Data preparation as a service based on Apache Spark

Nivethika Mahasivam, Nikolay Nikolov, Dina Sukhobok and Dumitru Roman

SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
nivemaham@gmail.com,
{nikolay.nikolov,dina.sukhobok,dumitru.roman}@sintef.no

Abstract. Data preparation is the process of collecting, cleaning and consolidating raw datasets into cleaned data of certain quality. It is an important aspect in almost every data analysis process, and yet it remains tedious and time-consuming. The complexity of the process is further increased by the recent tendency to derive knowledge from very large datasets. Existing data preparation tools provide limited capabilities to effectively process such large volumes of data. On the other hand, frameworks and software libraries that do address the requirements of big data, require expert knowledge in various technical areas. In this paper, we propose a dynamic, service-based, scalable data preparation approach that aims to solve the challenges in data preparation on a large scale, while retaining the accessibility and flexibility provided by data preparation tools. Furthermore, we describe its implementation and integration with an existing framework for data preparation – Grafterizer. Our solution is based on Apache Spark, and exposes application programming interfaces (APIs) to integrate with external tools. Finally, we present experimental results that demonstrate the improvements to the scalability of Grafterizer.

Keywords: Distributed Data Parallel Processing, Apache Spark, Big data preparation, Interactive data preparation

1 Introduction

The movement towards digitalization has spread in prominent domains such as health, industrial production, defense, and banking, to improve operations using data-driven decisions. Such domains deploy various tools including sensors, applications, logging and production databases to collect data in high velocity and large volumes, and extract relevant information in tabular or text formats [1]. This process produces raw data, often semi-structured or unstructured, that could contain missing, erroneous, incomplete, and duplicate values. The raw data needs to be cleaned and transformed into structured data to meet the expected quality for further usage. Data preparation is an important step to treat "dirty" datasets by collecting, combining and consolidating datasets that are suitable for further data analysis. Despite of the importance of data preparation, it remains a tedious and time-consuming process that requires significant effort [2, 3].

Furthermore, data preparation in the context of big data introduces even more challenges, both functional and nonfunctional.

One of the main challenges related to large volume data preparation is that existing frameworks and tools require expert knowledge of specific programming languages, data models, and computational models. Examples include native language libraries such as Pandas in Python [4] or Data Frame in the R language[1], which are widely used for data wrangling and preparation. Further, distributed data parallelization (DDP) is the most widely realized computational model in big data processing [5]. The basic computational abstraction of DDP performs a computation in parallel, by distributing smaller data partitions among a set of machines or processes. It provides scalability, load balancing and fault tolerance. Frameworks that realise DDP, such as Apache Hadoop and Apache Spark, are used to implement scalable solutions in the big data domain, and significantly increase technical complexity of data workers' data preparation routines. Implementation of a solution based on the DDP model should consume a large dataset, split it into partitions, and process and accumulate the results without losing the semantics of the expected outcome.

A scalable data preparation tool or framework is essential to process large volumes of data, but existing solutions lack capabilities to effectively perform such operations. Due to their architecture, it is difficult to realize large-scale data processing techniques including distributed and concurrent processing. Such solutions include spreadsheet tools (e.g. Excel and Open Office) that are often used to prepare datasets, especially in companies that lack the expertise to make use of the frameworks and tools for big data preparation and processing [6]. Data preparation is often implemented as an iterative process where new data quality issues can appear while existing ones are being addressed. Consequent iterations are performed by reviewing the intermediate results produced from previous iterations. Existing data preparation tools support interactive preparation by providing immediate rendering of intermediate results and suggestions for improvements. OpenRefine[2], Pentaho Data Integration (Kettle)[3], and Trifacta Wrangler[4] are examples of interactive solutions that are used in industries to accelerate the data preparation process. Such solutions are primarily desktop-oriented applications that are not dedicated to handling large amounts of data.

In this paper, we propose a solution that addresses the scalability, usability and accessibility issues in data preparation through a mixed approach that combines interactivity of data preparation tools with the powerful features of frameworks that support data preparation for big data. Our solution is based on extensions to the web-based Grafterizer transformation framework [7] that is part of the DataGraft platform [8, 9, 10][5]. We propose improvements to Graftwerk – the existing data transformation back-end of Grafterizer, with the aim to augment its capability to effectively process larger

[1] http://www.r-tutor.com/r-introduction/data-frame
[2] http://openrefine.org/
[3] http://community.pentaho.com/projects/data-integration/
[4] https://www.trifacta.com/products/wrangler/
[5] https://datagraft.io/

datasets. Our solution is a scalable data preparation as a service back-end that can dynamically build data preparation pipelines, and effectively support interactive cleaning and transformation of large volumes of data using DDP techniques.

The contributions of this paper are thereby two-fold:

1. First, we describe an approach for using DDP for scalable data preparation, based on the use of Apache Spark, which has been identified as a suitable framework that facilitates scalable data preparation.
2. Second, we propose a proof-of-concept realization of the approach for data preparation as a service in Grafterizer, along with validation and evaluation results that demonstrate the difference in performance between our proposed approach and the existing back-end.

