Hagedorn, Stefan; Kläbe, Steffen; Sattler, Kai-Uwe:

Putting Pandas in a Box

DOI: 10.22032/dbt.51534

URN: urn:nbn:de:gbv:ilm1-2022200108

This article was first published in the context of the following conference:

2021 Conference on Innovative Data Systems Research (CIDR) : Session 3: Data Analytics.
(Conference on Innovative Data Systems Research (CIDR) ; (Online) : 2021.01.11-15)

Original published: 2021-01-11
URL (Proceedings): http://cidrdb.org/cidr2021/index.html
URL (Paper): http://cidrdb.org/cidr2021/papers/cidr2021_paper07.pdf
[*Visited:* 2022-02-21]

Putting Pandas in a Box

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

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

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

ABSTRACT

Pandas – the Python Data Analysis Library – is a powerful and widely used framework for data analytics. In this work we present our approach to push down the computational part of Pandas scripts into the DBMS by using a transpiler. In addition to basic data processing operations, our approach also supports access to external data stored in files instead of the DBMS. Moreover, user-defined Python functions are transformed automatically to SQL UDFs executed in the DBMS. The latter allows the integration of complex computational tasks including machine learning. We show the usage of this feature to implement a so-called model join, i.e. applying pre-trained ML models to data in SQL tables.

1. INTRODUCTION

Data scientists and analysts regularly need to process large amounts of data. The Python Pandas framework is the defacto standard for reading data from different sources and formats and processing it. The processing scripts can be arbitrarily complex and include various kinds of operations. Typical operations are filtering, projection to certain columns or grouping, but also applying some user defined functions, e.g., for transformations. In recent years, the usage of machine learning (ML) models has become quite popular and several frameworks for Python have been developed to create, train, and apply different artificial neural networks on some input data. Such models can of course also be applied to data initially loaded and pre-processed with Pandas.

However, companies often have their actual data in database management systems (DBMS) for efficient storage, retrieval and processing, rather than in multiple (text) files. Processing such data with Pandas would mean to transfer the (complete) data from the database server to the client machine to process it there in Python. Not only the large transfer costs, but also the limited client memory impacts the usability of this approach, as data might exceed typical client memory amounts. This is reinforced by the fact that Pandas operations often create copies of the internal data and therefore occupy even more of the client's RAM and will eventually fail. A solution would be to not bring the data into the Python program (data shipping), but to bring the program to the data (query shipping). Though, the latter means the data scientists would need to write SQL queries instead of Python programs to access the data, which is often not desired since they are not familiar with SQL or the problem is difficult to express.

In [4] we sketched our initial idea of the Grizzly framework, a transpiler to generate SQL queries from a Pandas-like API. The idea is to let users write a Python program locally and send the generated SQL query to the DBMS storing the data, which is optimized for processing large amounts of data efficiently. In this paper we show the realization of the idea and extend it with new features, namely support for external files and Python functions. Our main contributions are:

- We provide a framework that creates SQL queries from `DataFrame` operations, moving program complexity to the optimized environment of a DBMS.
- Our framework supports the basic API of Pandas. This way, we achieve high scalability and fast performance while remaining the ease-of-use of Pandas.
- We extend the Pandas API with the possibility to process external files directly in the database by using DBMS specific external data source providers. This especially enables the user to join files with the existing data in the database directly in the DBMS.
- We move the complexity of user-defined functions to the DBMS by exploiting the Python UDF support of different DBMSs. This makes it possible to apply pre-trained ML models to the data inside the database.

2. DATA SHIPPING VS. QUERY SHIPPING

Data shipping and query shipping [5] are two major paradigms of data processing. Data shipping means that data is moved to the environment where the computation takes place, which is done in Pandas. On the contrary, query shipping means that the actual program logic is moved to the environment where data is situated in, e.g. a DBMS. In this section, we discuss the advantages and disadvantages of both paradigms and motivate our goal to combine the best of both worlds.

Reasons to use Pandas. The core data structure in Pandas is the `DataFrame`. It is a tabular structure consisting of columns and rows. Columns have a data type and an optional name that can be used to reference columns. Furthermore, a `DataFrame` can have an index column, which can be utilized to directly access tuples/rows by value. The Pandas API

CIDR '21 January 10-13, 2021, Chaminade CA, USA

includes various operations that let users transform, filter, and modify the `DataFrame` contents.

