFs in ETL workflow, which is realized by using re-usable style sheets (OSS) to facilitate the ETL developers to write efficient parallelizable UDFs in an ETL framework (PDI) with minimum programming effort and limited knowledge of the distributed environment and programming.

3. Running example

To motivate our discussion, we borrowed the product retailer use case of BigBench (Ghazal *et al.*, 2013), which covers 3Vs of big data. The data model consists of structured, semi-structured, and unstructured components. The structured part of data is adopted from the TPC-DS benchmark (Nambiar and Poess, 2006). The semi-structured data consists of user clicks on the product retailer's website. The unstructured portion of data comprises product reviews submitted online in the English language.

As a use case for the remainder of the paper we focus on an analytical scenario that compromises sentiment analysis of product reviews. The data set contains one or more reviews for each product submitted online by the users of the products. Since it would not be acceptable to perform sentiment analysis over a huge amount of data every time during query processing, we want to pre-compute the sentiments. The respective ETL process developed in Pentaho Data Integrator (PDI) is depicted in Fig. 1. The ETL workflow sources data from tables *INVENTORY*, *DATE DIM*, *WAREHOUSE*, *ITEM*, and *PRODUCT_REVIEWS*. The process fetches the products

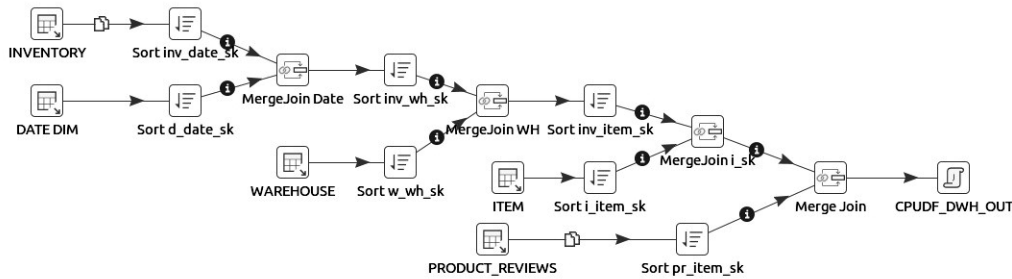

Fig. 1. ETL workflow for the running example.

present in the inventory since the year 2000 onward and are available in all the warehouses located in the US. Then it applies the sentiment analysis algorithm on the incoming data set in order to classify the unstructured product reviews as *Negative* or *Positive*. This functionality is implemented as a UDF called *CPUDF_DWH_OUT*. Finally, the computed result is exposed for analysis as a table that can be queried later by the analyst.

4. Orchestration style sheets (OSSs)

To support multiple, potentially differently parallelized target platforms with as little implementation overhead as possible, *parallel algorithmic skeletons (PASs)* can be used. PASs are defined as algorithmic skeletons, or parallelism patterns, which are high-level parallel programming models for parallel and distributed computing. These can be provided as libraries working on specific data structures like vectors and matrices as well as algorithms like MapReduce. However, these libraries are constrained to their supported data structures and algorithms, making them unsuitable for problems not meeting these constraints.

In these cases, pragma languages offer a very flexible, yet low-level, alternative: languages like OpenMP (Dagum and Menon, 1998) support a wide range of target platforms while offering high customizability. The price for this, however, is an increased implementation overhead.

Pragma languages are also known as directive languages. These are described as a construct that specifies how a compiler processes its input. Directives are not part of the grammar of a programming language, and may vary from compiler to compiler. They can be processed by a pre-processor to specify compiler behavior. In both task- and loop-based parallelization, the pragmas have to be repeated for every task or loop, each time with slightly different parameters.

OSSs (Mey *et al.*, 2016) provide a way to combine the simplicity of skeleton libraries with the flexibility of pragma languages using *invasive software composition (ISC)* (Aßmann, 2003), hence achieving language and platform independence. The central idea is to split the code into reusable code *fragments* that can be *woven* into different variants of the source code. Two aspects of this weaving process are particularly important.