The remainder of this paper is organized as follows. In Section 2, we provide a detailed description of our approach for DDP-based data preparation. In Section 3, we describe a proof-of-concept data preparation as a service back-end that realizes our proposed approach. Section 4 shows the performance of the implemented solution in experiments with large volumes of data. Section 5 discusses related works. Finally, Section 6 summarizes this paper, and outlines further development directions of the approach and implementation.

2 A proposed approach for DDP-based data preparation

2.1 DDP for data preparation

Distributed Data Parallelization is a computational model, proven to effectively perform large scale data processing [5]. MapReduce is a popular implementation of the DDP model that has been adapted in many big data analytic tools [11]. Following the spike of MapReduce, other programming models were introduced, including in-memory computing [5] and iterative map-reduce [12]. In-memory computing is a general-purpose solution for large-scale computing problems. It is a special form of DDP, where a computation is performed on a read-only representation of data in memory. The memory representation is held on a distributed-shared-memory (DSM) that is composed of memory distributed across several machines. Since data preparation processes are iterative, an in-memory computing model is suitable for the implementation of such processes. This approach keeps intermediate results in-memory and avoids high disk input/output latency, unlike MapReduce frameworks. In the following paragraphs, we discuss some of the most important characteristics of designing a data preparation approach using DDP.

Handling data parallelism
Parallelism is an important aspect of DDP. We aim to achieve parallelism in two ways:

- *Data-flow parallelism*: In data-flow parallelism, data is chunked into several partitions and distributed to facilitate ingestion of multiple processors in parallel. This enables processing large datasets that cannot be processed using traditional computational models. To achieve data parallelism, a computational operation should be adapted to be executed in parallel and to use partitioned data. Subsequently, the output data from each partition needs to be aggregated according to the semantics of the operation to produce the final result. DDP operations such as *map*, *reduce*, *group*, *match* are all examples of existing DDP patterns [11].

- *Control-flow parallelism*: Control-flow parallelism is achieved by parallel execution of multiple operations that do not depend on other operations' results [13]. As mentioned earlier, data preparation is an iterative process where the data preparation operations are performed in a pipeline. **Fig. 1** exemplifies control-flow parallelism optimization on such a pipeline. Suppose that operations p1 and p2 in **Fig. 1.**a operate independently on two parts of a dataset (i.e. they do not rely on their respective results). With control-flow parallelism, the total execution time of the control sequence *t1* of the given operation flow can be shortened to *t2* by executing them in parallel, as shown in **Fig. 1.**b. Control-flow parallelism provides high throughput of execution time and optimises resource allocation.

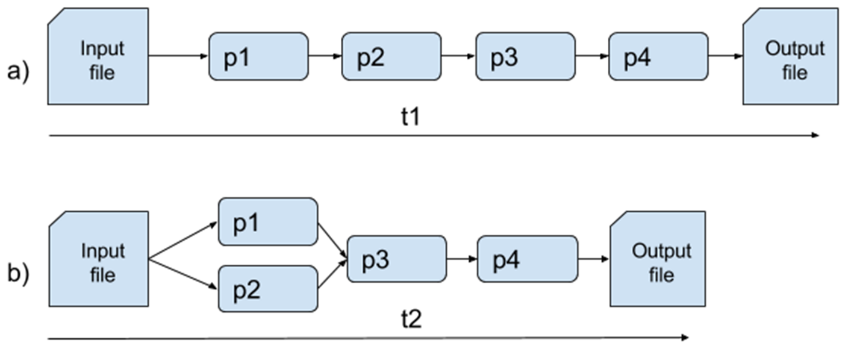


Fig. 1. Control-flow parallelization of a pipeline

Our proposed approach for DDP in data preparation is to use a combination of data-flow and control-flow parallelism using DDP. Data-flow parallelism is essential to enable large-scale data preparation that can scale horizontally. Control-flow parallelism can be used to improve the throughput by executing dynamically adapted pipelines that utilize optimal resources on data partitions.

Handling data ingestion

We want to allow cleaning and transformation of collected raw data that are available in various volumes and formats. A position article [14] shows that 80% of the published datasets are in tabular format. Hence, in our solution we aim to process datasets that are already collected and made available in common tabular formats such as CSV, TSV, Excel and other spreadsheet formats.

Data ingestion is a routine for receiving input data for data preparation processes. The most frequently used data ingestion techniques in big data context are stream processing, micro-batch processing and batch processing [15]. Batch processing techniques treat input data as a complete collection that needs to be considered for analysis [16]. Batch processing is generally used as "store-first, process second" model, where data is collected in advance and made available to be processed. In stream processing, the data is ingested as continuous, long running streams [17]. Data streams are produced based on event-by-event or the complex-event-model. Data streams are used for real-time analysis but are less appropriate when it comes to processing tabular formats of data. Micro-batch processing combines aspects of both batch processing and stream processing by treating data as a sequence of small batches. Streaming or micro-batching data can be a solution to overcome the problem of processing large data. However, since we aim to perform ETL operations on tabular data, these approaches assume data to be homogenous, and will not always result in the expected output, especially when performing collective operations or row-dependent operations. Hence, we focus on support for batch processing of input data since the approach is more suitable for processing the targeted input datasets.

