When Bears get Machine Support: Applying Machine Learning Models to Scalable DataFrames with Grizzly

Steffen Kläbe[1], Stefan Hagedorn[2]

Abstract: The popular Python Pandas framework provides an easy-to-use DataFrame API that enables a broad range of users to analyze their data. However, Pandas faces severe scalability issues in terms of runtime and memory consumption, limiting the usability of the framework. In this paper we present Grizzly, a replacement for Python Pandas. Instead of bringing data to the operators like Pandas, Grizzly ships program complexity to database systems by transpiling the DataFrame API to SQL code. Additionally, Grizzly offers user-friendly support for combining different data sources, user-defined functions, and applying Machine Learning models directly inside the database system. Our evaluation shows that Grizzly significantly outperforms Pandas as well as state-of-the-art frameworks for distributed Python processing in several use cases.

1 Introduction

Python has become one of the most widely used programming language for Data Science and Machine Learning. According to the TIOBE index the languages popularity has steadily grown and was awarded *language of the year* in 2007, 2010, and 2018[3]. The popularity obviously comes from its easy-to-learn syntax which allows rapid prototyping and fast time-to-insight in data analytics.

Python's success is also founded in the vast amount of libraries that help developers in their tasks. Nowadays, the most popular framework for loading, processing, and analyzing data is the Pandas library. Pandas had a huge success as it has connectors to read data in different file formats and represents it in a unified `DataFrame` abstraction. The `DataFrame` implementation keeps data in memory and comes with a variety of operators to filter, transform, join the `DataFrames` or executing different kinds of analytical operations on the data. However, the in-memory processing of Pandas comes with serious limitations and drawbacks:

- Data sizes are limited to the main memory capacity of the client machine, as there is no way of automatic disk-spilling and buffer management as found in almost every database systems.

[1] Technische Universität Ilmenau, Germany, steffen.klaebe@tu-ilmenau.de
[2] Technische Universität Ilmenau, Germany, stefan.hagedorn@tu-ilmenau.de
[3] `https://www.tiobe.com/tiobe-index/python/`, October 2020

doi:10.18420/btw2021-10

- Even if the data to process resides in a database system on a powerful server, Pandas will load all data onto the data scientist's computer for processing. This is not only time consuming, but one can also assume that a companies sales table will quickly become larger than the memory of the data scientist's work station.
- Operations on a Pandas `DataFrame` often create copies of the `DataFrame` instead of performing the operation in place, occupying additional precious and limited memory.

In order to solve the memory problems, many users try to implement their own buffer manager and data partitioning strategies to only load parts of the original input file. However, we believe that scientists trying to find answers in the data should not be bothered with data management and optimization tasks, but this should rather be addressed by the storage and processing system.

Besides data analytics, Machine Learning models have become more and more popular during recent years. There are numerous frameworks for Python to create, train, and apply artificial neural network models on some input data. Often, these Machine Learning frameworks directly support Pandas `DataFrames` as input data. However, the programming effort to apply these models to data situated in arbitrary sources is high and not all users trained the models themselves, but want to use existing pre-trained models in their applications. This use case is therefore also of high importance in the field of data analytics, but not yet integrated into Pandas in an easy-to-use way.

Contribution In this paper we present our Grizzly[4] framework, which provides a `DataFrame` API similar to Python Pandas, but instead of shipping the data to the program, the program is shipped to where the data resides.

In [Hag20] we sketched our initial idea of the Grizzly framework, a transpiler to generate SQL queries from a Pandas-like API. We argue that for many scenarios data is already stored in a (relational) database and used for different applications. Therefore, analysts using this data should neither be bothered with learning SQL to access this data nor with implementing buffer management strategies to be able to process this data with Pandas in Python. In this paper we present an extension to the initial overview in [HK21] by providing API extensions and make the following contributions:

- We present a framework that provides a `DataFrame` API similar to Pandas and transpiles the operations into SQL queries, moving program complexity to the optimized environment of a DBMS.
- The framework is capable of processing external files directly in the database by automatically generating code to use 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.

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

- User-defined functions (UDFs) are also shipped to the DBMS by exploiting the support of the Python language for stored procedures of different database systems. By automatically generating the UDF code to apply the models to the data, Grizzly enables users to apply already trained Machine Learning models, e.g., for classification or text analysis to the data inside the database in a scalable way.

The remainder of the paper is organized as follows: We discuss related work and compare existing systems with a Pandas-like `DataFrame` API in Section 2. In Section 3 we present the architecture of our Grizzly framework as well as the transpilation of `DataFrame` operations to SQL code. Afterwards the important features of Grizzly are explained in detail, namely the external data source support in Section 4, the UDF support in Section 5 and the model join feature in Section 6. We evaluate the performance impact and scalability of Grizzly in Section 7 before concluding in Section 8.