Since Pandas is just a framework to be used within any Python program, it combines the advantages of a procedural programming language like loops, if-else branching or modules and an algebra for data analyses and manipulation:

- The many different functions to access diverse data sources (CSV, JSON, spreadsheets, database tables, and many more) allow to focus on the actual data processing instead of data loading and formatting.
- Pandas operations can be chained to solve a problem step by step. It allows to apply user defined functions to implement custom functionality for modification.
- With the procedural language features, one can react on application parameters, data characteristics, etc. This way complex processing pipelines can be built.

While these advantages let data workers easily create their programs, every operation on a Pandas `DataFrame` is executed eagerly and for many operations, Pandas creates copies of the `DataFrame` contents internally. This means, when chaining a few Pandas operations, many copies of the actual data are created in memory, which needs effort to clean up (if at all). Especially the `read_sql_table` method occupies a large amount of RAM, because it uses other libraries (e.g. `sqlalchemy`) that have their own data wrappers. Thus, the local in-memory processing clearly becomes a bottleneck already when the input data set grows to a few GB.

Reasons to use a DBMS. Relational databases have been around since the 1970s and since then have been highly optimized to deal with large data sets:

- The declarative and standardized query language SQL gives the DBMS the opportunity to "understand" the operations and to optimize the query to reduce latency as well as the overall throughput in the system.
- Indexes can be used to speed up queries. Especially selections and joins benefit from them.
- DBMS are optimized to handle data sets much larger than the available RAM by maintining a buffer pool and implementing buffer eviction strategies. This way, a DBMS will not run out of RAM when processing large data sets. Furthermore, main memory DBMS are highly optimized in terms of memory access and memory utilization (e.g. by data compression) [3]. While Pandas can also chunk the input data set when loading, it has to be implemented manually and burdens the developer with the responsibility to choose an appropriate chunk size. Also, chunking does not work with `read_sql_table`!
- Most DBMS support a cluster setting with distributed storage and query processing. Tables can be partitioned and distributed among several nodes and the DBMS takes care of distributing a query to these nodes to fetch the results.

However, using a full-fledged DBMS is sometimes complicated, too expensive, or simply too much to just process a file-based data set with just a few hundred (thousand) records. Especially in the latter case, the file would have to be imported into the DBMS prior to querying it. User defined functions to apply on the columns need to be created as stored procedures first, too.

Getting the best of both Worlds. On the one hand, DBMSs are able to process large amounts of data very efficiently and easily scale with terabytes of input data due to their highly developed optimizers and efficient storage and buffer management. On the other hand, not every problem can be (easily) expressed in SQL. A procedural programming language with a framework to represent, modify, and query tabular data greatly reduces the development efforts by data workers. Thus, to combine these two worlds, in this project we aim to execute operations on Python `DataFrame`s in a DBMS. This will give users the opportunity to create programs with potentially complex logic that process large data sets. In order to achieve this goal, we need to overcome a major drawback of query shipping, namely the conversion of the client context to the server context. In our example, a mapping from `DataFrame` operations to SQL queries or expressions is needed.

3. GRIZZLY: MAPPING PYTHON OPERATIONS TO SQL

In order to achieve the combination of the two worlds, maintaining the ease-of-use of the data shipping paradigm while achieving the scalability and the performance of the query shipping paradigm, we introduce the Grizzly framework[1], which consists of a `DataFrame` replacement and a SQL code generator. It is intended to solve the scalability issues of Pandas by transforming a sequence of `DataFrame` operations into a SQL query that is executed by a DBMS or even on Apache Spark via SparkSQL. The ultimate goal is to make Grizzly a drop-in replacement for Pandas.

In this section, we describe the main architecture of the Grizzly framework. We show how SQL code generation is realized using a mapping between `DataFrame`s and relational algebra and present our approach to support external files as well as pre-trained ML models.