2.2 Apache Spark as a framework for DDP-based data preparation

According to [18], the biggest challenge faced in data preparation solutions for big data is providing a processing model that can do complex reasoning of small volumes of data, simple processing of large volumes of data, and parallel processing of very large volumes of data. In [18], the authors argue that a feasible big data preparation tool should provide both expressiveness and scalability of functionalities. Thereby, we compared available highly expressive and scalable solutions. Native libraries such as Pandas in Python or Data Frame in the R language are widely used in data preparation implementations. However, deploying these frameworks as a scalable solution in a distributed environment is challenging since no native support for this type of deployment currently exists. On the other hand, big data frameworks provide native tools to deploy an application in a distributed environment out of the box. Among big data frameworks Apache Hadoop and Apache Spark are used to implement scalable, batch-processing solutions. Apache Spark has been selected for the implementation of our solution since experiments show that Spark has better performance than Hadoop for big data analytics [19].

Apache Spark is a general-purpose, in-memory data processing framework that realizes DDP. It runs on master-slave architecture, which can scale out with additional master and/or slave nodes. One of the advantages of Spark is its computational abstraction called *resilient distributed datasets (RDDs)* [20]. RDDs are immutable collections of objects that are partitioned across different Spark nodes in the network. It represents the data in-memory on a DSM to enable data flow parallelism and support batch-processing of large volumes of data.

Transformations in Spark are operations that do not depend on inputs from other partitions such as map, filter, rename. An RDD is transformed into another RDD when a transformation is executed. Spark implements the notion of lineage for RDDs to keep the information of how a newer RDD is derived from parent RDD. When a partition is lost, Spark rebuilds the partition from stored data to facilitate e□cient fault-tolerance to the system and implement data-flow parallelism.

Furthermore, Spark benefits from lazy execution of transformations to create a directed acyclic-graph (DAG) of data and transformations instead of applying transformations immediately. Once the graph is followed by an action, Spark executes the formed DAG by distributing it as several tasks among nodes. An action in Spark is an operation that reduces the output from all partitions into a final value such as *reduce*. Lazy execution of transformations is used to further optimize the operations in accordance with the control-flow. As an example, if a user wants to apply two independent filters on two different parts of a dataset, and a global operation on the entire data, Spark would optimize by jointly (as opposed to sequentially) applying the independent filters without constructing the intermediate dataset after each filter. This optimization benefits the evaluation of several consecutive RDDs, and provides efficient implementation of iterative execution of the pipeline.

In addition, *DataFrame* is an extended model of RDD in the SparkSQL package, which organizes data into named columns, conceptually equivalent to a relational database structure [19]. It provides a domain-specific language for relational operations, including select, filter, join, groupBy, and enables users to perform SQL-like queries on *DataFrames*. By extending these APIs, we provide an expressive data preparation framework that accommodates most of the data cleaning operations in relatively less complex APIs. Finally, SparkSQL extends a novel optimizer called *Catalyst* [19], that implements query analysis, logical optimization and physical planning. Catalyst supports both rule-based and cost-based optimization of relational operations. Generating optimized queries based on the features in Catalyst allows us to indirectly realize relational query optimization and control-flow parallelization. Therefore, in our solution, a pipeline created using DataFrame in Spark will be optimized before it is executed using Catalyst.

3 Realization of DDP-based data preparation approach in Grafterizer

In this section, we introduce a proof-of-concept implementation of the approach to provide a scalable, dynamic data preparation service using Spark. The service is deployed as a data preparation back-end for Grafterizer that processes, cleans and transforms large volumes of tabular data. Currently, the implementation supports CSV, TSV, and flat-JSON files, and provides rich, procedural APIs for data preparation operations that can be easily used to build pipelines on Spark. Our solution allows for dynamic creation of pipelines, which is the ability to create and/or modify a data preparation pipeline during run-time. This enables the execution of the data preparation process interactively, so that users can perform incremental modifications of the data cleaning pipeline and observe the results.

3.1 Architecture

The high-level architecture of the service is illustrated in Fig. 2. Below, we describe each component and its functional and technical contributions in more detail.

