To UDFs and Beyond: Demonstration of a Fully Decomposed Data Processor for General Data Wrangling Tasks

Nico Schäfer
RPTU Kaiserslautern-Landau
nico.schaefer@cs.rptu.de

Damjan Gjurovski
RPTU Kaiserslautern-Landau
damjan.gjurovski@cs.rptu.de

Angjela Davitkova
RPTU Kaiserslautern-Landau
angjela.davitkova@cs.rptu.de

Sebastian Michel
RPTU Kaiserslautern-Landau
sebastian.michel@cs.rptu.de

ABSTRACT

While existing data management solutions try to keep up with novel data formats and features, a myriad of valuable functionality is often only accessible via programming language libraries. Particularly for machine learning tasks, there is a wealth of pre-trained models and easy-to-use libraries that allow a wide audience to harness state-of-the-art machine learning. We propose the demonstration of a highly modularized data processor for semi-structured data that can be extended by means of plain Python scripts. Next to commonly supported user-defined functions, the deep decomposition allows augmenting the core engine with additional index structures, customized import and export routines, and custom aggregation functions. For several use cases, we detail how user-defined modules can be quickly realized and invite the audience to write and apply custom code, to tailor provided code snippets that we bring along to own preferences to solve data analytics tasks involving sentiment analysis of Twitter tweets.

PVLDB Reference Format:
Nico Schäfer, Damjan Gjurovski, Angjela Davitkova, and Sebastian Michel. To UDFs and Beyond: Demonstration of a Fully Decomposed Data Processor for General Data Wrangling Tasks. PVLDB, 16(12): 4018 - 4021, 2023.
doi:10.14778/3611540.3611610

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/JODA-Explore/JODA.

1 INTRODUCTION

Recent years have witnessed an unprecedented competition of data management researchers and companies to provide solutions to handle vast amounts of increasingly heterogeneous data. This has spurred the development of novel data management solutions (aka. NoSQL), tailored to specific data characteristics and query languages, followed by the augmentation of traditional relational database management systems to support novel features. However, many functionalities are provided solely in programming language libraries, most prominently (complex) pre-trained models and toolkits around machine learning tasks (e.g., [16]). Such features are difficult to integrate with existing systems without considerable effort. In the proposed demonstration, we show that by enabling user-defined modules, seemingly simple data processors can combine the advantages of a multitude of systems to enable new data processing pipelines that outperform existing solutions in efficiency and usability. Our findings are based on a prototypical modularized data processor adapted from our previous work on JODA [10], where core parts of JODA have been rewritten to accept Python scripts as implementations. While this adds versatility, the high-performance, multi-thread core of JODA, written in C++17, remains untouched. This combination offers blazing fast parsing and processing of JSON data in a scalable fashion [11], combined with the possibility to write extensions like user-defined functions, custom indices, and support of additional data formats and I/O routines, provided in plain Python. Note that such extensions should be employed only if the core functionality of JODA does not support the required functionality as unnecessary calls of single-threaded Python code are expected to be slower than native JODA routines.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611610

```
LOAD FROM FILE "twitter.json"
CHOOSE EXISTS ('/text')
AS ('/sentiment':SENTIMENT('/text')), ('/text':'/text')
STORE AS SQL_EXPORT "postgresql://... twitter_sentiment"
```

Listing 1: Sample JODA query with external function

2 CORE ARCHITECTURE AND MODULES

JODA [10] is an open-source, lightweight, multi-threaded JSON document processor, designed to ease data pre-processing. It has demonstrated superior performance for iterative query workloads compared to MongoDB, PostgreSQL, Spark, and JQ [11].

