DfAnalyzer: Runtime Dataflow Analysis of Scientific Applications using Provenance

Vítor Silva
COPPE / UFRJ
Brazil
silva@cos.ufrj.br

Daniel de Oliveira
IC / UFF
Brazil
danielcmo@ic.uff.br

Patrick Valduriez
Inria and LIRMM
France
patrick.valduriez@inria.fr

Marta Mattoso
COPPE / UFRJ
Brazil
marta@cos.ufrj.br

ABSTRACT

We present DfAnalyzer, a tool that enables monitoring, debugging, steering, and analysis of dataflows while being generated by scientific applications. It works by capturing strategic domain data, registering provenance and execution data to enable queries at runtime. DfAnalyzer provides lightweight dataflow monitoring components to be invoked by high performance applications. It can be plugged in scientific code scripts, or Spark applications, in the same way users already plug visualization library components. During this demo, we will show how DfAnalyzer captures the dataflow, provenance, as well as how it provides runtime data analyses of applications. We will also encourage attendees to use DfAnalyzer for their own applications.

PVLDB Reference Format:
Vítor Silva, Daniel de Oliveira, Patrick Valduriez, Marta Mattoso. A Sample Proceedings of the VLDB Endowment Paper in LaTeX Format. *PVLDB*, 11(12): 2082-2085, 2018.
DOI: https://doi.org/10.14778/3229863.3236265

1. INTRODUCTION

Scientific applications typically involve the execution of complex computational models and, consequently, the generation of a huge volume of heterogeneous data. These data are commonly stored in several workspaces as raw data files, which often follow a *de facto* standard format established by the application domain, *e.g.*, HDF5 and FITS. However, despite the big data volume, spread in thousands of files, typically only a small subset of the data is relevant and used for analysis [6].

These complex scientific applications are long lasting even when executing in parallel with High Performance Computing (HPC). They often require fine-tuning of the parameters or changing functions due to their exploratory nature [8]. Supporting data monitoring and analysis at runtime allows for anticipating the evolution of results, avoiding waiting for the whole execution to finish or aborting the execution to adjust it and resubmit. Visualization tools, like ParaView and VisIt, are present in most scientific applications, particularly in HPC to help on data analysis and monitoring [2]. Computational scientists (our target users) already include visualization library calls in their simulation script codes to share data with visualization tools to generate images and videos to be analyzed at runtime. Despite being mandatory, these tools have very limited query support and no provenance data, limiting the scope of runtime data analysis support.

There are several open issues in data analysis in long lasting parallel executions, like supporting the identification of data regions of interest and the dataflow implicit in the contents of raw data files. We present DfAnalyzer, a tool that supports runtime dataflow analysis for HPC applications. DfAnalyzer relates raw data files, exposes strategic domain data associated to these files, and generates dataflow provenance with debugging data all in the same columnar database, which is managed by MonetDB. This database acts as a global view of raw data and metadata, which can be queried during a long application execution complementing visualization tools. DfAnalyzer has several monitoring and raw data extraction components that are invoked in the same way users already do for the visualization tools.

DfAnalyzer incurs in negligible overhead (less than 0.5% of the application elapsed time) as measured while supporting some high performance applications [3,8]. DfAnalyzer components are efficient because they extract domain data as they are being generated, often directly from the same memory space avoiding opening and accessing raw data files. The resulting database is also very small as compared to the raw data itself. At the same time, it allows for complex query submissions.

These extracted strategic domain data (*e.g.*, quantities of interest) are often scalar data associated to large graphs or raw datasets as a result of a complex computation, like the volume of a region, the residual norms or a probability measure. In DfAnalyzer, all raw data is kept in files with their original format, but the strategic data representing these files are extracted, along with a reference to its source file, and the dataflow. As a result, DfAnalyzer provides a rich set of data to help tracking the evolution or the top values of the main results of the application, and directly pointing to the associated raw data file. Otherwise, it would require an *ad-hoc* program using a third-party tool to parse all raw data files to find and to extract the relevant contents from them.