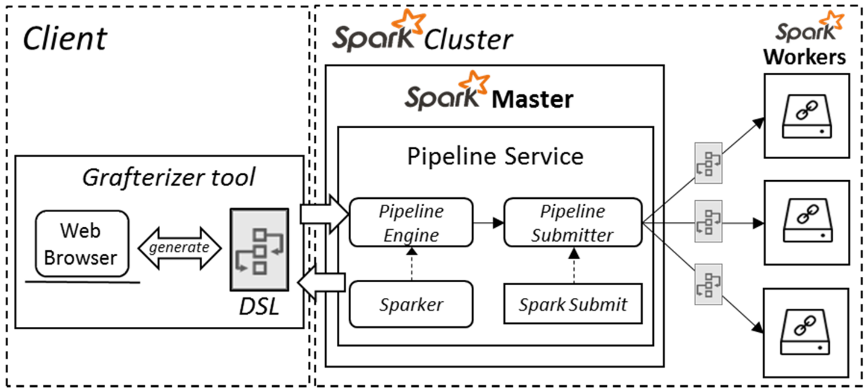


Fig. 2. High-level architecture of the service

Sparker

Sparker is a component that provides APIs to enable data preparation pipelines as Spark jobs. It implements data preparation operations [21] using DataFrame and RDD APIs, and provides pattern-oriented procedural APIs that can be used to build a pipeline. Sparker encapsulates the implementation complexity of data cleaning operations in Spark. It provides commonly used APIs in data preparation operations that are not already available in DataFrame. Especially reshape functions merging grouped data, concatenating multiple columns, splitting of a given column using a separator into multiple columns, custom group functions, filling missing values, custom query functions with simple filters, pagination of queried data and adding additional rows of values to a dataset are notable features. To adapt a pipeline to perform data preparation, we designed every API to receive DataFrame objects with additional parameters depending on the operation's semantics, and return the result as an altered DataFrame object according to the pipeline pattern. This allows us to create a chain of operations by using the output of any operation as an input to another operation. The main categories of Sparker's APIs are: 1) converting input files to DataFrame; 2) data cleaning/preparation operations; 3) converting DataFrame into suitable output format as shown in **Fig. 3**. A standard pipeline would start with the conversion of an input file, one or more data cleaning/preparation operations, and finally the generation of the final result as output.

Pipeline Service

The *Pipeline Service* provides dynamic creation of pipelines using Sparker APIs, and submits the current version of the pipeline to the Spark cluster. The *Pipeline Service* has two main components: The *Pipeline Generator* and the *Pipeline Submitter*. The

Pipeline Generator initially receives requests with input data and corresponding pipeline instructions, and dynamically generates pipelines using the requested information. The pipeline is implemented in Clojure – a dynamic programming language that allows the creation, modification and execution of programming instructions during runtime.

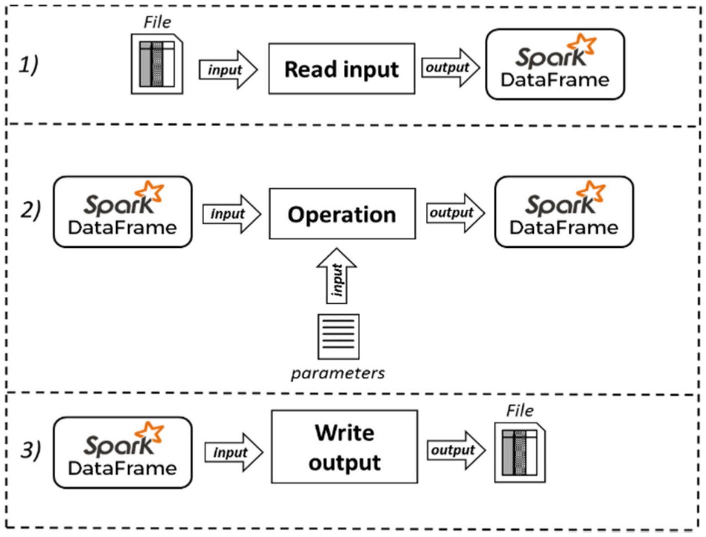
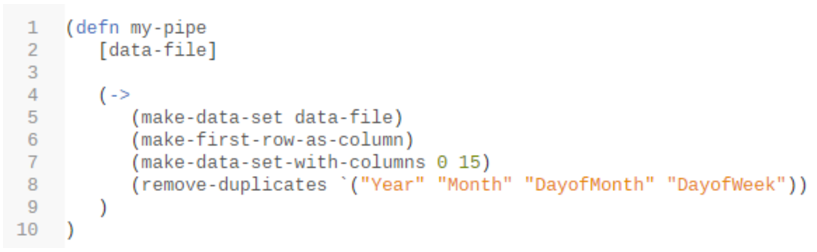


Fig. 3. Pipeline pattern implementation using DataFrames

Furthermore, pure functions in Clojure transform immutable data structures into some output format. We use the thread-first macro, which is denoted by "->" in a pipeline. Taking an initial value as its first argument, the macro threads it through one or more expressions, thus constituting a pipeline. A sample pipeline using *Pipeline Service*'s APIs is depicted in **Fig. 4**.



```
(defn my-pipe
   [data-file]

   (->
      (make-data-set data-file)
      (make-first-row-as-column)
      (make-data-set-with-columns 0 15)
      (remove-duplicates `("Year" "Month" "DayofMonth" "DayofWeek"))
   )
)
```

