Conquering a Panda's weaker self - Fighting laziness with laziness

Demo Paper

Stefan Hagedorn
TU Ilmenau
Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Steffen Kläbe
TU Ilmenau
Ilmenau, Germany
steffen.klaebe@tu-ilmenau.de

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

ABSTRACT

The Python programming language has become very popular among data scientists because of its easy-to-learn syntax and rich ecosystem of libraries. Especially the Pandas framework is widely used for various data processing and analytics tasks. However, due to its memory management and eager evaluation Pandas does not scale and workstations quickly come to their limits even for moderate data set sizes. With Grizzly, we introduce a framework that produces SQL queries for operations on `DataFrames`, moving complexity from workstations to database servers. Grizzly allows to not only access data already stored in a database, but also to combine it with external data from files. Furthermore, users can use their own user-defined functions or use Grizzly's model join feature to easily apply machine learning models to data, both being executed inside the database server. This allows for fast and scalable data analytics operations, even with a small workstation.

1 INTRODUCTION

Data Science and Machine Learning are hot topics, not only in research but also in industrial and commercial applications. Although the terms *Data Analysis* and *Data Science* date back to the early 1960s and 1970s, respectively, with the rise of *Big Data* in the early 2000s more and more companies started to collect and analyze every piece of information they could generate about their, e.g., sales and customers with the goal to gain insights that help to improve the companies productivity and business. One of the standard languages for data science tasks is Python (besides Julia and R). Python has become very popular because of its easy to learn syntax that allows to quickly build prototypical data processing pipelines. One of the most popular libraries for Python in the field of data analytics is Pandas. It allows to easily load data from various sources and represents it in `DataFrames`– a table-like abstraction with column names, types and more meta information. Using Pandas, one can load data in various formats from local or remote locations into `DataFrames` and apply operations such as projection, filter, grouping, join – all well known from the relational algebra. In Pandas, these operations are executed on the local machine of the data scientist and create copies of the data in the local RAM. Since this is slow and means larger-than-RAM data sets cannot be processed easily, data scientists who should actually focus on their data analytics tasks, started to build their custom solutions for parallel processing or buffer management. However, the database community has built fast, robust, and scalable systems to perform exactly this kind of operations efficiently, even if the complete data set does not fit into RAM. Often the data to analyze is already stored in such reliable database systems. Thus, instead of moving the data from the large DBMS server into the potentially small data scientist's laptop for processing, the operations defined in the Python script should be transferred to the DBMS and be translated into a language it understands, i.e. SQL.

During recent years, a few systems have been developed to address these needs, such as RIOT-DB [12], AIDA [3], Modin [9], AFrame [10], and IBIS[1]. IBIS was initiated by the creators of Pandas and tries to overcome the scalability issues of Pandas by using lazy evaluation and converting operations on `DataFrames` into a (sequence of) SQL queries. However, it is not possible to join data from different database systems and UDFs can only be executed using the Pandas backend (i.e. local execution) or on Google Big Query. The `DataFrame` concept has also been adopted by other frameworks like Apache Spark [11], Koalas[2], and Nvidia Rapids[3]. Internally, these systems use optimizers to tune the query and produce good execution plans. The idea of providing a `DataFrame` API over graph data has been studied for example in [2] and [8].

In this paper, we demonstrate our Grizzly[4] framework for transpiling Python code to SQL queries, with the following features:

- Grizzly uses query-shipping and lazy evaluation to achieve high scalability.
- It supports relational operations in standard SQL syntax and uses configurable templates for DBMS-specific dialects without changing the underlying execution system.
- External data can be combined with in-DBMS data.
- Users can define their own functions (UDFs) in Python to apply within the generated query.
- Grizzly uses the UDF mechanism to load and execute pre-trained machine learning models inside the database.

In our demonstration, we show how easy it is to exchange Pandas with Grizzly as the execution engine for `DataFrame` programs. Using a web application, users can compare Pandas and Grizzly scripts, either using our provided examples for various scenarios or writing own code, side-by-side and run them on prepared datasets. Our demo system automatically generates comparison charts for query performance and memory consumption. As these charts are maintained over multiple runs, users can built their own evaluations by varying parameters like the dataset size.