First, fragment specification and weaving specification are performed declaratively in *style sheets* and *recipes*. The former contain *styles* consisting of code fragments and *addressing expressions* determining the positions in the source code in which they can be inserted. The code fragments themselves can contain variability points, *slots*, that can serve as positions in which other fragments can be inserted. The latter specification is done with *recipes* that determine the selection and order of styles to be applied to the code. Different recipes can be used to acquire different code variants. Figure 2 gives an overview over the involved artifacts in OSS code weaving.

The second important aspect is that fragment specifications can contain attributes, special code fragments that are not contained in the style definition but rather computed using the program code. Using attributes, the results of problem-specific, user-defined static analyses of the source code can be directly inserted into the code. In the presented use case, this analysis is employed to discuss and transform data types to adapt them to the interfaces provided by the distributed framework—Hadoop.

Furthermore, attributes can be used to specify application-specific variability points. This permits all OSS-specific parts of the code to be hidden from the application developer. Code fragments specified by him/her do not have to be included in style files but can also be specified implicitly, e.g., as class members at specifically defined positions. This method is used in the example as shown in Listing 1.

Source code composition of OSSs including fragment computation with attributes is performed utilizing *reference attribute grammars (RAGs)* (Hedin, 2000), a well-known technology in compiler construction. The OSS processor uses SkAT (Karol, 2015), a composition system built using the RAG tool JastAdd (Ekman and Hedin, 2007), which

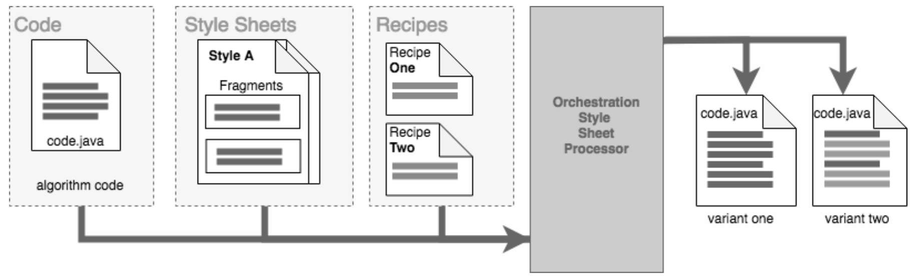

Fig. 2. Workflow of the OSS processor showing the required and generated artifacts.

allows type-safe and well-formed composition of code fragments and utilization of an extensible Java compiler, ExtendJ. The OSSs processor replaces the programmatic code composition of SkAT with the presented declarative approach. The computed fragments used by OSS are implemented as attributes of the attribute grammar, which in JastAdd are simply specifically annotated methods added to language elements with an aspect weaver (Kiczales *et al*., 1997). Thus, writing new, user-defined, and problem-specific attributes is a feasible task for a Java and Hadoop developer, enabling future extensions to distributed frameworks and programming paradigms other than Hadoop, e.g., Spark, Flink.

5. Generating parallelizable UDFs for an ETL workflow

In order to facilitate the ETL developer to write parallelizable UDFs without the parallelization and optimization aspects of a program, we contribute the *configurable-parallelizable UDF generator* (cp-UDF). It can easily be integrated into an open source ETL framework, e.g., Pentaho Data Integrator (Spoon), as a third-party tool. It utilizes OSSs to generate parallelizable code. In this approach we used an OSS because it is a lightweight and extensible approach that can be adopted to any target run-time on any high performance computing (HPC) cluster, e.g., a graphics processing unit (GPU), or can be adopted by parallel computing frameworks such as Hadoop and Spark.

Figure 3 shows the three-tier architecture of our proposed framework. The top layer depicts Pentaho Data Integrator (PDI), which provides a graphical user interface to create ETL workflows. The middle layer comprises cp-UDF and the OSS composition system to provide parallel skeletons to the ETL developer, thus enabling him/her to write parallelizable code without taking care of the critical parallelization details (i.e., degree of parallelization—specified by the number of data partitions, the number of map and reduce tasks in the case of Hadoop as a distributed framework). Subsequently, an OSS is used to generate configurations for the user's code, which are finally executed in a distributed environment—Hadoop, as shown in the bottom tier.