Fig. 4. A sample pipeline using Pipeline Service APIs realizes pipeline pattern

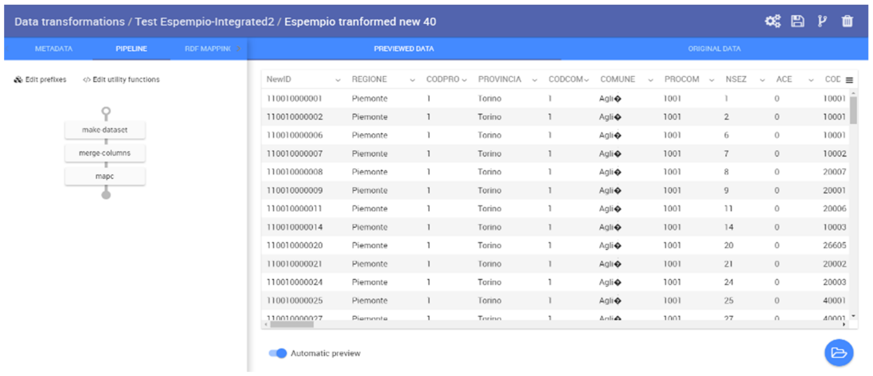
The Clojure function names are used as APIs in the Pipeline Service to enable creation and execution of Sparker pipeline instructions during run-time. Once a pipeline is created, the *Pipeline Generator* forwards the created pipeline to the *Pipeline Submitter*. The *Pipeline Submitter* submits given pipeline instructions as a Spark job using *spark-*

submit. *spark-submit* is a script provided by Apache Spark that submits any given programming instruction created using Spark as a Spark job, which can then be executed on a Spark cluster. Once the job is executed *Pipeline Service* sends a response with the processed data.

These APIs are used to create a pipeline of data preparation operations in client applications. Fig. 4 shows a sample pipeline that uses the Pipeline Service APIs and demonstrates how the pipeline pattern was implemented. Rows 1 and 2 define the pipeline function and the data input parameter to the pipeline through which we pass the dataset as argument. This is followed by the thread-first operator in line 4. Lines 5 through 8 are calls to the APIs exposed through the Pipeline Service. They consecutively create the data structure (DataFrame) out of the input data, take the first row as table headers, take the first 16 columns of data (discarding the rest), and, finally, filter out duplicate rows based on a vector of unique values of the cells in each row that correspond to the columns "Year", "Month", "DayofMonth" and "DayofWeek".

Grafterizer

Grafterizer has been integrated to generate pipelines using calls based on the DSL/APIs provided by the *Pipeline Service* during user interaction. Once a pipeline is created/altered an HTTP request is sent to the pipeline service with the metadata of input and the generated pipeline using the DSL/APIs. The pipeline is then executed by the Spark-based back-end and the resulting data/output is sent back as an HTTP response to Grafterizer and immediately previewed. The back-end service proposed in this paper has been integrated with the currently available user interface of Grafterizer, which is shown in **Fig. 5** (the left part representing an example of data transformation pipeline and the table depicting the data on which the pipeline executes).



NewID	REGIONE	CODPRO	PROVINCIA	CODCOM	COMUNE	PROCOM	NSEZ	ACE	COD
110010000001	Piemonte	1	Torino	1	Agli�	1001	1	0	10001
110010000002	Piemonte	1	Torino	1	Agli�	1001	2	0	10001
110010000006	Piemonte	1	Torino	1	Agli�	1001	6	0	10001
110010000007	Piemonte	1	Torino	1	Agli�	1001	7	0	10002
110010000008	Piemonte	1	Torino	1	Agli�	1001	8	0	20007
110010000009	Piemonte	1	Torino	1	Agli�	1001	9	0	20001
110010000011	Piemonte	1	Torino	1	Agli�	1001	11	0	20006
110010000014	Piemonte	1	Torino	1	Agli�	1001	14	0	10003
110010000020	Piemonte	1	Torino	1	Agli�	1001	20	0	26605
110010000021	Piemonte	1	Torino	1	Agli�	1001	21	0	20002
110010000024	Piemonte	1	Torino	1	Agli�	1001	24	0	20003
110010000025	Piemonte	1	Torino	1	Agli�	1001	25	0	40001
110010000027	Piemonte	1	Torino	1	Agli�	1001	27	0	40001

Fig. 5. Grafterizer's preview of cleaned data using current pipeline

4 Evaluation

We conducted a set of experiments to evaluate the performance of the proposed service. The experiments were conducted on a cluster consisting of a master node with Intel

Core i7 3.3GHz, 4 CPU Cores, 15 GB of RAM, 512GB of SSD, running on Ubuntu 14.04 LTS and 4 worker nodes each having Intel Core i7 3.3GHz, 12 CPU Cores, 64 GB of RAM, 480GB SSD and running on CentOS 7. A Spark executor is a process that serves a Spark application, and typically runs on a worker node. A Spark driver is a process that coordinates the execution of a Spark job on worker nodes. A Spark application can scale-out with more executors and scale-up with more concurrent tasks assigned to a process.