3.1 Grizzly Architecture

Figure 1 shows the general (internal) workflow of Grizzly. As in Pandas, the core data structure is a `DataFrame` that encapsulates the data. Operations on these `DataFrame`s are tracked and stored in a lineage graph (a directed acyclic graph), building an operator tree. For every operation a new `DataFrame` object is created that has a reference to its parent(s). Initially, such a `DataFrame` represents a table (or view) that is stored in the DBMS. The operations resemble the well-known operations from the relational algebra, and hence, their results are again `DataFrame`s . Exceptions are aggregation functions which are not called in the context of grouping operations as they produce scalar values to summarize the complete content of a table or query result.

Building the lineage graph of `DataFrame` modifications follows the design goal of lazy evaluation. `DataFrame` objects that are created by operations simply record the operation, but do not contain any query results. This is the same evaluation behavior of RDDs in Apache Spark [9]. Apache Spark divides the operations into *transformations* which are just recorded and *actions* that trigger the computations. In our setting, all operations can be recorded as transformations, except for aggregation functions such as `count`, etc. To view the result of queries that do not use aggregation, special actions such as `print` or `show` are available to manually trigger the computation. When an action is encountered in a program, the lineage graph is traversed bottom up, starting from the `DataFrame` on which the action was called.

[1]Available on GitHub: https://github.com/dbis-ilm/grizzly

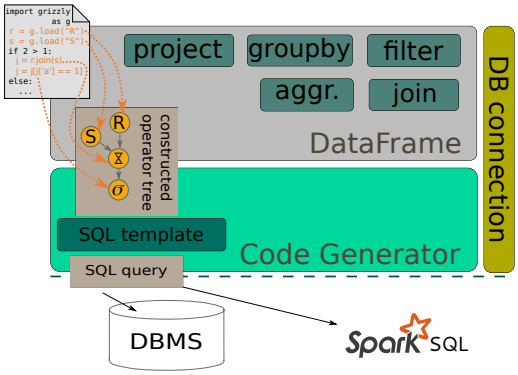

Figure 1: Overview of Grizzly's architecture.

While traversing the tree, for every encountered operation its corresponding SQL expression is constructed as a string and filled in a SQL template. As mentioned in Section 2 we need a mapping from Pandas operations to SQL. Table 1 shows such a mapping.

Based on the lineage graph, the SQL query can be constructed in two ways: *(1)* generate nested sub-queries for every operation on a `DataFrame` , or *(2)* incrementally extend a single query for every operation found in the Python program. We found the first variant the easiest to implement. Variant (2) has the drawback to decide if the SQL expression of an operation can be merged into the current query or if a sub-query has to be created. The SQL parser and optimizer in the DBMSs have been implemented and optimized to recognize such cases. Therefore, in Grizzly we decided to generate nested queries for every `DataFrame` operation. As an example, the following Python code

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

will be transformed into the nested query:

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

The reason we do not try to unnest the queries is to keep code complexity low. Unnesting would mean to check if a referenced column name, e.g., in a `WHERE` clause already really exists or is being computed using a user defined function in the projection. Although the unnesting imposes some overhead to the optimizer in the DBMS, they are mostly well tested and our experiments showed that they can rewrite such nested queries easily to a flat query.

The generated query is sent to a DBMS using a user-defined connection object, as it is typically used in Python and specified by PEP 249[2]. Grizzly produces standard SQL without any vendor-specific SQL dialect, except for statements to create functions or access external data, as we will discuss below. In such cases, the vendor-specific statement template can be defined in a configuration file.

3.2 Accessing External Data

Executing `DataFrame` operations on a SQL database requires that users can specify the tables and views to be used. However, not every data set is ingested into a database system first. Often, especially when data is exchanged with other people, they are exported into text file formats as CSV or JSON and used only a few times. Manually importing these files into a database is not desired and these files should rather be loaded directly. Since it is our goal to shift the complete processing into the DBMS, the files need to be transferred and imported into the DBMS transparently. In our framework, we achieve this by using the ability of many modern DBMS to define a table over an external file. In PostgreSQL this can be achieved by `Foreign Data Wrappers` which can also access data in other DBMSs. Loading flat files is also supported, e.g., in Actian Vector and IBM Netezza, where this feature is called `external table`. If a user references an external file, Grizzly must instantiate the corresponding external table or data wrapper (depending on the DBMS) before the actual query is performed. This is realized using a list of *pre-queries*. Every time an external file access `DataFrame` is found during code generation, a pre-query is generated and appended to this list. This query might be vendor-specific, so the template to create an external data source is taken from the config file. Before the actual SQL query is sent to the database server, all statements in this *pre-query* list are executed on the server to make sure all objects exist when the query is eventually run.