We fully decomposed JODA into a set of independent modules or `tasks`. Each task is defined as a class that can be executed, given an input queue, an output queue, or both. Depending on the query, tasks are added into a query execution `pipeline` where compatible tasks are connected via I/O queues. The `scheduler` then executes instances of each `task` using all available or a configured number of CPU cores. A `task` may limit the number of instances that can be executed in parallel. For example, a task that reads data from a file system may limit the number of instances to one. A task may stop its execution when the input queue and the task are finished, the output queue is full (stalled), or the input queue is temporarily empty (starved). Stalled and starved tasks are restarted at a later time by the scheduler. Moving from a static query planner calling hard-coded functions to a dynamic query planner based on tasks greatly improved flexibility. Based on user queries and optimization rules, tasks can now be added, replaced, reordered, or removed.

While the data being transmitted between tasks can be of different formats, JODA mainly uses `containers` as atomic units of data. A container bundles multiple JSON documents and supplementary data structures like (cracking) indices, query caches, and data synopses. This enables an (almost) entirely lock-free execution of queries as the documents are passed through the pipeline without requiring supporting centralized data. Listing 1 demonstrates a JODA query using JODA native functions, user-defined function for *sentiment* analysis and a customized *PostgreSQL export.*

Within the tasks that interact with specific parts of the documents, like filtering, transforming, and aggregating, we use `values` as interfaces to the data. A `value` may be a simple pointer to a part of the document or a function with parameters that uses inputs to calculate a new value. All values in JODA are also implemented as independent classes being loaded and dynamically added to the internal query processing. They define their output type, how many parameters of which type they expect, and provide a function that calculates the result, given a list of parameters. For instance, a value calculating the length of the string defines its return type as `int` and expects one parameter of type `string`. With JODA being decomposed into independent modules that are scheduled and connected based on rules, we can easily extend the system. For instance, a filter task can now be added—and automatically connected—to every other task that returns a `container`.

This enables the user to add new tasks to the system without understanding or changing the core code. Unlike previous work, through the modularized architecture, JODA allows the substitution of every component of the system while maintaining the unmatched fast parsing and processing of JSON documents.

2.1 User-Defined Modules

`Modules` enable users to extend the system. A `module` is a single script file that can be loaded into JODA to provide some sort of functionality. It has to provide specific functions and variables depending on the feature it implements. We support the following types of modules: (1) **Import** - Connects the processor to new data sources that may previously not be supported. (2) **Export** - Exports JSON documents into a user-supplied format or system. (3) **Index** - Implements a new index for improving the filter performance of JODA. (4) **Agg** - Supplies a new aggregation function that uses a set of data to compute a single result. (5) **Value** - A typical user-defined-function as known from other database systems. It provides a single JSON value, given one or multiple parameters.

A `ModuleRegistry` is used that organizes a set of `modules`, depending on the task to be solved. When a user supplies a new module, the registry loads the given script and infers the type of the module by the given functions. The module can be used when querying until the user deregisters the module. The registry stores a mapping of module names to script location such that the query pipeline can find and invoke it. Internally, each module initializes a `task` or `value` class which is stored in the registry.

During query parsing, the registry is accessed to find the user-supplied module if an aggregation or a value function is referenced but not found in JODA's native function list. The value created from the module is embedded into the internal query representation. During query execution, the module is then invoked to compute the result. Similarly, if the query contains an unknown import or export statement, the registry is checked for an import or export task to be added to the query pipeline. Indices form a special case, as they are not directly referenced in queries but chosen by the query planner. All internal and user-supplied indices are collected during query planning, and the planner then chooses the best index using an estimator given by the index. Given a query, the internal index data, and a `container`, the index estimates how many documents have to be reevaluated with the actual predicate if it is executed.

2.2 Connecting Scripts and System

A major aspect of executing user-supplied code is to enable an efficient and versatile communication between scripts and the core system itself. In JODA, data is stored internally as a dynamic in-memory JSON data structure. As most languages have some kind of support for JSON data or at least have third-party libraries that provide such support, it would be possible to translate the internal structure back into a JSON string-representation and pass it to the user-supplied scripts. Then a language-specific function can parse and interpret the JSON data. However, translating and parsing such a document would cause significant overhead, especially if the query deals with large amounts of documents. Hence, it is preferable to immediately translate the internal data structure into a language-specific one. Before a user function is executed, the system will initialize a variable in the chosen language environment and call the translation function of the implemented language.