To analyze the scalability of the proposed solution, we tuned the Spark cluster with different numbers of executors to study its ability to scale out. By increasing the number of executors, we multiplied the memory allocated to process input, CPUs and concurrent tasks per executor. In this experiment, we used the Price Paid Data (PPD)[6] dataset which is approximately 3.5 GB. We created a sample data pipeline to load the input data, use the first-row as a header row for the schema and then use filters on selected columns followed by grouping of data. We executed the same pipeline on the given input dataset multiple times with varying number of executors, and recorded the execution times of the first request sent to the cluster once a cluster setup was initialized. In addition, we recorded the average of the three following requests sent to that cluster setup, because of the significant difference between the execution time of the first and any following requests. The difference is due to overhead of distributing dependent source files to worker nodes from drivers, which is done only when the first request is executed. The result of the experiment is illustrated in **Fig. 6**.

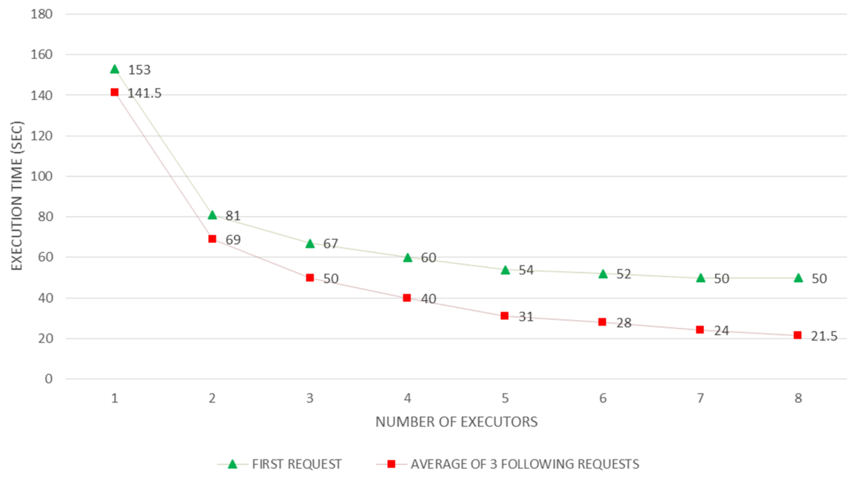


Fig. 6. Execution time of Pipeline Service with increasing number of executors

The experiment clearly shows that the execution time is gradually decreasing. This validates that the performance of the service increases for a given input when the cluster scales-out with more executors. Further, this proves that the proposed service can process large volumes of data and scale out with more executors. The service can scale out

[6] https://www.gov.uk/government/statistical-data-sets/price-paid-data-downloads

with the size of input data, since it is based on in-memory computation, and adding more computing resources increases the capability of processing larger volumes of data.

Further, we benchmarked the performance of Grafterizer with the proposed service on a single host compared to original Grafterizer with traditional back-end to measure the improvement with respect to the current system. Due to the limitations in the ability to process large volumes of data by existing system, we created input datasets by sampling the UK Road Accidents Safety Data in different sizes. We created inputs that increase by approximately 10 MB from 10 MB to 100 MB. Experiments were performed on a computer with Intel Core i5 2.5GHz processor with 4 CPU Cores, and 8GB of RAM. Equivalent pipelines that can be executed by each system were created, and the execution time for each input of every system were recorded. The results of the experiment are depicted in **Fig. 7**.

The graph clearly shows that the proposed service is almost four times faster than the existing Grafterizer back-end on a single node deployment. Further, the existing Grafterizer back-end was not able to effectively process data bigger than 75 MB on the test hardware, whereas the proposed service could easily process larger input in a short time. This shows that the new service has significantly improved Grafterizer's performance and capacity to process large data even as a single-node deployment.