The involved artifacts and procedures are enumerated in Fig. 3 and will be explained in the following section to illustrate the process of generating optimized UDFs to be executed in a distributed environment. We will explain the working of cp-UDF and application of the OSS processor with the help of our running example described in Section 3.

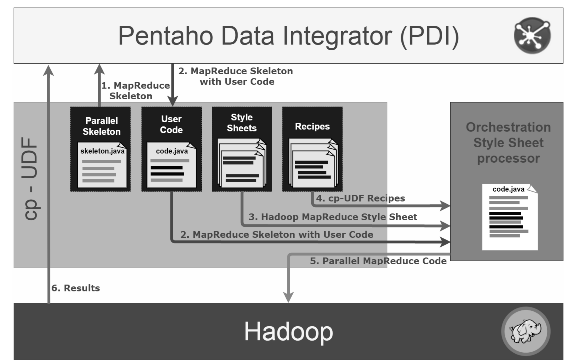

Fig. 3. cp-UDF high-level design for the generation of parallelizable code using an OSS.

5.1. Using map-reduce OSS for sentiment analysis UDF. The running example in Section 3 discusses the use case of sentiment analysis of product reviews as a UDF in an ETL workflow performed by a user-defined step depicted as *CPUDF_DWH_OUT* in Fig. 1 (cf. Section 3). The idea is to pre-compute the user sentiments during the ETL phase and propagate the computed results into a data mart. This would help the data analysts to avoid executing the high latency query every time they want to make decisions based on the user sentiments.

To assist the ETL developer to write parallelizable code for sentiment analysis, we provide him/her with different *parallel algorithmic skeletons (PASs)* or code skeletons that can be executed in a distributed environment, for example, worker-farm, divide and conquer, branch and bound, systolic, MapReduce or Spark code skeleton, where the ETL developer has to insert his/her code for sentiment analysis into the provided

skeleton (Fig. 3, Step 1).

Listing 1 shows the template for the Hadoop MapReduce use case with comments highlighting the positions at which the user can include his/her code[1].

As the template is a regular Java class, additional code like helper functions and definitions of fields can be added, if necessary. The template is filled with the required map and reduce code by the ETL developer and is sent to cp-UDF, as depicted in Fig. 3, Step 2. The ETL developer only has to know the theoretical details of MapReduce paradigm so that he/she can logically divide the UDF code into the map and reduce functions.

The critical and most important mode of parallelization, i.e., deciding on the optimal number of mappers and reducers, the number of partitions to make, the processing power of the virtual machine to execute the UDF in order to achieve maximum performance and all modifications to the code, is denoted in style sheets (*Runner Class* in the case of the MapReduce paradigm) automatically provided by cp-UDF, hence separating the parallelization concerns from the code.

Therefore, the ETL developer does not have to provide the critical details of parallelization. cp-UDF will provide the best possible configuration for executing the UDF in a distributed framework using *recipes* in the form of a *Runner class*.

Listing 2 depicts an example of a Map *style sheet*, which modifies the `map` method provided by the ETL developer in Listing 1, inserts the required data types and returns values.

A *recipe* is used to trigger the composition (Fig. 3, Steps 3 and 4). The recipe determines the selection and order of styles to be applied to the code. Listing 3 contains the recipe used, which first applies map and then reduce style sheets (cf. Lines 2 and 3, respectively) on to the provided UDF code as well as appends the runner class to the UDF in order to specify the parallelization configuration. Finally, with the provided skeleton code including the UDFs, cp-UDF invokes the OSS processor to build a parallelized version. The OSS processor is considered a black-box for cp-UDF and we assume that the output of the OSS processor will always be correct and accurate.

Using these input artifacts, the OSS generates a MapReduce (parallelized) version of the user-defined function (Fig. 3, Step 5) that is subsequently processed by Hadoop.

Currently, in the proposed framework, we used a *runner* class (i.e., a configuration class for optimal parallelization) best suited for the available distributed environment. However, in a future work, we will introduce the mechanism in our proposed framework to generate multiple *recipes*, i.e., multiple parallel configurations, and then to choose an optimal configuration for the distributed environment based on cost and computation performance requirements. Furthermore, we will add more code skeletons (PAS), e.g., for Spark, Flink, etc., in our library of skeletons.