An important point to highlight here is that the database server must be able to access the referenced file. We argue that with network file systems mounts, NAS devices or cloud file systems this is often the case. Even actively copying the file to the database server is not a problem since such data files are rather small, compared to the amount of data stored in the database.

3.3 User-Defined Functions

A big challenge when converting Pandas scripts into SQL queries is that that developers can create and apply custom functions in Python. Such functions typically perform more or less complex computations to transform or combine values. In order to execute these functions within the DBMS, their definitions must be read and transferred to the DBMS. This requires that the Python program containing the Pandas operations can somehow access the function's source code definition. In Python, this can be done via reflection tools[3]. Most DBMS support stored procedures and some of them, e.g. PostgreSQL, Actian Vector, and MonetDB also allow to define them using Python. This way, defined functions in Python can be transferred to the server and dynamically be created as a (temporary) function.

Although the DBMSs support Python as a language for UDFs, SQL is strictly typed language whereas Python is not. In order to get type information from the user's Python function, we make use of *type hints*, introduced in Python 3.5. A Python function using type hints looks like this:

```
def repeat(n: int, s: str) -> str:
  r = n*s # repeat s n times
  return r
```

Such UDFs can be used, e.g., to transform, or in this example case combine, columns using the `map` method of a `DataFrame` :

```
# apply repeat on every tuple using
# columns name, num as input
df['repeated'] = df[['num','name']].map(repeat)
```

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

[3]Using the `inspect` module: https://docs.python.org/3/library/inspect.html#retrieving-source-code

Table 1: Overview of basic Pandas operations on a `DataFrame df` and their corresponding operations in SQL.

	Python Pandas	SQL
Projection	`df['A']` `df[['A','B']]`	`SELECT a FROM ...` `SELECT a,b FROM ...`
Selection	`df[df['A'] == x]`	`SELECT * FROM ...WHERE a = x`
Join	`pandas.merge(df1, df2, left_on="x", right_on="y",how="inner|outer|right|left")`	`SELECT * FROM df1 inner|outer|right|left join df2 ON df1.x = df2.y`
Grouping	`df.groupby(['A','B'])`	`SELECT * FROM ...GROUP BY a,b`
Sorting	`df.sort_values(by=['A','B'])`	`SELECT * FROM ...ORDER BY a,b`
Union	`df1.append(df2)`	`SELECT * FROM df1 UNION ALL SELECT * FROM df2`
Intersection	`pandas.merge(df1, df2, how="inner")`	`SELECT * FROM df1 INTERECTION SELECT * FROM df2`
Aggregation	`df['A'].min() max()|mean()|count()|sum()`	`SELECT min(a) FROM ... max(a)|avg(a)|count(a)|sum(a)`
	`df['A'].value_counts()`	`SELECT a, count(a) FROM ...GROUP BY a`
Add column	`df['newcol'] = df['a'] + df['b']`	`SELECT a + b AS newcol FROM ...`

Using the type hints, Grizzly's code generator can produce the code to create the function on the server. For PostgreSQL, the generated code is the following:

```
CREATE OR REPLACE FUNCTION repeat(n int, s
↪  varchar(1024))
RETURNS varchar(1024)
LANGUAGE plpython3u
AS 'r = n*s # repeat s n times
return r'
```

The command to create the function in the system is taken from a configuration file for the selected DBMS (PostgreSQL, Actian Vector, or MonetDB currently). We then extract the name, input parameters, source code and return type using the `inspect` module and use the values to fill the template. The function body is also copied into the template. Similar to external data sources in Section 3.2, the generated code is appended to the *pre-query* list and executed before the actual query.

The actual `map` operation is translated into a SQL projection creating a computed column:

```
SELECT t0.*, repeat(t0.num, t0.name) as repeated
FROM ... t0
```

As explained above, the previous operation from which `df` was derived will appear in the `FROM` clause of this query.