As we use Python to write modules, we decided to map JSON values into their Python-native counterpart by traversing the JSON document depth-first. Basic data types like `integer`, `float`, `null`, `Boolean`, and `string` are directly initialized in the Python environment. The composite `array` can also be directly converted to JSON, as Python supports arrays with heterogeneous data types. Every JSON `object` is converted into a Python `dict`. The functions in the user-supplied script are then called with the translated variables as parameters. The potential return values are translated back to the JSON format using the inverse mapping of the previous step. Most language APIs return a handle to the internal result, which is translated by the system using the language-specific `translate` function. As next, the language-specific result is uninitialized, and the translated value is passed on to the internal query engine.

3 SAMPLE USE CASES

3.1 User-Defined Functions

JODA can be extended with user-defined functions that can be used during querying. To supply the extension, a script following a predefined template needs to be provided. We distinguish between two different types of extensions, per record and for a collection of records. For the per-record extension, the user needs to implement the method `get_value`. For a feature that operates on a collection of records, the user has to implement *four* functions. Initializations are done in `init_state`. The function `aggregate` is executed over the individual JSON partition assigned to different cores. The aggregation happens per-tuple with an aggregator storing the intermediate result. Merging of the per partition results is done in `merge`, and the final result is computed in `finalize`. To extend JODA, we implemented **language detection**, **sentiment analysis**, **training and evaluation of a learned model**, and **computation of number statistics** as extensions in Python.

Certain functions are common in specific languages and thus incorporated in advanced libraries. By using them, one can avoid recreating the same functionality in the native language of the used system. Examples include machine learning libraries, i.e., language detectors or sentiment analyzers. Enabling the loading of such a model in JODA will produce the needed results with minimal effort.

```
model = fasttext.load_model(PRETRAINED_MODEL_PATH)
def get_value(arg1):
    return model.predict(arg1).split('__label__')[1]
```

Listing 2: Language identification in Python

A script for extending JODA with language classification utilizing the *fastText* [8] language classifier is shown in Listing 2. The registered function (`LANG`), can be used directly in the query language:

```
LOAD Twitter AS ('/t':'/text'),('/lang':{LANG('/t'));
```

As one example of an extension that operates on a collection of records, we implemented the computation of simple statistics such as average over a given attribute in Python (Listing 3).

```
def aggregate(state, num):
  return [state[0]+num, state[1]+1]
def merge(state, other):
  return [state[0]+other[0], state[1]+other[1]]
def finalize(state):
  return (state[0] / state[1], state[0], state[1])
def init_state():
  return [0, 0]
```

Listing 3: Statistics computation in Python

Adding new capabilities to JODA can also simplify the handling and improve the performance of existing code. For example, for training a machine-learning model where the data is too large to be stored in memory, the user will first need to load the data by taking only the relevant parts for the approach and perform the required pre-processing. However, JODA already provides an efficient mechanism to load and process data in batches. We consider training a **Stochastic Gradient Descent (SGD) Classifier**, following the extension template working over a collection of records. In the `init_state` method, the model is initialized. If a model exists, it is loaded for the current JSON partition. The input records are added to the training batch in the `aggregate` method. If the batch reaches the predefined size, the training of the model is invoked. The `merge` method is necessary for considering remaining documents from the covered JSON partition. These documents will be added to the current data batch. In the `finalize` method, once the remaining documents are used for training, the model will be saved. To use the trained model, we realize the template for extension over single records. Hence, in the `get_value` method, we use the model for predicting over the input records. Native JODA features are used to filter, clean, and scale the dataset before training the model.