In situ raw data query engines, like Slalom [6], provide for efficient scientific data analysis with adaptive mechanisms that access raw data directly from the files avoiding data copies, loading, and indexing overheads. However, these raw data query engines require that the files are all generated before starting the query submission for the analysis. In addition, this offline approach might not identify the implicit dataflow of the data transformation. It requires a complex analysis to obtain the relationship from within contents of heterogeneous files that compose the dataflow. Also, some relevant data might be available only during the execution and will no longer be available offline. DfAnalyzer and *in situ* raw data query engines are complementary. DfAnalyzer can be seen as a first step data analysis for its runtime support. Its generated global

Proceedings of the VLDB Endowment, Vol. 11, No. 12

DOI: https://doi.org/10.14778/3229863.3236265

view database can be used by *in situ* raw data query engines as an index to execute queries on the strategic data and directly identify regions of interest to be further analyzed offline, without having to parse all the files.

Despite being targeted for long running scientific applications [3,8], in this demo, we will use a simple dataflow use case with parallelism managed by Apache Spark [1]. We will walk it through the process of dataflow modeling, data capture, provenance management, and analysis supported by DfAnalyzer. We will show (a) how users point to strategic data to be extracted by DfAnalyzer, (b) raw data extraction and indexing, (c) the graphical interface of DfAnalyzer to visualize dataflows, and (d) runtime raw data and dataflow analysis using the query interface.

2. BACKGROUND

During the execution of scientific applications, users need to analyze data consumed and produced by different programs or data transformations. In a previous work [5], our workflow system was in charge of collecting data and registering provenance, available for queries at runtime. Provenance capturing registers the flow of data transformations with data input/outputs, but the dataflow remains implicit. DfAnalyzer is an alternative that, similarly to noWorkflow [7] avoids having the workflow system being in charge of the execution control flow and having to wrap data transformations, which can be a problem when using HPC libraries. However, noWorkflow is specific for Python scripts and has no support for raw data or HPC. DfAnalyzer is based on a dataflow representation to register the flow of datasets and data elements. In this section, we present a simple use case for predicting sales forecasts, dataflow concepts and how our dataflow-aware approach is able to analyze data elements manipulated by transformations.

2.1 SalesForecasts: a data science application

This demonstration paper concentrates on a simple application for predicting the sales of a clothing company based on the customers consumption patterns adapted from [4], renamed as *SalesForecasts*. Company's leaders, or the Decision Supporting System (DSS) specialists, aim to finalize their requests with providers to maximize profits. More specifically, these DSS specialists design systems to analyze the quantities of items to be sold, while managing the inventory. In this scenario, they often start the design by modeling the sales prediction on top-selling items, meaning using a reduced input dataset. Once the predictions meet the expected budget and inventory capacity, they can evaluate new items by adjusting the input dataset.

Figure 1 shows the *SalesForecasts* data transformations as tagged black edges. It starts with the transformation *deduplicate_customers* that reads customer records from different lists (input dataset *customers*) and removes duplicated records (output dataset *deduplicated_customers*). The next data transformations, *filter_us_customers* and *filter_ue_customers*, filter deduplicated customers into two customers datasets. Then, *transformation merge* join all customers from these countries in the output dataset *combined_customers*.

Since there are deduplicated customers from specific countries, the transformation *cross_product* combines these customers with clothing items from a list with all clothing items to be analyzed by the predictive model (input dataset *clothing_items*), generating the dataset *combined_customers_with_items*. After that, its results are considered with the customer buying patterns (input dataset *buying_patterns*) to obtain the predictive model with the data transformation *predict*, which produces the probabilities of selling clothing items (dataset *probabibilites_of_selling_items*). Finally, these probabilities are grouped by the clothing item identifier to calculate the expected number of items to be sold on the next season (output dataset *sales_forecasts* generated by the data transformation *aggregate*).