6. Evaluation

In this section we discuss the execution performance (i.e., execution time of the ETL workflow) of the sentiment analysis code as a UDF generated by cp-UDF (i.e., parallelizable code). The generated parallelized UDF is executed in a cloud-based distributed environment and a non-parallelizable UDF program executed in a cloud based non-distributed environment.

We created two versions of the sentiment analysis algorithm in order to show two different variants of the same algorithm, which are semantically equivalent.

The first sample variant of a sentiment analysis algorithm, called naïve, is a handwritten custom code and is taken from the work of Cloudera (2016). It counts the number of positive and negative words in a review by comparing them with positive and negative dictionaries loaded into the memory as a part of the initial configuration.[2] For each positive and negative word it increments a respective counter. Finally the sentiment score is calculated by the formula *positivity=good / (good + bad)*. To categorize the user review, each result above a *THRESHOLD* value is classified as *Positive*, otherwise as *Negative*.

The second variant of the algorithm, called CoreNLP, is a long running, computing-intensive program, which uses the Stanford CoreNLP framework (Manning *et al.*, 2014). It provides natural language tools to annotate sentences to indicate parts of speech, named entities, word dependencies, and sentiment.

The idea behind testing different variants in different environments (i.e., a cloud-based distributed environment, pseudo-distributed environment (single node EMR cluster), and a non-distributed and a non-EMR environment) was to prove the following two assumptions:

- Non computing-intensive code (i.e., naïve) normally does not effect the overall performance of an ETL workflow whether it is executed in a distributed or a non-distributed environment. In most cases, the non-compute intensive program has an extra overhead if executed in a distributed environment. Nevertheless, if execution performance is a strict requirement, it will require a lot more resources and

[1] In addition to the method bodies, the parameter types have to be adapted accordingly.

[2] The configuration can also be done using the `setup` method of the `Map` class.

```
1  // #HADOOP_MAP_REDUCE#
2  public class Template {
3      public static class Map {
4          void setup() {
5              // add setup code here
6          }
7          void map(MapKeyType key, MapValueType value, Context context) {
8              // add map code here
9          }
10     }
11     public static class Reduce {
12         void setup() {
13             // add setup code here
14         }
15         public void reduce(ReduceKeyType key, Iterable<ReduceValueType> values, Context context) {
16             // add reduce code here
17         }
18     }
19     void config() {
20         // add configuration code here
21     }
22 }
```

Listing 1. Empty template (PAS) provided to the user.

```
1  style map:hadoop {
2    fragment MethodSlot * if(isMapMethod) {
3      slot KEYTYPE : mapKeyType
4      slot VALUETYPE : mapValueType
5      code:
6        <Method>
7          @Override
8          public void map(#KEYTYPE# key, #VALUETYPE# value, Context context) throws IOException,
               InterruptedException {
9            #INNER#
10         }
11       </Method>
12   }
13   // other fragments
14 }
```

Listing 2. Excerpt from the Hadoop-MapReduce OSS style.

```
1 recipe cpUDF {
2   map:hadoop
3   reduce:hadoop
4   runner:benchmark
5 }
```

Listing 3. OSS recipe used in cp-UDF.

the improvement of execution performance would still be much lower than expected.

- The computing-intensive tasks become a bottleneck in an ETL workflow and must be optimized, because even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow.

To evaluate our approach, we used a BigBench (Ghazal *et al*., 2013) data set on around 20 million product review tuples. Each product has about 3 reviews, and each review consists of approximately 36 words. Following are the details and learning of our experiment.

To carry out the evaluation, we used the M3.xlarge model of EC2 instances, each having similar specifications, i.e., Intel processors with 4 core vCPU and 15 GB RAM. We evaluated the non-parallelizable version of the UDF on the non-distributed EC2 instance with the same system configurations and specifications. We evaluated our MapReduce UDFs on a single node (MR-SN) Amazon Elastic MapReduce (EMR) cluster to depict a pseudo-distributed environment and on a six node EMR cluster (MR-EMR) as a distributed environment. One node served as the master and the rest as core workers. We tested the execution time of both parallel and non-parallel UDFs with different sizes of data sets ranging from one thousand tuples to 20 million tuples of unstructured data. Each test was executed at least five times and the presented results are the average values of those test runs.