2 REQUIREMENTS

Companies typically store their valuable data in some durable and integer database. This database consists of various tables,

[1] http://ibis-project.org/
[2] https://www.github.com/databricks/koalas
[3] https://developer.nvidia.com/rapids
[4] https://github.com/dbis-ilm/grizzly

containing, e.g., the customer information, the products the company sells, and of course the orders the customers have made over time. This basically resembles the well-known TPC-H [1] benchmark and for larger companies this database can quickly store several GBs. Below we briefly describe use case scenarios from which the requirements for a system that shifts the processing into the DBMS can be derived.

Basic Data Processing. As an example use case, a data scientist may be interested in the names of the customers, who ordered a specific product most often. Using Pandas, the scientist would load the tables `customer`, `orders`, and `products` onto her local workstation and then filter the products by the product name she is interested in, join the remaining products with orders and then with customers. Finally, the join result may be grouped by the customer and aggregated to count how often they have ordered the specific product.

At this point we would like to emphasize that with Pandas, all these processing steps take place in the scientists workstation and not on the actual database server that stores this data. Though, Pandas is able to send a SQL query to a database server in order to fetch proprocessed data. However, we argue that many data scientists often don't know or use SQL and prefer their higher level languages. Thus, to improve the query performance and often make the evaluation possible at all, the computational part should be shifted into the database server while letting the users still express their analysis tasks in Python.

Accessing External Data. After getting these top customers, the data scientist needs to find out how these products are perceived in social media platforms. The company may track social media posts using some additional systems and collects the product reviews in a text file. However, these posts are not critical to the company and are not ingested into the DB. Thus, the data scientist needs to join the data in the DB with the data in files. Currently, this operation would typically also be executed on the workstation, but should optimally be executed by the DBMS. In order to do so, the DBMS must import the file as a (temporary) table. This import should be transparent to the user so that she only needs to specify a file path, but does not have to bother with DBMS specific import code.

UDFs & Model Join. The well-known data processing libraries all provide a vast variety of algorithm implementations and operators for various tasks. However, these may not be enough for every problem to solve and users fall back to implement their own logic in user-defined functions which they want to apply. Such functions could either be applied to every record individually or to one table/data set as a whole. Again, the function could be applied on the workstation using Pandas, but then subsequent operations which could be handled in the DBMS are not possible anymore. Thus, the function's code should also be shipped to the database server transparently for convenient execution.

A special case is the application of some existing machine learning model. An existing model like RoBERTa [6] may be trained to detect the sentiment of some text, i.e. if it is positive or negative. After the data scientist joined the top customers and products with the product reviews from a text file, she needs to classify how the reviews rate the corresponding product. Thus, she needs to join the model with the data (intermediate query result) in the database. For this, it must be possible to select and load an existing model and join it with the data in the database, i.e. apply it to every tuple of a table/intermediate result. In further discussion we name this concept model join.

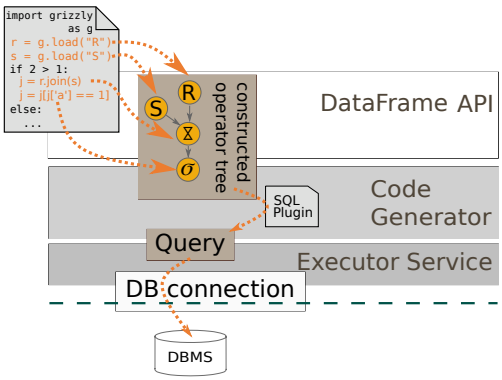
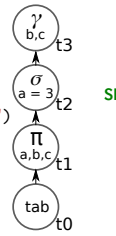


Figure 1: Overview of Grizzly's architecture.

```
# load table (t0)
df = grizzly.read_table("tab")
# projection to a,b,c (t1)
df = df[['a','b','c']]
# selection (t2)
df = df[df.a == 3]
# group by b,c (t3)
df = df.groupby(['b','c'])
```

(a) Source Python code.



(b) Operator tree.

```
SELECT t3.b, t3.c FROM (
  SELECT * FROM (
    SELECT t1.a, t1.b, t1.c
    FROM (
      SELECT * FROM tab t0
    ) t1
  ) t2 WHERE t2.a = 3
) t3 GROUP BY t3.b, t3.c
```

(c) Produced SQL query.