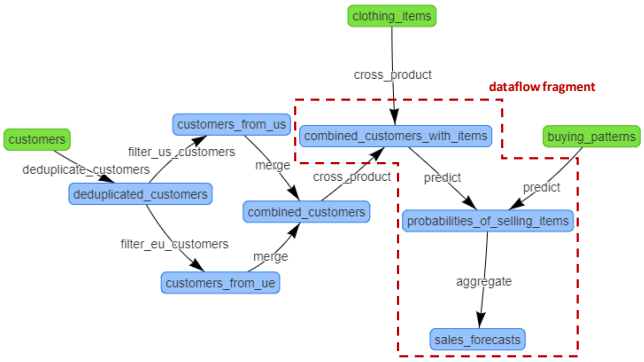

Figure 1. Data perspective view of SalesForecasts application.

Several analyses depend on the entire dataflow. Considering the dataflow shown in Figure 1, DSS specialists need to design systems that present probabilities of selling a specific clothing item with its sales forecast, and its description. Since such data is stored in different datasets produced by different data transformations, reconstructing this implicit dataflow from raw data files can be error prone. DfAnalyzer represents dataflows as they occur.

2.2 Dataflow Concepts

The smallest unit of interest is the *data element* (e). A data element has values (v) for each predefined attribute (a) that represents e. The schema that represents e is a set A, where each a is represented as (name, type). A set of data elements consists of a *data collection* (c). Then, a *dataset* (s) is composed of a set of data collections (C). A *data transformation* (t) consumes data from one (or more) dataset(s) as input (S_{input}) and produces data in one (or more) dataset(s) as output (S_{output}). Furthermore, two data transformations can present a *data dependency* (φ) with relation to a dataset, when the data is produced by one data transformation ($t_{previous}$) and consumed by another (t_{next}). Based on such concepts, a *dataflow* (D_F) is defined by the data resulting from the composition of data transformations (T), manipulating datasets (S) concerning data dependencies (Φ).

Table 1. Dataflow definitions.

Concept	Definition
data element	$e = (v_1, v_2, \dots, v_\alpha) \therefore$ v_i *is an attribute value for each* a_i *in* A $\wedge\ a_i = (name, type)$
data collection	$c = \{e_1, e_2, \dots, e_\beta\}$
dataset	$s \equiv C$
data transformation	$t = (S_{input}, S_{output})$
data dependency	$\varphi = (s, t_{previous}, t_{next})$
dataflow	$D_F = (T, S, \Phi)$

Figure 1 shows a dataflow view of SalesForecasts, where nodes in the graph represent datasets and edges represent data transformations. Input datasets are colored in green and output/input datasets in blue. Figure 2 shows some dataset views of SalesForecasts with the data elements that belong to each collection

of the dataset. With this representation at the data element level, DSS specialists are able to relate the sales forecasts (attribute *quantity* of dataset s_{10}) with specific clothing items (attribute *description* of dataset s_8), by correlating data elements (attribute *item_id*) among datasets.

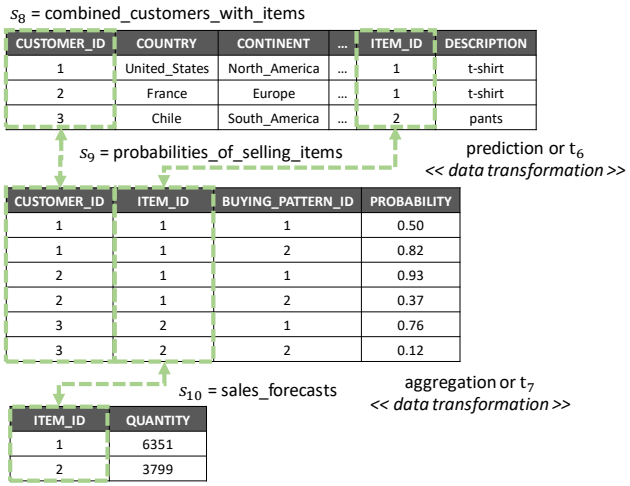

Figure 2. Excerpt of a data element flow in SalesForecasts.