The performance execution comparison of a non-parallelizable naïve sentiment analysis program vs. a semantically equivalent parallelizable MapReduce version is shown in Figs. 4 and 5. The graph in Fig. 4 shows (1) the execution time of the non-parallelizable variant of naïve code, (2) its corresponding MapReduce version executed in a single-node (MR-SN) EMR cluster, and (3) the MapReduce version executed in the six node Amazon EMR cluster (MR-EMR).

For a data set ranging from one thousand to less than one million tuples, the naïve non-parallelizable program is more efficient in terms of execution performance as compared to its MapReduce variants (MR-SN and MR-EMR). As the number of tuples increases to more than one million, the execution time of MR-EMR reduces as compared to MR-SN as well as the non-parallelizable program. The graph in Fig. 5 shows that the speedup of MR-EMR is small, i.e., by a factor of two, and the speedup of MR-SN is worse than the non-parallelizable variant because of the Hadoop overhead. The speedup is determined as the execution time of a MapReduce program divided by the execution time of a non-parallelizable program. This proves our first assumption that non computing-intensive code normally does not effect the overall performance of an ETL workflow whether it is executed in a distributed or a non-distributed environment.

Performance execution comparison of a non-parallelizable CoreNLP sentiment analysis program vs. the semantically equivalent parallelizable MapReduce version is shown in Figs. 6 and 7.

Figure 6 shows the execution time of the CoreNLP variant of the sentiment analysis program. The execution time for MR-SN and for the non-parallelizable version is similar. The execution time difference between the MR-EMR and the non-parallelizable program is small for the number of rows less than or equal to 0.1 million. However, as the data size increases, the MR-EMR execution performance increases as the number of Map workers increases and computing intensive tasks are executed in parallel. Figure 7 shows a notable speedup for MR-EMR for 0.5 million tuples and above. However, there is no speedup for MR-SN since there are only two Map workers in a pseudo-distributed environment.

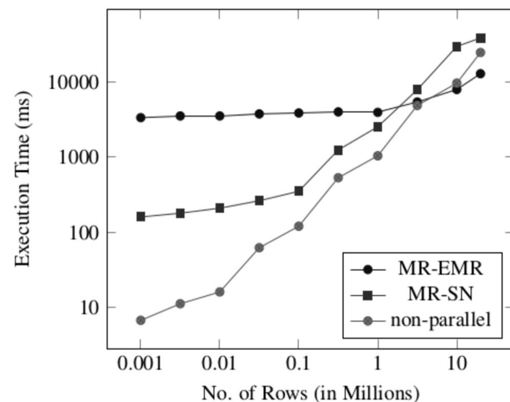

Fig. 4. MapReduce vs. non-parallelizable naïve code execution time.

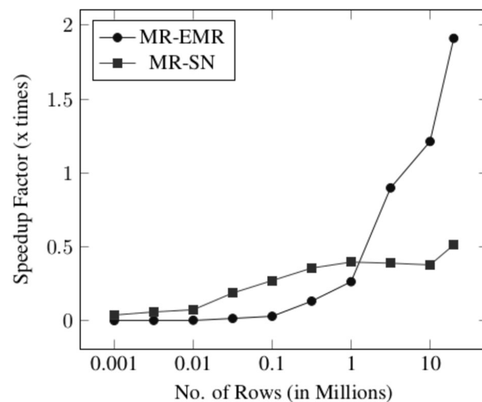

Fig. 5. Parallelizable MapReduce speedup for naïve code.

Hence, the results prove our second assumption that,

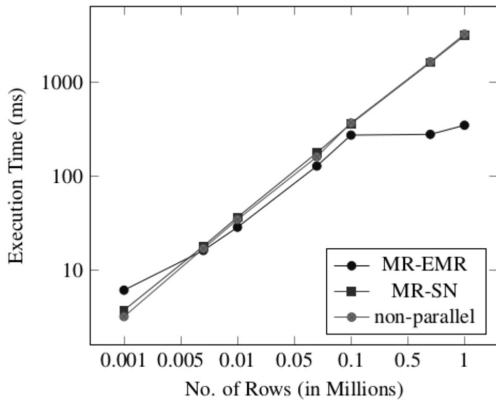

Fig. 6. MapReduce vs. non-parallelizable CoreNLP code execution time.