3.2 Replacement and Augmentation

JODA also allows replacement of data structures and query processing behavior. Consider the case of replacing existing membership indices with more efficient ones. Often this warrants modification of the system code, specifically where such structures are tightly bound with the system. In JODA, one has to provide a replacement script where the implementation logic will be realized through five methods: The `init_index` method initializes the index with a persistent state. The method `estimate_usage` estimates the remaining work after a predicate from the query has been evaluated on the index. For each container the `execute_state` function is called, which may return a filtered document set, using only the state of the index. If this is not possible, and the actual contents are needed, `None` is returned, and the system calls the `execute_docs` method to determine which of the given documents fulfill the predicate. The method `improve_index` updates the index with the final query results if they are suitable for the index. As a proof-of-concept, we realized a *Bloom filter* and a *query cache*. The module for replacing an existing index with a query cache is depicted in Listing 4.

```
def estimate_usage(predicate, state):
  return 0 if predicate in state else None
def execute_state(predicate, state):
  return (state[predicate], True)
def execute_docs(predicate, docs, state):
  return None
def improve_index(predicate, state, doc_index):
  state[predicate] = doc_index
def init_index():
  return dict()
```

Listing 4: Query cache in Python

This feature can be used to implement domain-specific indices that are too specific to be implemented in a general-purpose data processor. For example, if the user works with geospatial data, one can implement a spatial index that is optimized for the data set.

3.3 Customized Data Import and Export

Consider the case where data is distributed over multiple files or systems, e.g., a company employs a relational database of customers and sales data, a key-value store for online shopping carts, and local CSV files of access logs. Joint data processing over such data sources is a tedious task and requires human-involved pre-processing, before the system of choice is able to execute the actual query. A solution to this problem can be wishful thinking—to hope that the developers add support for the required data. A more realistic solution is to have the system provide simple means to extend the import and export mechanisms. For many use cases, it is enough to let the user specify how the data has to be transformed, read, and written.

```
def init(param):
  return <State>
def set_next(state, data):
  export(transform(data))
def finalize(state):
  pass
```

Listing 5: Data export module interface

Listing 5 shows our suggestion for a simple interface supporting user-given export tasks. Like import modules, export modules work with an internal state that can optionally represent a (stateful) resource to be used, like a file handle or an open database connection. As a proof-of-concept, we implemented a CSV file and a PostgreSQL table reader and writer.

4 DEMONSTRATION DESCRIPTION

The goal of the system demonstration is to motivate the need for extensible data processors and to show that custom extensions can be plugged into JODA quite easily. The demonstration is centered around performing data analysis tasks over Twitter tweets using an external library for **sentiment analysis**. To ease the process of adding user-defined modules, we provide a user-interface through which the modules can be easily uploaded as scripts (Figure 1 part 1) and used as functions when querying (Figure 1 part 2).

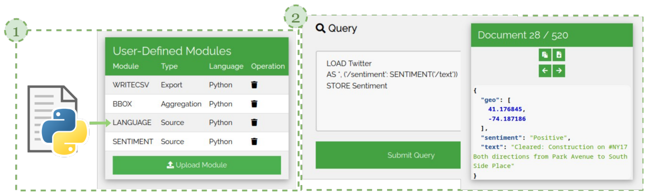
Adding Functions: First, using the visual interface of JODA, through a Python script we will automatically add the current date and time to each tweet to keep track of the time of import. The Python module supplying the custom datetime value will be briefly explained, and then imported into the JODA data processor. The added module will be utilized by a sample query through which the value will be added to every document. We will additionally present a more complex module, which imports a third-party sentiment-analysis library. The module implementation will be showcased, and then used in JODA to add a sentiment to each tweet. We will analyze the performance of this sentiment analysis, by comparing it to a native Python implementation.

Adding Aggregates: Using the same process, we will illustrate how to execute custom aggregation functions in JODA. Employing the previously introduced sentiment-analysis library we will write an aggregation function, which calculates the average sentiment of a group of texts. The user will be able to evaluate the usefulness of the module by using it in a query, such as the one that calculates the average sentiment of all tweets per user, and inspect the results through the visual interface. In addition, we will display a common scenario when working with *geo* data. The module will calculate the minimum bounding rectangle of tweet coordinates.