2 Related work

There have been several systems proposed to translate user programs into SQL. The RIOT project [ZHY09] 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 [DDK18], but focuses mainly on linear algebra with NumPy as an extension besides relational algebra. The AIDA client API connects to the AIDA server process running in the embedded Python interpreter inside the DBMS (MonetDB) to send the program and retrieve the results. AIDA uses its TabularData abstraction for data representation which also serves to encapsulate the Remote Method Invocation of the client-server communication.

Several projects have been proposed to overcome the scalability and performance issues in the Pandas framework. These projects can be categorized by their basic approaches of optimizing the Python execution or transpiling the Pandas programs into other languages. Modin [Pet+20] is the state-of-the-art system for the Python optimization approach. By offering the same API as Pandas, it can be used as a drop-in-replacement. In order to accelerate the Python execution, it transparently partitions the `DataFrames` and compiles queries to be executed on Ray [Mor+18] or Dask[5], two execution engines for parallel and distributed execution of Python programs. Additionally, Modin supports memory-spillover, so (intermediate) `DataFrames` may exceed main memory limits and are spilled to persistent memory. This solves the memory limitation problem of Pandas. However, Modin also uses eager execution like Pandas and still requires the client machine to consist of powerful hardware, since data from within a database system is fetched onto the client, too.

[5] `https://www.dask.org/`

In the field of systems that use the transpiling approach, Koalas[6] brings the Pandas API to the distributed spark environment using PySpark. It uses lazy evaluation and relies on Pandas UDFs for transpiling. The creators of the Pandas framework also tackle the problem of the eager client side execution in IBIS[7]. IBIS collects operations and converts them into a (sequence of) SQL queries. Additionally, IBIS can connect to several (remote) sources and is able to run UDFs for Impala or Google BigQuery as a backend. Though, tables from two different sources, such as different databases, cannot be joined within an IBIS program. With a slightly modified API, AFrame [SC19] transpiles Pandas code into SQL++ queries to be executed in AsterixDB. In contrast, in Grizzly we produce standard SQL which can be executed by any SQL engine and use templates provided in a configuration file to account for vendor-specific dialects.

An approach to integrate Machine Learning into columnar database systems was proposed in [Raa+18]. The approach uses handcrafted Python UDFs to train the model inside the database, store the model as a DBMS-specific internal serialized object and apply the model by deserializing it again. In comparison, Grizzly supports pre-trained, portable model formats, automatically generates code to apply the models and also introduces a caching approach to cache the model, which reduces the loading (or deserialization) overhead and is therefore of major importance for deep and complex neural networks.

The main features of the presented systems are compared to Grizzly in Tab. 1. Grizzly also uses the approach of transpiling Python code to SQL, making it independent from the actual backend system and therefore being more generic than Koalas or AFrame. Similar to the proposed systems, Grizzly provides an API similar to the Pandas `DataFrame` API with the goal to abstract from the underlying execution engine. However, Grizzly extends this API with two main features that clearly separates it from the other systems. First, it provides in-DBMS support for external files. This enables server-side joins of different data sources, e.g. database tables and flat files. As a consequence, performance increases significantly for these use cases compared to the client-side join of the sources that would be necessary in the other systems. Additionally, the result of the server-side join remains in the database, enabling subsequent operations to be also executed in the DBMS instead of the client machine. Second, Grizzly offers an easy-to-use API for applying Machine Learning models directly in the DBMS. Compared to the other systems where this feature could be simulated by handcrafting UDF code to apply the model on the client side, Grizzly exploits the UDF feature of DBMS and automatically generates code to cache the loaded model in memory and apply the pre-trained models directly on the server side. Furthermore, applying the model requires several (Python) functions, which can only be realized non-optimally using handcrafted UDFs. Again, as results remain in the DBMS, subsequent operations can be executed efficiently by the DBMS before returning the result to the client. Besides the performance aspect, this feature makes it significantly easier to apply Machine Learning models to the data compared to handcrafted UDFs.