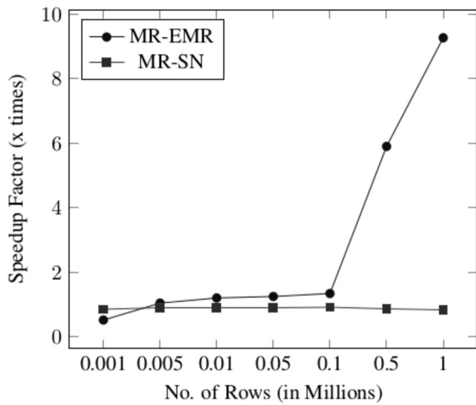

Fig. 7. Parallelizable MapReduce speedup for coreNLP code.

for computing-intensive ETL tasks, even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow.

We also analyzed the time saved by the programmers in developing or designing such UDFs using or not using cp-UDF to address the second contribution of this paper, i.e., code skeletons or design patterns to be used in cp-UDF help to reduce the amount of effort required by the ETL developer in writing complex and efficient programs. Figures 8(a) and 8(b) shows the effort required by the ETL developer in terms of line of code (LOC) to write an efficient parallel sentiment analysis program. The black portion of the figure shows the user-defined code weaved together with the gray portion of the code, which is generated by cp-UDF using the OSS processor. In the naïve version, almost 50% of the code and in case of the CoreNLP version almost 65% of the code is generated automatically by the cp-UDF framework.

Overall, we can observe that cp-UDF significantly reduces the effort required by the ETL developer to write parallel code and ensures error-free code by replacing otherwise manual steps in the parallelization process with automatized semantic analysis. Also, it hides the low level details of parallel execution of a program from the ETL developer, in addition accomplishing considerable speedup without worrying about controlling the costs occurred by processes, communication, and data distribution. The directive code and parameterized attributes make the OSS flexible and extensible, easily adoptable to other use cases with special, user-defined analysis attributes, and even to other languages (e.g, Python, Fortran).

7. Conclusion

This paper is the first step towards a fully automated ETL framework, which overcomes the deficiencies and limitations of existing ETL frameworks as discussed by Ali and Wrembel (2017) as well as Ali (2018). That is, there is a need for an ETL framework that will reduce the work of the ETL developer from a design and performance optimization perspective. The framework should provide recommendations on (1) an efficient design of an ETL workflow according to the business requirements, (2) how and when to improve the performance of an ETL workflow without conceding other quality metrics. Furthermore, there is a need to consolidate and fully support UDFs in an ETL workflow along with traditional ETL operators.

In this paper, we focused on providing a component to assist ETL developers in writing parallelizable UDFs for an ETL workflow. Although most ETL tools provide the functionality to write custom code as UDFs, a UDF to transform large and partially structured data can be very complex and may become a performance bottleneck if not implemented optimally. One of the possible ways to implement an efficient program is parallelizing its execution, which requires expertise as well as deep understanding and knowledge of parallel and distributed programming. To provide the ETL developer with out-of-the-box functionality in an ETL framework to write efficient parallel custom programs, we proposed cp-UDF, which separates the parallelization concern from the development of UDF activities for data-intensive ETL workflows. Currently, cp-UDF supports Hadoop as a distributed framework to execute UDFs in a parallel environment. However, it is extensible and can be integrated with other parallel and distributed frameworks, e.g., Flink and HPC clusters.

On top of that, we showed that cp-UDF provides an easy, fast, and flexible way for the ETL developer to write efficient and error-free parallel programs for data-intensive tasks.