3.4 Applying Machine Learning Models

The UDFs can be arbitrarily complex and even import additional packages as long as they are installed on the server. We use this opportunity to let users apply pre-trained ML models onto their tables. In the following, we name this operation of applying a model to the data a "model join". Instead of realizing the this over a *map*-function in Pandas, which leads to a client-side execution of the model join and therefore faces the same scalability issues as Pandas, we realize the model join functionality using Python UDFs. As a consequence, we achieve a server-side execution of the model join directly in the database system, allowing automatic parallel and distributed computation.

Note that we talk about the usage of pre-trained models, since database systems are not optimized for model training. However, applying the model directly in the database has the advantage that users can make use of the database functionality to efficiently perform further operations on the model outputs, e.g. grouping or filters. For our discussions, we assume that necessary Python modules are installed the model files are accessible from the server running the DBMS.

Performing a model join on an `DataFrame` triggers the generation of a *pre-query*, which performs the creation of the respective database UDF. As the syntax for this operation is vendor-specific, the template is also taken from the configuration file. The generated code has four major tasks:

1. Load the provided model.
2. Convert incoming tuples to the model input format.
3. Run the model.
4. Convert the model output back to an expected format.

While steps 2-4 have to be performed for every incoming tuple, the key for an efficient model join realization is caching the loaded model in order to avoid unnecessary model loading. (Re-)Loading the model is necessary if it is not cached yet or if the model changed. These cases can be detected by maintaining the name and the timestamp of the model file.

The actual caching mechanism is carefully designed to support arbitrary UDF realizations of the underlying system. For PostgreSQL and Actian Vector, Python UDFs are only available as a beta version. The main reason for this is that there are severe security concerns about using the feature, as especially sandboxing a Python process is difficult. As a consequence, users must have superuser access rights for the database or demand access to the feature from the administrator in order to use the Python UDF feature, and therefore also the model join feature. While this might be a problem in production systems, this should not be an issue in the scientific use cases where Pandas is usually used for data analytics. Additionally, the actual architecture of running Python code in the database differs in the systems. While some systems start a Python process per-query, other systems keep a single process alive over the system uptime. The per-query approach has the advantage that it offers isolation in the Python code between queries, which is important for ACID-compliance. As a drawback, the isolation makes it impossible to cache a loaded model to use it in several queries. As a consequence, loading the model adds significant overhead to every query. On the contrary, keeping the Python process alive allows to load the model only once and use it in several queries until the model changes or the database system is restarted. As a drawback this approach violates isolation, so UDF code has to be written carefully in order to avoid side effects that might impact other queries.

We realize the caching mechanism by attaching the loaded model, the model file name and the model time stamp to

a globally available object, e.g. an imported module. The model is loaded only if the global object has no model attribute for the provided model file yet or the model has changed, which is detected by comparing the cached timestamp with the filesystem timestamp. In order to avoid that accessing the filesystem to get the file modification timestamp is performed for each call of the UDF (and therefore for every tuple), we introduce a magic number into the UDF. The magic number is randomly generated for each query by Grizzly and cached in the same way as the model metadata. In the UDF code, the cached magic number is compared to the magic number passed and only if they differ, the modification timestamps are compared and the cached magic number is overwritten by the passed one. As a result, the timestamps are only compared once during a query, reducing the number of file system accesses to one instead of once-per-tuple. With this mechanism, we automatically support both Python UDF realizations discussed above, although the magic number and timestamp comparisons are not necessary in the per-query approach, as it is impossible here that the model is cached for the first tuple. This is also a drawback of the per-query approach. We exploit the isolation violation of the second approach that keeps the Python process alive and carefully design the model join code to only produce the caching of the model and respective metadata as intended side effects.

In Grizzly, we support three most popular model formats, namely PyTorch[4], Tensorflow[5] and ONNX[6]. For PyTorch and Tensorflow, models can be stored in a serialized way during the training process and reused by loading and restoring the model. However, this needs some additional information from the user, e.g. the model class implementation is needed for PyTorch or name tags for input and output nodes are needed for Tensorflow. The more easy-to-use approach is therefore the ONNX model format, which is a intended as a portable format and comes with it's own runtime environment as a Python package. Thus, only the path to the model needs to be given by the user.