Figure 2: Steps for transpiling Python code to a SQL query: The operations on `DataFrames` (a) are collected in an intermediate operator tree (b) which is traversed to produce a nested SQL query (c).

3 GRIZZLY ARCHITECTURE

The scalability issues of the Pandas framework are a consequence of two major bottlenecks:

- Pandas operations are executed eagerly, producing numerous intermediate results that increase the memory consumption of a Pandas program.
- Pandas uses the data-shipping paradigm, which means that data is transferred to the place where the program is executed. Despite being easy to use, the data-shipping paradigm limits scalability as potentially large amounts of data are transferred. Operators are executed on the client-side, which usually consists of weaker hardware compared to (cloud) servers.

The design of the Grizzly framework focuses on solving these two problems. First, Grizzly replaces the eager operator execution approach with a lazy approach by collecting operators and generating a query when the result is needed. Second, Grizzly combines the convenience of the data-shipping paradigm with the scalability of the query-shipping paradigm by abstracting from data access and pushing query execution to the DBMS.

An overview of the Grizzly architecture is given in Figure 1. Grizzly provides a Pandas-like `DataFrame` API for compatibility with existing Pandas programs. In order to achieve the lazy execution paradigm, Grizzly collects operators and builds a lineage graph as an internal query representation, following a similar approach as RDDs in Apache Spark [11]. Operators are classified as transformations (e.g. projections, filters, joins) or actions (e.g. show, count, sum). While transformations are collected in the lineage graph, actions trigger code generation and execution. In order to meet the second design goal of using the query-shipping

paradigm, Grizzly transpiles the lineage graph of `DataFrame` to standard SQL, compatible with a broad range of DBMSs. The transpilation is achieved by traversing the lineage graph and using a mapping between `DataFrame` operators and SQL query constructs, which is described in detail in [4]. There are two options for incrementally constructing a SQL query:

(1) Incrementally extend `SELECT`, `FROM` and `WHERE` blocks of a (single) SQL query, or
(2) generate a separate query for each operator and nest the existing query in the `FROM` clause.

While producing more compact and easier-to-read queries, option (1) has the drawback that nesting is still required in some cases, e.g. for a filter on a computed column. These cases require special handling in the code generation as well as a careful naming. We observed that modern query optimizers of existing DBMSs have excellent capabilities for unnesting queries, and thus decided to follow option (2) and apply a generic naming schema for sub-queries. Queries are send to the DBMS using a standard connection object as specified by PEP 249[5]. Additionally, Grizzly uses a template file in order to match vendor-specific SQL dialects. An example workflow for transpiling a Pandas program into a nested SQL query is shown in Figure 2.

4 API EXTENSIONS

Modern data analytics tasks not only use traditional operators, but also include the use of external data sources, user-defined functions (UDFs) or machine learning models for prediction or classification as we argued in the use case discussion in Section 2. In order to address these challenges, we extended the `DataFrame` API of Pandas with a set of operators. All additional operators are designed with the goal of hiding complexity and providing an easy-to-use interface for complex operations.

The basic approach of supporting the described features is the generation of additional queries to create functions or define external data sources. Such queries are required to be run before the actual analysis starts. Grizzly maintains a list of pre-queries and whenever the lineage graph traversal reaches an operator that requires a pre-query, it is generated and appended to the list. Finally, all pre-queries are automatically executed before the actual generated query.

External Data Sources. Various database systems offer support for external data sources by creating a table over a file, e.g. using foreign data wrappers in PostgreSQL or external tables in Actian Vector. Similar to ordinary database tables using `read_table`, external tables can be used as leaf nodes in the lineage graph. Grizzly offers the `read_external_table` function for this, which takes the path to the external source as well as the schema as parameters and returns a `DataFrame` for further usage. During code generation, the external table is generated using a pre-query and given a temporary name to be referenced in the actual query.