3. OVERVIEW OF DFANALYZER

DfAnalyzer follows ARMFUL [8], a component-based reference architecture for dataflow analysis. DfAnalyzer has six components, shown, in Figure 3, as gray rounded rectangles: (i) Provenance Data Extractor (PDE); (ii) Raw Data Extractor (RDE); (iii) Raw Data Indexer (RDI); (iv) Dataflow Viewer (DfViewer); (iv) Query Interface (QI); and (vi) Database (DfDB). DfAnalyzer captures provenance and domain-specific data (*i.e.*, strategic data obtained during the application execution). DfAnalyzer enables raw data extraction from such files and content indexing by direct access to memory or invoking third-party programs or tools. The first three components are invoked by plugging calls on the application, while the other two have independent interfaces for the user to submit data analyses at runtime.

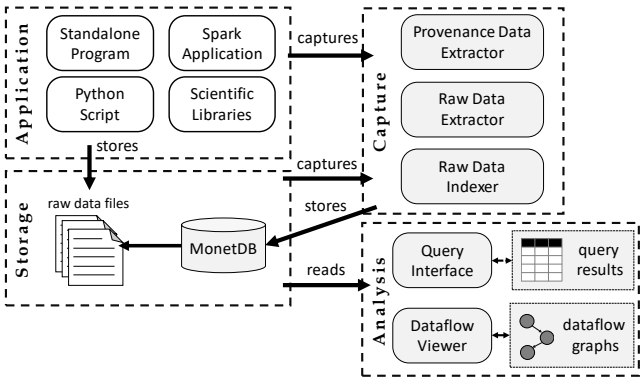

Figure 3. Architecture of DfAnalyzer.

We plugged DfAnalyzer to the SalesForecasts application parallelized with Spark in a 1000-core computer and we observed an execution time overhead of up to 0.34% of the application elapsed time (1h:48min), which can be considered negligible. Following we detail each component of DfAnalyzer.

3.1 Provenance and Raw Data Extraction

The PDE component provides a RESTful API, in which the body of HTTP requests represents the mapping between the data processing steps of an application and the dataflow concepts presented in Section 2. Figure 4 shows the modifications in the data transformation *predict* of SalesForecasts using PDE for extracting provenance data. Therefore, users are able to define which computational methods in their applications correspond to a data transformation to be registered. They can also define which data elements consumed and produced by each data transformation are relevant to be registered. If raw data is stored in files, it requires the invocation of the component RDE and, if indexing is desired, RDI. Figure 4 shows a method, named *rawDataAccess*, developed for SalesForecasts application using RDE and RDI to extract and index raw data from Spark Resilient Distributed Datasets (RDD) stored in files. Since extraction is at runtime, often data is still cached.

```
public JavaRDD<String> predict(String transformationTag,
                JavaRDD<String> combined_customers_with_items) {
    String inputDataset = Utils.getInputDataset(transformationTag);
    String outputDataset = Utils.getOutputDataset(transformationTag);
    // DfAnalyzer - PDE
    pde.task(dataflowTag, transformationTag, config.getTaskID(),
            "RUNNING", outputWorkspace, config.getResource());
    pde.collection(inputDataset,
            "{{" + getElements("buying_patterns") + "}}");
    pde.dependency("{cross_product}", "{" + config.getTaskID() + "}");
    pde.sendRequest();
    // Spark
    JavaRDD<Tuple2<String, String>> cartesianProduct =
            combined_customers_with_items.cartesian(buyingPatterns).
            rdd().toJavaRDD();
    …
    JavaRDD<String> probabilities_of_selling_items =
                                    similaritiesPair.map(new Prediction());
    // Write RDD
    prediction.saveAsTextFile(outputWorkspace + transformationTag);
    // DfAnalyzer – RDE and RDI
    rawDataAccess(transformationTag);
    // DfAnalyzer – PDE
    pde.changeTaskStatus("FINISHED");
    pde.collection(outputDataset,
                "{{" + getElement(transformationTag) + "}}");
    pde.sendRequest();
    return probabilities_of_selling_items;
}
```

Figure 4. Transformation *predict* tracked by DfAnalyzer.