All three model formats have in common, that the user additionally needs to specify model-specific conversion functions for their usage in order to specify how the expected model input is produced and the model output should be interpreted. These functions are typically provided together with the model. With A, B, C, D being a list of data types, these conversion functions have the signature $in\_conv : A \rightarrow B$ and $out\_conv : C \rightarrow D$, if the model converts inputs of type B into outputs of type C. With A and D being set as type hints, the overall UDF signature can be infered as $A \rightarrow D$ as described in Section 3.3

4. EVALUATION

In this Section, we show that the presented Grizzly framework outperforms the Pandas framework. We present different experiments for external data access as well as applying a ML model in a model join. Please note again, that our basic assumption is that data is already stored in a DB. The experiments were run on a server consisting of a Intel(R) Xeon(R) CPU E5-2630 with 2.30 GHz and 128 GB RAM. This server runs Actian Vector 6.0 in a docker container. The client runs on the server host system and connects to the database inside the container. For fairness, we ran the Pandas experiments on the same server machine. As this reduces the transfer costs when reading tables from the database server, this assumption is always in favor of Pandas.

[4]https://www.pytorch.org/
[5]https://www.tensorflow.org
[6]https://www.github.com/onnx/

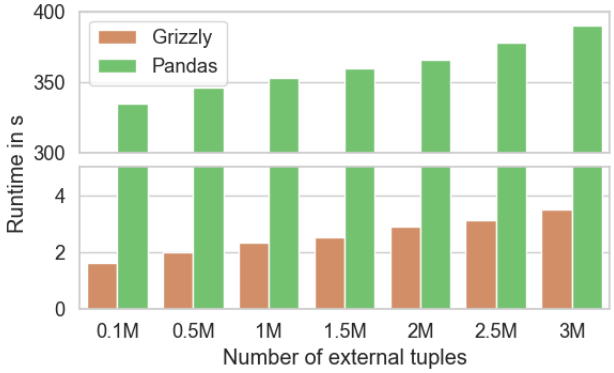

Figure 2: Query runtime with database tables and external sources

4.1 External Data Access

In order to evaluate data access performance, we investigate a typical use case, namely joining flat files with existing database tables. We base our example on the popular TPC-H benchmark dataset on scale factor SF100 [1], which is a typical sales database and is able to generate inventory data as well as update sets. We draw the following use case: The daily orders (generated TPC-H update set) are extracted from the productive system and provided as a flat file. Before loading them into the database system, a user might want to analyze the data directly by combining it with the inventory data inside the database. As an example query, we join the daily orders as a flat file with the customer table (1.5M tuples) from the database and determine the number of orders per customer market segment using an aggregation. For Pandas and Grizzly, the Python script is similar except the data access methods. While Pandas uses a `read_sql_table` and `read_csv` for table and flat file access, Grizzly uses a `read_table` and a `read_external_table` call. This way, an external table is generated in Actian Vector, encapsulating the flat file access. Afterwards, the join as well as the aggregation are processed in the DBMS, and only the result is returned to the client.

For the experiment, we varied the number of tuples in the flat files. The runtime results in Figure 2 show that Grizzly achieves a significantly better runtime than Pandas. Additionally, it shows that Pandas suffers from a bad `read_sql_table` performance, as the runtime is already quite slow for small number of external tuples. Regarding scalability we can observe that runtime in Pandas increases faster with increasing number of external tuples than in Grizzly, caused by the fast processing of external tables in Actian Vector. Therefore, we did not increase the input data size to real large data sets as these small values already show a significant difference between the two systems. Overall we can conclude that Grizzly significantly outperforms Pandas in this experiment and offers the possibility to process significantly larger datasets.

Memory profiling showed that Pandas has a significantly higher memory consumption than Grizzly. For the this test data Pandas required 11 GB RAM, which is mainly caused by the size of the 1.5M. tuples read with `read_sql_table`. Using

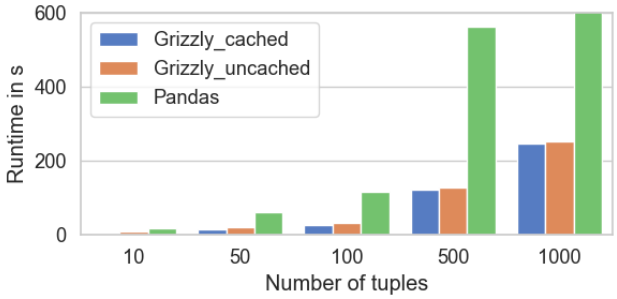

Figure 3: Runtime for model join query.

larger input data would cause the memory consumption to grow beyond the typical RAM of a desktop workstation. The Grizzly client program required only 0.33GB, as it pushes the program logic into the database system that is able to cope with memory shortage using buffering strategies.