UDF Support. In Pandas, users can apply custom functions to `DataFrames` using the `apply` function in an elementwise fashion (scalar UDFs) or as a reduce function (table UDFs). Modelling the apply operator as an action, and therefore executing the subquery and applying the UDF at the client side, has the major drawback that further operators also need to be executed at the client side. In order to avoid this, we model UDFs as transformations in Grizzly by exploiting the recent upcome of Python UDF support in database systems. The source code of the UDF is accessed via reflection and transferred to the database system using a UDF creation as a pre-query. The function is created with a generic name and is then applied in the actual query in the projection list. Note that many database systems currently only support scalar UDFs and only offer this feature as a beta version due to security concerns. Consequently, Grizzly is currently also limited to scalar UDFs and requires the vendor-specific activation of the UDF feature in order to support UDFs. As a result of modelling UDFs as transformations, UDF computation can also be pushed to the database, enabling efficient subsequent operations on the UDF result.

```python
# Define conversion functions
def input_to_model(a: str):
        ...
def model_to_output(a) -> str:
        ...

# Use external file
df = grizzly.read_external_table("path/to/file", schema, options)
# Apply onnx classification model using conversion functions
df['class'] = df['text'].apply_model("/path/to/model",
↪  input_to_model, model_to_output)
# Count elements per classification
df = df.groupby('class').count()
# Trigger execution and show result
df.show()
```

Listing 1: Example for external file usage and model join

Machine Learning Model Join. Database systems are a non-optimal environment for training complex machine learning models, as this task is mainly performed on massively parallel engines like GPUs and involve a hybrid workload of intensive computations as well as large updates of e.g. weights in the model. However, there are various pre-trained models available that can be easily used for data analytics, e.g. in the Model Zoo on Github[6]. Applying machine learning models to data is a special case of UDFs and can also be applied to Pandas `DataFrames` using the `apply` function and handcrafted code for model execution. Grizzly offers a family of specialized `apply` functions for the most popular model formats and execution engines PyTorch[7], Tensorflow[8] and ONNX[9]. Similar to UDFs, these operators are modelled as transformations and designed for comfortable usage, demanding only necessary, model type specific parameters from the user. The generated code for model execution (application) is handled like a UDF: it is defined in the database system using a pre-query and used in the projection list of the generated SQL query. For a more detailed discussion of challenges that come with this features and their solutions in Grizzly, we kindly refer to [5].

For the discussion of the presented features we assume that necessary files like data sources or machine learning models are accessible from the database server and that required Python modules are installed. We argue that using cloud file systems or NAS storage this is not a limitation and that root access or contacting the database administrator is currently necessary anyway in order to use the UDF feature.

As an example, Listing 1 shows a program that loads an external file, applies a classification model in the ONNX format on column *text* and then counts the number of entries per class. ONNX models are typically provided with conversion functions to convert an input to a tensor and convert the output tensor

[5]https://www.python.org/dev/peps/pep-0249/

[6]https://www.github.com/onnx/models
[7]https://www.pytorch.org/
[8]https://www.tensorflow.org/
[9]https://www.github.com/onnx/