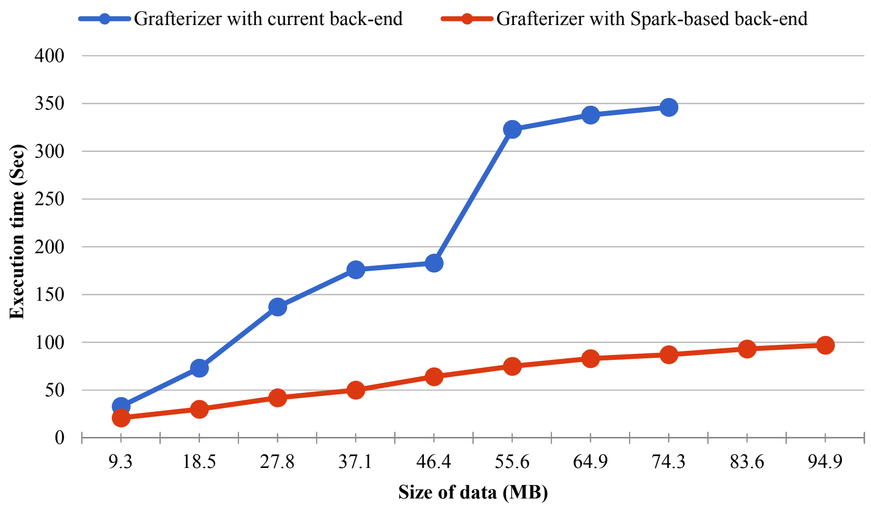


Fig. 7. Performance evaluation of the service on a single-node

5 Related work

The solution proposed in this paper is an alternative to the existing Grafterizer back-end: it enables Grafterizer to work with larger volumes of data by integrating a scalable-backend system that can effectively scale out on a distributed environment with the size of data, and efficiently execute data preparation pipelines. This solution eliminates the dependencies and complexities of implementing and executing a scalable data preparation pipeline in Grafterizer.

Furthermore, the work presented in this paper is related to scalable data processing systems, and data cleaning and transformation tools for big data. In the following we discuss the most relevant recent works in these areas, pointing out the main differences between existing solutions and our proposed approach.

SparkGalaxy [22] is a big data processing toolkit designed to perform complex experiments using data mining and analysis for large amounts of bio-medical data. SparkGalaxy uses Apache Spark's RDD and Graph features to represent data and workflows in a distributed fashion. SparkGalaxy follows a similar methodology to our proposed solution to solve scalability problems. SparkGalaxy focuses on enabling large-scale, workflow-based data mining of biomedical data whereas our solution focuses on enabling a general purpose scalable data preparation tool. Our solution is provided as a service and can be used by other client systems than Grafterizer, using HTTP requests. SparkGalaxy was not designed to be a solution provided as a service. On the other hand, compared to SparkGalaxy, our solution does not directly support integration of machine learning algorithms.

Cleanix [23] is a prototype system for cleaning relational big data. It ingests data from multiple sources and cleans them on a shared-nothing cluster. The backend system of Cleanix is built on top of an extensible and flexible data-parallel framework called Hyracks. A key difference is that our solution is based on Apache Spark. Compared to Spark, Hyracks does not support iterative algorithms and is not an in-memory computing framework [24], making Apache Spark more attractive for data cleaning. Nevertheless, we are not aware of any studies that directly compares Spark and Hyracks performance. Furthermore, Cleanix provides data cleaning operations mainly in four categories of operations (value detection, incomplete data filling, data deduplication and conflict resolution), while our solution supports expressive APIs to perform Cleanix's four types of operations, as well as other operations such as data reshaping and grouping. Furthermore, one could argue that our solution is more user-friendly compared to Cleanix since the data cleaning workflow is supported by graphical interactive previews, and data upload through an intuitive graphical drag-and-drop component.

OpenRefine[7] is an open-source tool for data cleaning/transformation and integration, and provides interactive user-interfaces with spreadsheet style interactions to easily support data cleaning, and previews similar to Grafterizer. OpenRefine was designed as a desktop application rather than a service. It is a memory-intensive tool that runs on a desktop system which limits the size of data that can be processed. There are attempts to extend OpenRefine to support large data processing, e.g., BatchRefine[8] and OpenRefine-HD[9]. OpenRefine-HD extends OpenRefine to use Hadoop's MapReduce jobs on HDFS clusters. However, Apache Spark is considered faster for iterative data prep-

[7] http://openrefine.org/
[8] https://github.com/fusepoolP3/p3-batchrefine
[9] https://github.com/rmalla1/OpenRefine-HD

aration process [23]. Such OpenRefine extensions require manual execution of transformation in a distributed environment whereas our solution eliminates such overhead by integrating it with Grafterizer in an automated workflow.

6 Summary and outlook

In this paper, we proposed a data preparation as a service solution that addresses the scalability, usability and accessibility issues in data preparation. We proposed an approach for using DDP for scalable data preparation, based on the use of Apache Spark, and presented a proof-of-concept realization of the approach in Grafterizer, along with validation and evaluation results that demonstrate the difference in performance in data preparation between our proposed approach and the existing back-end of Grafterizer. Experiments show that the proposed implementation scales out with more executors, and performs better than the existing Grafterizer back-end on a single-node deployment. It is worth mentioning that the functional benefits of the proposed solution include user-friendliness, flexibility and ease of use for users with moderate technical skills. Overall, the service is effective and efficient for large-scale data preparation.

As part of future work, we are considering extending the proposed solution to support various data formats as input for data preparation, and operationalize it for the production environment of DataGraft.

Acknowledgements. The work in this paper is partly supported by the EC funded projects proDataMarket (Grant number: 644497), euBusinessGraph (Grant number: 732003), and EW-Shopp (Grant number: 732590). The authors would like to thank Bjørn Marius von Zernichow for his help in improving the readability of the camera-ready version of the paper.