4.2 Model Join

As a possible use case for a model join, we evaluated a sentiment analysis on the IMDB dataset [7] using the state of the art RoBERTa model [6] with ONNX. The investigated query applies the model to the review column and groups for the output sentiment afterwards, counting positive and negative review sentiments. Figure 3 shows the resulting runtimes for different number of tuples. We can observe a significant performance gain of using Grizzly (here with Actian Vector as the underlying database system) compared to Pandas for all data sizes. The figure is limited to 600 seconds to make the runtimes for small input sizes visible. With Pandas it takes around 1 second per input tuple. Thus, for an input size of 1000 tuples, Pandas needs 1113 seconds (approx. 19 minutes) to complete. Additionally, with the Python implementation of Actian Vector, which keeps the Python interpreter alive between queries, it is possible to reuse a cached model from a former query, leading to an additional performance gain of around 6 seconds.

5. RELATED WORK

There have been some systems proposed to translate user programs into SQL. The RIOT project [10] proposed the RIOT-DB to execute `R` programs I/O efficiently using a relational database. RIOT can be loaded into an R program as a package and provides new data types to be used, such as vectors, matrices, and arrays. Internally objects of these types are represented as views in the underlying relational database system. This way, operations on such objects are operations on views which the database system eventually optimizes and executes. Another project to perform Python operations as in-database analytics is AIDA [2], which uses NumPy as an extension for linear algebra. The AIDA client API connects to the embedded Python interpreter in the DBMS (MonetDB) to send the program and retrieve the results. The creators of the famous Pandas framework also tackle the problem of the eager client side execution in IBIS[7]. Like Grizzly, IBIS collects operations and converts them into a (sequence of) SQL queries. However, though IBIS can connect to different database systems, it is not possible to combine data from two different systems in a server-side join. Also, UDFs can only be applied when either Pandas (i.e. the program is executed on the client side) or Google's Big Query are used as a backend. Other frameworks and platforms like Apache Spark[8] and Nvidia Rapids[9] also provide a `DataFrame` API which internally is optimized and used for efficient execution. The idea of Grizzly is similar to IBIS and AIDA, as all three systems provide an API similar to the Pandas `DataFrame` API with the goal to abstract from the underlying execution engine.

However, in Grizzly we make use of only the existing features of a DBMS, so that users can use their already existing installations without any modifications. Furthermore, only with Grizzly it is possible to join data sets from different sources inside the *main* database system and it allows to execute pre-trained ML models on the database server, instead of on the client.

6. CONCLUSION

In this paper we have shown how data science and ML frameworks can benefit from utilizing DBMSs by pushing down compute-intensive parts to avoid expensive data transfer into the client program. For this purpose we have a developed a framework to convert operations on `DataFrame`s into SQL queries for lazy evaluation. By exploiting DBMS features such as foreign table interfaces and UDFs we can also provide access to external file-based data and integrate user-defined Python code. Particularly, the latter allows a seamless integration of ML functionalities towards the vision of ML systems [8].

7. REFERENCES

[1] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, volume 8391 of *LNCS*, pages 61–76. Springer, 2013.

[2] J. V. D'silva, F. D. De Moor, and B. Kemme. AIDA - Abstraction for advanced in database analytics. *VLDB*, 11(11):1400–1413, 2018.

[3] F. Faerber, A. Kemper, et al. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[4] S. Hagedorn. When sweet and cute isn't enough anymore: Solving scalability issues in python pandas with grizzly. In *CIDR*, 2020.

[5] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[6] Y. Liu, M. Ott, et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692 [cs]*, July 2019.

[7] A. L. Maas et al. Learning word vectors for sentiment analysis. In *ACL-HLT*, pages 142–150, 2011.

[8] A. Ratner et al. SysML: The new frontier of machine learning systems. *CoRR*, abs/1904.03257, 2019.

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

[10] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O efficient numerical computing without SQL. In *CIDR*, 2009.

[7] http://ibis-project.org/

[8] https://spark.apache.org/

[9] https://developer.nvidia.com/rapids