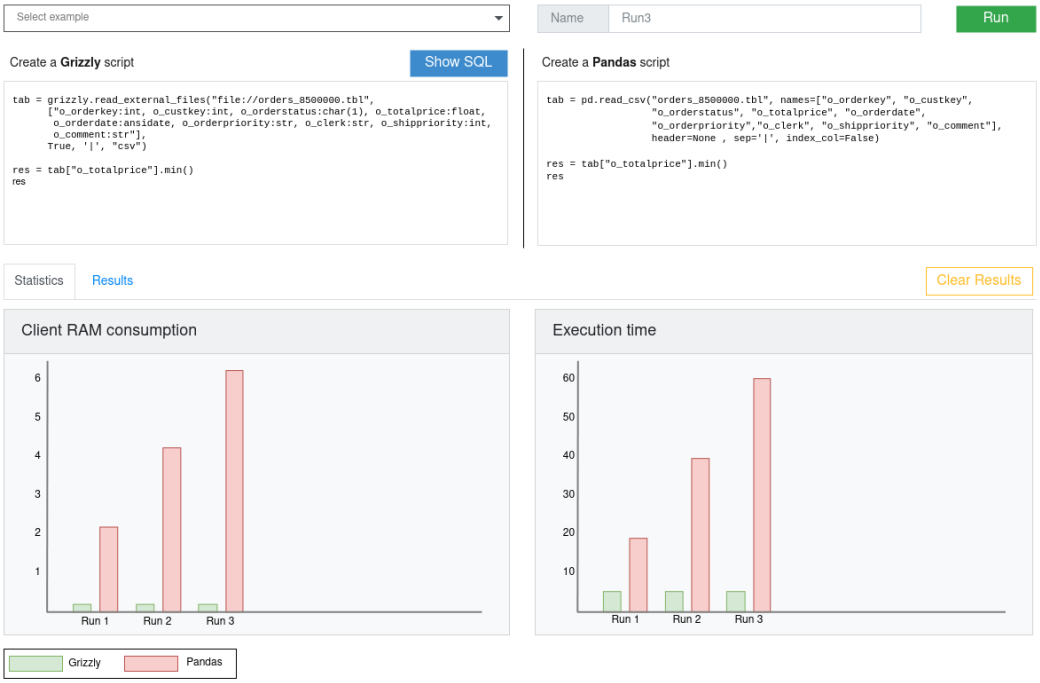


Figure 3: Screenshot of the Grizzly Web Application.

back. The type hints in the signature of these functions are used to determine the type of the overall UDF. The types are required to define the functions in SQL. Grizzly generates pre-queries for the external table and the UDF, ultimately executing three SQL statements.

5 OBSERVING THE BEARS IN THE WILD

The demonstration highlights the benefits users can expect when using Grizzly over Pandas. It invites visitors to interact with the system over a web application and Jupyter notebooks.

The application lets users choose between prepared scenarios, but also allows to run own scripts. For both cases we provide a pre-loaded collection of data sets (from TPC-H [1] and IMDB [7]) with varying sizes. The scenarios range from simple queries to more complex tasks that make use of the features we explained in the previous sections: executing UDFs, combining existing tables with external files, and performing the model join. For every scenario, a prepared Grizzly and Pandas script can be selected and executed. The side-by-side code editors demonstrate how similar the Grizzly and Pandas APIs are, but especially for the model join this will also highlight how much work Grizzly saves developers compared to Pandas. Using the "Show SQL" button, one can inspect the SQL query Grizzly transparently generates for an entered Python program. As an example, the model join scenario offers different models available in ONNX, PyTorch, and Tensorflow format to classify entries in the tables. It follows the use case described in Section 2: We connect to a movie database and join this data with reviews from a text file and classify every movie using a pre-trained model into the categories *positive* and *negative*. The final result is grouped in order to count the number of positive and negative reviews per movie. The Grizzly code is as easy as shown in Listing 1, being significantly smaller than the respective Pandas implementation. Additionally, the application shows the effort to handcraft the SQL code, which is transparently generated by Grizzly. Users can also compare the external table feature of Grizzly to the respective `read_csv` feature of Pandas as sketched in Figure 3 and observe a significant performance gain when using Grizzly. The web application tracks the execution time as well as the memory consumption during the execution of a program over multiple runs and visualizes both metrics for comparison. Through the collected result graphs and by using our input data sets of different sizes, users can build their own evaluation and investigate the scalability of the systems.

A second part of the demonstration uses Jupyter notebooks to demonstrate the easy integration of Grizzly in such environments and how it can be used to process and visualize data and query results interactively.

REFERENCES

[1] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*. Springer, 61–76.

[2] Ankur Dave, Alekh Jindal, et al. 2016. GraphFrames: an integrated API for mixing graph and relational queries. In *GRADES*. ACM, 2.

[3] Joseph Vinish D'silva, Florestan D. De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for advanced in database analytics. *VLDB* 11, 11 (2018), 1400–1413.

[4] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *CIDR*.

[5] Steffen Kläbe and Stefan Hagedorn. 2021. When Bears get Machine Support: Applying Machine Learning Models to Scalable DataFrames with Grizzly. In *BTW*.

[6] Yinhan Liu, Myle Ott, et al. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692

[7] Andrew L. Maas, Raymond E. Daly, et al. 2011. Learning Word Vectors for Sentiment Analysis. In *ACL-HLT*. Portland, Oregon, USA, 142–150.

[8] Aisha Mohamed, Ghadeer Abuoda, et al. 2020. RDFFrames: Knowledge Graph Access for Machine Learning Tools. *Proc. VLDB Endow.* 13, 12 (2020), 2889–2892.

[9] Devin Petersohn, William W. Ma, et al. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046.

[10] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *Big Data*. 359–371.

[11] Matei Zaharia, Mosharaf Chowdhury, et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*.

[12] Yi Zhang, Herodotos Herodotou, and Jun Yang. 2009. RIOT: I/O efficient numerical computing without SQL. In *CIDR*.