3.2 Data Loading and Dataflow Analysis

As provenance and raw data have been extracted/indexed, PDE loads such data into the DfDB database. DfDB follows the schema PROV-Df [8], compliant to W3C PROV. All data expected in PROV traces are available for queries. In distributed and parallel computing environments, we deploy MonetDB and DfDB in a dedicated computational node for loading data asynchronously without jeopardizing the performance of parallel applications.

DfAnalyzer provides DfViewer and QI for online dataflow analysis. DfViewer is a Web application that accesses all dataflows from the DfDB database and generates a view of the dataflows selected by the user. QI is a RESTful service that aids users to run their SQL queries. The user provides a dataflow fragment of interest, attributes to be returned (as the SELECT SQL clause), and conditions on attribute values (as the WHERE SQL clause) so that QI automatically generates the SQL to submit the queries and show the results. In Section 4, we present a query example using QI.

4. DEMONSTRATION PLAN

In our demonstration session, we encourage conference attendees to experience runtime dataflow analysis using SalesForecasts application in DfAnalyzer, for example by defining raw data extraction and querying using DfViewer and QI. This demonstration application using DfAnalyzer is available at https://github.com/vssousa/dfanalyzer-spark.

Use Case. John is the IT leader of a clothing company, *i.e.*, a DSS specialist, that designs an application to predict the sales to the next season based on the consumption patterns of the customers. He uses the SalesForecasts application (Figure 1). He initially uses the original version of this application, without DfAnalyzer, *i.e.*, only the source code using Spark operators, as shown in Figure 4.

The SalesForecasts output data is the *sales_forecasts.rdd* file, with the clothing item identifier and the quantity of sales for such item. However, to present a predictive data analysis, John needs to provide the description of the clothing item (to understand which is the category, the size, and other informations) and the probability of selling a specific clothing item to a customer according to a specific buying pattern. He uses this selling probability to only select clothing items that have a high probability (*i.e.*, *probability* > 0.65) of being sold in accordance with a specific customer buying pattern. This value probability can be set/changed at runtime.

Without DfAnalyzer, he has to write programs to access and extract raw data from the *clothing_items.csv* (attributes *item_id* and *description*), and the intermediate file stored in the RDD format, *i.e.*, *combined_customers_with_items.rdd* (attributes *item_id* and *probability*), besides the sales data from the output file *sales_forecasts.rdd* (*item_id* and *quantity*). Even with the raw data extracted and, maybe, indexed, he has to correlate these raw data elements from different files and formats (*i.e.*, he has to develop a query for a dataflow analysis based on the extracted raw data).

John would also have to wait until the end of the application execution for running his own developed data analysis programs. It is time-consuming and error-prone. Now let us see how to plug DfAnalyzer to the SalesForecasts application.

Dataflow Specification. As the first step to use DfAnalyzer, database specialists scheduled a meeting with John to know which are strategic data and metadata for his predictive analyses. Then, database specialists model SalesForecasts dataflow according to provenance data of the main data transformations and strategic data to the DSS specialist. As a result, a dataflow specification is obtained with transformations, datasets, data dependencies, and attributes to be monitored by DfAnalyzer.

Raw and Provenance Data Extraction. Since database specialists have a dataflow specification to guide the flow of data elements generation, they help John to plug DfAnalyzer components, *i.e.*, RDE, PDE, RDI, for extracting raw and provenance data at runtime, as well as indexing. Figure 4 shows the method *rawDataAccess()* introduced in the original source code of the application for raw data extraction and indexing, which is based on the invocations of RDE and RDI as shown in Figure 5. Since DfAnalyzer monitoring components are plugged in the predictive data science application, John can submit its execution.