In the future, we will extend our framework by introducing more programming paradigms and data processing engines, e.g., Flink, Spark, Hive, Impala and HPC clusters, where an ETL developer can chose from

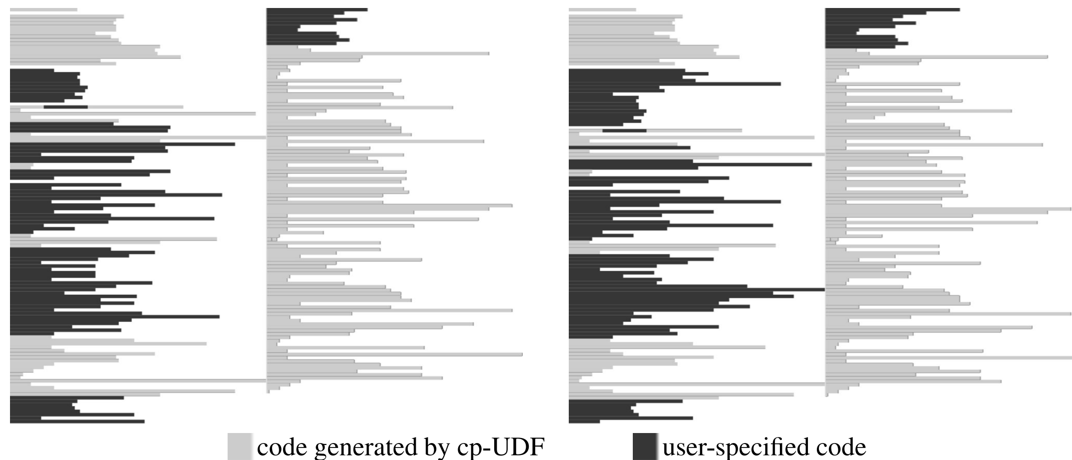

Fig. 8. Estimate of effort required by the ETL developer to write a MapReduce program using an OSS in terms of LOC: naïve version (a), CoreNLP version (b).

the engine specific PAS in order to achieve the desired degree of performance. Furthermore, we will extend our approach by developing an automated ETL framework, which allows the ETL developer to utilize cp-UDF as an integrated component. The ETL framework will generate the more efficient solution for a particular ETL scenario based on execution and monetary cost constraints. We will also provide a cost model to choose an optimal variant of the UDF generated by cp-UDF.

Acknowledgment

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate *Information Technologies for Business Intelligence Doctoral College* (IT4BI-DC) and trivago N.V.

References

Ali, S.M.F. (2018). Next-generation ETL framework to address the challenges posed by big data, *Workshop Proceedings of the EDBT/ICDT Joint Conference, Vienna, Austria*.

Ali, S.M.F. and Wrembel, R. (2017). From conceptual design to performance optimization of ETL workflows: Current state of research and open problems, *The VLDB Journal* **26**(6): 1–25.

Aßmann, U. (2003). Invasive software composition, *Invasive Software Composition*, Springer, Berlin/Heidelberg, pp. 107–145.

Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V. and Warneke, D. (2010). Nephele/PACTs: A programming model and execution framework for web-scale analytical processing, *Proceedings of the Symposium on Cloud Computing, Indianapolis, IN, USA*, pp. 119–130.

Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S. and Zhou, J. (2008). Scope: Easy and efficient parallel processing of massive data sets, *Proceedings of the VLDB Endowment* **1**(2): 1265–1276.

Cloudera (2016). Example: Sentiment analysis using MapReduce custom counters, `https://www.cloudera.com/documentation/other/tutorial/CDH5/topics/ht_example_4_sentiment_analysis.html`.

Dagum, L. and Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming, *IEEE Computational Science and Engineering* **5**(1): 46–55.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters, *Communications of the ACM* **51**(1) 107–113.

Ekman, T. and Hedin, G. (2007). The JastAdd system modular extensible compiler construction, *Science of Computer Programming* **69**(1–3): 14–26.

Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A. and Jacobsen, H.-A. (2013). Bigbench: Towards an industry standard benchmark for big data analytics, *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA*, pp. 1197–1208.

González-Vélez, H. and Kontagora, M. (2011). Performance evaluation of MapReduce using full virtualisation on a departmental cloud, *International Journal of Applied Mathematics and Computer Science* **21**(2): 275–284, DOI: 10.2478/v10006-011-0020-3.