Connecting Data Sources: We will further demonstrate how JODA can be extended with a data export module. This module will show the ease in which JODA can be extended to enable the user to write JSON data to a specified PostgreSQL table. In our presentation, we will filter out all users that have an aggregated positive sentiment and add them to a PostgreSQL database.

5 RELATED WORK

Gupta and Ramachandra [5] investigate the execution of procedural extensions in an RDBMS. Friedman et al. [4] present a framework to implement user-defined functions where they can be included as SQL sub-queries. Crotty et al. [2] describe an architecture that automatically compiles workflows of user-defined functions with complex operations. Hellerstein et al. [6] present an open-source library incorporating SQL-based algorithms for ML and data mining run within a database engine. Passing et al. [9] assess the inclusion of data analytics, and allow integration of analytical operators directly in SQL, using novel user-defined lambda expressions. Schüle et al. [12] use the PostgreSQL JIT compilation to allow user-written lambda functions and they demonstrate combining them with data mining algorithms. Others [3, 7] allow for interpretation of procedural programs as subqueries by an SQL engine, by transforming them to recursive CTEs. Sichert and Neumann [14] present user-defined operators for the inclusion of algorithms from an arbitrary programming language into an existing DBMS. Boehm et al. [1]



Module	Type	Language	Operation
WRITECSV	Export	Python	
BBOX	Aggregation	Python	
LANGUAGE	Source	Python	
SENTIMENT	Source	Python	

```
LOAD Twitter
AS *, ('/sentiment': SENTIMENT('/text'))
STORE Sentiment
```

```
{
  "geo": [
    41.176845,
    -74.187186
  ],
  "sentiment": "Positive",
  "text": "Cleared: Construction on #NY17
Both directions from Park Avenue to South
Side Place"
}
```

Figure 1: JODA's interface for adding external functionality

introduce an open-source system for end-to-end execution of ML models. Schüle et al. [13] use database systems for data inspection providing support for end-to-end pipelines including model training and testing. Tuplex [15] focuses on producing optimized end-to-end code for pipelines with Python UDFs. In an earlier JODA demonstration [10], we presented the core concepts of the approach, particularly the query language and runtime performance.

ACKNOWLEDGEMENTS

This work has been partially funded by the German Federal Ministry of Food and Agriculture under grant number 28DE113C18 (DigiVine). The responsibility for the content of this publication lies with the authors.

REFERENCES

[1] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. N. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqui, and S. B. Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. www.cidrdb.org.

[2] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.* 8, 12 (2015), 1466–1477.

[3] C. Duta, D. Hirn, and T. Grust. 2020. Compiling PL/SQL Away. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[4] E. Friedman, P. M. Pawlowski, and J. Cieslewicz. 2009. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2 (2009), 1402–1413.

[5] S. Gupta and K. Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.

[6] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.

[7] D. Hirn and T. Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *SIGMOD*. ACM, 723–735.

[8] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607.01759* (2016).

[9] L. Passing, M. Then, N. C. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*. OpenProceedings.org, 84–95.

[10] N. Schäfer and S. Michel. 2020. JODA: A Vertically Scalable, Lightweight JSON Processor for Big Data Transformations. In *ICDE*. IEEE, 1726–1729.

[11] N. Schäfer and S. Michel. 2022. BETZE: Benchmarking Data Exploration Tools with (Almost) Zero Effort. In *ICDE*. IEEE, 2385–2398.

[12] M. E. Schüle, J. Huber, A. Kemper, and T. Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM, 6:1–6:12.

[13] M. E. Schüle, L. Scalerandi, A. Kemper, and T. Neumann. 2023. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *EDBT*. OpenProceedings.org, 40–52.

[14] M. Sichert and T. Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131.

[15] L. F. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD*. ACM, 1718–1731.

[16] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 38–45.