Dataflow Analysis. John may run DfViewer for visualizing the dataflow specification in a dataset perspective view (Figure 1) and checking if the registered dataflow is in accordance with the data processing steps of the application. Then, he uses this visualization for defining the source target, the destination target, the attributes to be returned by the query, and the conditions for selecting specific raw data elements from the dataflow fragment of interest. Figure 6 shows the input arguments specified to QI for generating and running the SQL-based query.

```
//command line to run RDE with cartridge Program
./RDE PROGRAM:EXTRACT probabilities_of_selling_items
    /root/sales_forecasts probabilities_of_selling_items.rdd
    {customer_id:numeric, item_id:numeric, …, probability:numeric }
//command line to run RDI with cartridge FastBit
./RDI FASTBIT:INDEX probabilities_of_selling_items
    /root/sales_forecasts probabilities_of_selling_items
    {customer_id:numeric, item_id:numeric, …, probability:numeric }
```

Figure 5. Raw data extraction and indexing using DfAnalyzer.

```
source(clothing_items)
target(sales_forecasts)
projection(clothing_items.description; sales_forecasts.quantity)
selection(probabilities_of_selling_items.probability > 0.65)
```

Figure 6. Predictive data analysis using the QI component.

5. ACKNOWLEDGEMENTS

We thank Thiago Perrotta, Thaylon Guedes, and Luciano Leite for their help in development. We would like to thank INRIA, CAPES, CNPq, FAPERJ, HPC4E (EU H2020 and MCTI/RNP-Brazil). This work has been performed (for P. Valduriez) in the context of the Computational Biology Institute. The HPC Center at COPPE/ Federal University of Rio de Janeiro has provided computing and storage resources on the Lobo Carneiro supercomputer.

6. REFERENCES

[1] Armbrust, M., Zaharia, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., et al. Scaling spark in the real world: performance and usability. *PVLDB*, 8(12):1840–1843, 2015.

[2] Ayachit, U., Bauer, A., Duque, E.P.N., Eisenhauer, G., Ferrier, N., Gu, J., Jansen, K.E., Loring, B., Lukić, Z., et al. Performance Analysis, Design Considerations, and Applications of Extreme-scale in Situ Infrastructures. *Supercomputing conference*, 79:1–12, 2016.

[3] Camata, J.J., Silva, V., Valduriez, P., Mattoso, M., Coutinho, A.L.G.A. In situ visualization and data analysis for turbidity currents simulation. *Computers & Geosciences*, 110:23–31, 2018.

[4] Ikeda, R., Widom, J. Panda: A System for Provenance and Data. *IEEE Data Engineering Bulletin*, 42–49, 2010.

[5] Ogasawara, E., Dias, J., Oliveira, D., Porto, F., Valduriez, P., Mattoso, M. An Algebraic Approach for Data-Centric Scientific Workflows. *PVLDB*, 4(12):1328–1339, 2011.

[6] Olma, M., Karpathiotakis, M., Alagiannis, I., Athanassoulis, M., Ailamaki, A. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *PVLDB*, 10(10):1106–1117, 2017.

[7] Pimentel, J.F., Murta, L., Braganholo, V., Freire, J. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB*, 10(12):1841–1844, 2017.

[8] Silva, V., Camata, J., de Oliveira, D., Coutinho, A.L.G.A., Valduriez, P., Mattoso, M. In Situ Data Steering on Sedimentation Simulation with Provenance Data. *Poster session of Supercomputing conference*, 2016.

[9] Silva, V., Leite, J., Camata, J., Oliveira, D., Coutinho, A.L.G.., Valduriez, P., Mattoso, M. Raw Data Queries during Data-intensive Parallel Workflow Execution. *Future Generation Computer Systems Journal*, 75:402–422, 2017.