Große, P., May, N. and Lehner, W. (2014). A study of partitioning and parallel UDF execution with the SAP HANA database, *Proceedings of the 26th International Conference on Scientific and Statistical Database Management, Aalborg, Denmark*, p. 36.

Hedin, G. (2000). Reference attributed grammars, *Informatica (Slovenia)* **24**(3): 301–317.

Karagiannis, A., Vassiliadis, P. and Simitsis, A. (2013). Scheduling strategies for efficient ETL execution, *Information Systems* **38**(6): 927–945.

Karol, S. (2015). *Well-formed and Scalable Invasive Software Composition*, PhD dissertation, Technische Universitat Dresden, Dresden.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997). Aspect-oriented programming, *in* M. Akşit and S. Matsuoka (Eds.), *European Conference on Object-oriented Programming*, Springer, Berlin/Heidelberg, pp. 220–242.

Kumar, N. and Kumar, P.S. (2010). An efficient heuristic for logical optimization of ETL workflows, *International Workshop on Business Intelligence for the Real-Time Enterprise, Singapore, Singapore*, pp. 68–83.

Liu, X., Thomsen, C. and Pedersen, T.B. (2013). ETLMR: A highly scalable dimensional etl framework based on MaprEduce, *in* A. Hameurlain *et al.* (Eds.), *Transactions on Large-Scale Data-and Knowledge-Centered Systems VIII*, Springer, Berlin/Heidelberg, pp. 1–31.

Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Baltimore, MD, USA*, pp. 55–60.

Mey, J., Karol, S., Aßmann, U., Huismann, I., Stiller, J. and Fröhlich, J. (2016). Using semantics-aware composition and weaving for multi-variant progressive parallelization, *Procedia Computer Science* **80**: 1554–1565.

Nambiar, R.O. and Poess, M. (2006). The making of TPC-DS, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea*, pp. 1049–1058.

Simitsis, A., Vassiliadis, P. and Sellis, T. (2005). State-space optimization of ETL workflows, *IEEE Transactions on Knowledge and Data Engineering* **17**(10): 1404–1419.

Simitsis, A., Wilkinson, K., Dayal, U. and Castellanos, M. (2010). Optimizing ETL workflows for fault-tolerance, *IEEE 26th International Conference on Data Engineering (ICDE), Long Beach, CA, USA*, pp. 385–396.

Thomsen, C. and Pedersen, T.B. (2011). Easy and effective parallel programmable ETL, *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP, New York, NY, USA*, pp. 37–44.

Tziovara, V., Vassiliadis, P. and Simitsis, A. (2007). Deciding the physical implementation of ETL workflows, *Proceedings of the International Workshop on Data Warehousing and OLAP, New York, NY, USA*, pp. 49–56.

Vassiliadis, P., Simitsis, A. and Baikousi, E. (2009). A taxonomy of ETL activities, *Proceedings of the ACM 12th International Workshop on Data Warehousing and OLAP, New York, NY, USA*, pp. 25–32.

Weinberg, A.I. and Last, M. (2017). Interpretable decision-tree induction in a big data parallel framework, *International Journal of Applied Mathematics and Computer Science* **27**(4): 737–748, DOI: 10.1515/amcs-2017-0051.

Syed Muhammad Fawad Ali received his MSc degree from the Lahore University of Management Sciences, Pakistan, in 2012. He is a technical lead data engineer at trivago N.V., Leipzig, Germany, and a PhD candidate at the Poznań University of Technology, Poland. His research interests include ETL performance optimization and user-defined functions in the domain of ETL and big data.

Johannes Mey is a PhD student at the Chair of Software Technology at TU Dresden. His research interests are in the areas of component-based software engineering and the modelling of adaptive systems, for both of which he employs attribute grammar-based approaches.

Maik Thiele is a postdoc researcher at the Database Systems Group at TU Dresden, where he finished his dissertation on quality-driven data production controlling in real-time DW systems in 2010 and received his doctorate with distinction. He has been a visiting scientist at UBS Zurich, GfK Nuremberg, and HP Labs Palo Alto. His research interests include large-scale data processing, information extraction and data integration.

Received: 4 March 2018
Revised: 2 October 2018
Accepted: 10 December 2018