References

1. M. Atzmueller, S. Oussena and T. Roth-Berghofe, "Data Preparation for Big Data Analytics: Methods and Experiences.," in *Enterprise Big Data Engineering, Analytics, and Management*, IGI Global, 2016, pp. 157-170.
2. S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. V. Ham, N. H. Riche and P. Buono, "Research directions on data wrangling: Visualizations and transformations," *Information Visualization,* pp. 271-288, 2011.
3. S. Krishnan, M. J. Franklin, K. Goldberg and E. Wu, "ActiveClean: An Interactive Data Cleaning Framework For Modern Machine Learning," in *International Conference on Management of Data*, San Francisco, California, USA: ACM, 2016.
4. W. McKinney, "pandas: a Foundational Python Library for DataAnalysis and Statistics," *NEM (Networked & Electronic Media),* 2011.
5. C. J. Jackson, V. Vijayakumar, A. M. Quadir and C. Bharathi, "Survey on Programming Models and Environments for Cluster Cloud, and Grid Computing that defends Big Data," *Procedia Computer Science (2nd International Symposium on Big Data and Cloud Computing (ISBCC'15),* p. 517–523, 2015.

6. S. Kandel, A. Paepcke, J. Hellerstein and J. Heer, "Enterprise data analysis and visualization: An interview study.," *IEEE Trans. Vis. Comput. Graph,* pp. 2917-2926, 2012.
7. Sukhobok, D., Nikolov, N., Pultier, A., Ye, X., Berre, A., Moynihan, R., Roberts, B., Elvesæter, B., Mahasivam, N. and Roman, D., 2016, May. Tabular data cleaning and linked data generation with Grafterizer. *ESWC (Satellite Events) 2016: 134-139.*
8. Roman, D., Nikolov, N., Putlier, A., Sukhobok, D., Elvesæter, B., Berre, A. J., Ye, X., Dimitrov, M., Simov, A., Zarev, M., Moynihan, R., Roberts, B., Berlocher, I., Kim, S., Lee, T., Smith, A., & Heath, T. DataGraft: One-Stop-Shop for Open Data Management. To appear in the *Semantic Web Journal (SWJ) – Interoperability, Usability, Applicability* (published and printed by IOS Press, ISSN: 1570-0844), 2017, DOI: 10.3233/SW-170263.
9. Roman, D., Dimitrov, M., Nikolov, N., Putlier, A., Sukhobok, D., Elvesæter, B., Berre, A. J., Ye, X., Simov, A. & Petkov, Y. DataGraft: Simplifying Open Data Publishing. *ESWC (Satellite Events) 2016: 101-106*.
10. Roman, D., Dimitrov, M., Nikolov, N., Putlier, A., Elvesæter, B., Simov, A., Petkov, Y. DataGraft: A Platform for Open Data Publishing. In the *Joint Proceedings of the 4th International Workshop on Linked Media and the 3rd Developers Hackshop. (LIME/SemDev@ESWC 2016)*.
11. J. Wang, D. Crawl, I. Altintas, K. Tzoumas and V. Markl, "Comparison of Distributed Data-Parallelization Patterns for Big Data Analysis: A Bioinformatics Case Study," in *Proceedings of the Fourth International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2013.
12. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu and G. Fox, "Twister: A Runtime for Iterative MapReduce.," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.
13. M. Bala, O. Boussaid and Z. Alimazighi, "Big-ETL: Extracting-Transforming-Loading Approach for Big Data.," in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Neveda, USA., 2015.
14. A. Krukowski, Y. Kompatsiaris,, S. Papadopoulos and et al., "Big and Open Data Position Paper," 2013.
15. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Lax and S. Whittle, "The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.," in *Proceedings of the 41st International Conference on Very Large Data Bases (pp. 1792-1803)*, Kohala Coast, Hawaii: VLDB Endowment., 2015.
16. M. Sims, J. F. Kurose and V. R. Lesser, "Streaming versus Batch Processing of Sensor Data in a Hazardous Weather Detection System.," in *Proceedings of Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2005)*, 2005.
17. S. Shahrivari, "Beyond batch processing: towards real-time and streaming big data.," *Computers,* vol. 3, no. 4, pp. 117-129, 2014.
18. T. Furche, G. Gottlob, B. Neumayr and E. Sallinger, "Data wrangling for big data: Towards a lingua franca for data wrangling.," 2016.
19. 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 (pp. 1383-1394). ACM.*, 2015.
20. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-

memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.*, 2012.

21. D. Sukhobok, N. Nikolov and D. Roman, "Tabular Data Anomaly Patterns," in *3rd International Conference on Big Data Innovations and Applications. Innovate-Data 2017*, 2017 accepted. In press.
22. S. Riazi, "SparkGalaxy: Workflow-based Big Data Processing.," 2016.
23. H. Wang, M. Li, Y. Bu, J. Li, H. Gao and J. Zhang, "Cleanix: A parallel big data cleaning system," *ACM SIGMOD Record,* vol. 44, no. 4, pp. 35-40, 2016.
24. M. Kaur and G. Dhaliwal, "Performance Comparison of Map Reduce and Apache Spark on," *International Journal of Computer Sciences and Engineering,* vol. 3, no. 11, 2015.