[6] `https://www.github.com/databricks/koalas`
[7] `http://ibis-project.org/`

	Modin	**Koalas**	**IBIS**	**AFrame**	**Grizzly**
Approach	Python optimization	Transpiling	Transpiling	Transpiling	Transpiling
Backends	Ray, Dask	Spark	Arbitrary DBMS with SQL-API	AsterixDB	Arbitrary DBMS with SQL-API
Query Evaluation	Eager	Lazy	Lazy	Lazy	Lazy
UDF Support	On local Pandas `DataFrames`	Over Pandas UDFs	In-DBMS execution for Impala and BigQuery	In-DBMS execution for AsterixDB	In-DBMS execution for Postgres and Vector
Ext. File Support	Read to `DataFrame`	Read to `DataFrame`	Read to `DataFrame`	Read to `DataFrame`	In-DBMS support using ext. tables/ foreign data wrappers
ML Model Support	Handcrafted UDFs	Handcrafted UDFs	Handcrafted UDFs	API for Scikit models	API for ONNX, Tensorflow, PyTorch

Tab. 1: Comparison of available systems with Pandas-like API.

3 Architecture

There are two major paradigms of data processing: data shipping and query shipping [Kos00]. While in the data shipping paradigm, as found in Pandas, the possibly large amount of data is transferred from the storage node to the processing node, in query shipping the query/program is transferred to where the data resides. The latter is found in DBMSs, but also in Big Data frameworks such as Apache Spark and Hadoop.

In this section we discuss the architecture of our Grizzly framework which is designed to maintain the ease-of-use of the data shipping paradigm in combination with the scalability of the query shipping approach. Grizzly is available as Open Source and in its core it consists of a `DataFrame` implementation and a Python-to-SQL transpiler. It is intended to solve the scalability issues of Pandas by transforming a sequence of operations on `DataFrames` into a SQL query that is executed by a DBMS. However, we would like to emphasize that the code generation and execution is realized using a plug-in design so that code generator for other languages than SQL or execution engines other than relational DBMSs (e.g., Spark or NoSQL systems) can be implemented and used. In the following, we show how SQL code generation is realized using a mapping between `DataFrames` and relational algebra.

Figure 1 shows the general (internal) workflow of Grizzly. As in Pandas, the core data structure is a `DataFrame` that encapsulates operations to compute result data. However, in Grizzly a `DataFrame` is only a hull and it does not contain the actual data. Rather, the operations on a `DataFrame` only create specific instances of `DataFrames`, such as

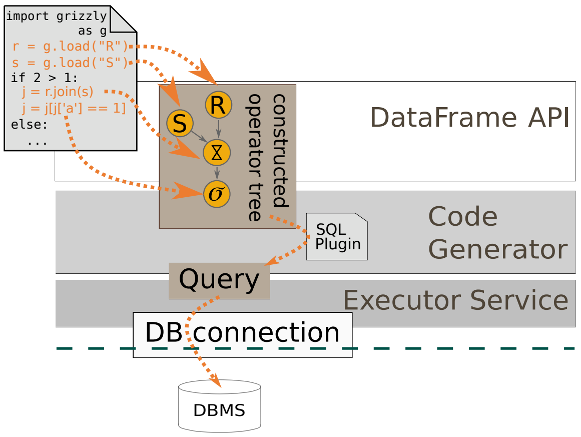


Fig. 1: Overview of Grizzly's architecture.

`ProjectionDataFrame` or `FilterDataFrame`, to track the operations. A `DataFrame` instance stores all necessary information required for its operation as well as the reference to the `DataFrame` instance(s) from which it was created. This lineage graph basically represents the operator tree as found in relational algebra. The leaves of this operator tree are the `DataFrames` that represent a table (or view) or some external file. Inner nodes represent transformation operations, such as projections, filters, groupings, or joins, and hence, their results are `DataFrames` again. The actual computation of the query result is triggered via actions, whose results are directly needed in the client program, e.g., aggregation functions which are not called in the context of `group by` clause. 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.

Building the lineage graph of `DataFrame` modifications, i.e., the operator tree, follows the design goal of lazy evaluation behavior as it is also found in the RDDs in Apache Spark [Zah+12]. When an action is encountered in a program, the operator tree is traversed, starting from the `DataFrame` on which the action was called. While traversing the tree, for every encountered operation its corresponding SQL expression is constructed as a string and filled in a SQL template. For this, we apply a mapping of Pandas operations to SQL statements. This mapping is shown in Table 2. Based on the operator tree, 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.

In Grizzly we implement variant (1), because variant (2) has the drawback to decide whether the SQL expression of an operation can be merged into the current query or a sub-query has to be created. Though, the SQL parser and optimizer in the DBMSs have been implemented and optimized to recognize such cases. As an example, Figure 2 shows how a Python script

	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 df`
		`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`
		`INTERSECTION SELECT * FROM df2`

Aggregation	`df['A'].min()`
	`max()|mean()|count()|sum()`
		`SELECT min(a) FROM ...`
		max(a)|avg(a)|count(a)|sum(a)

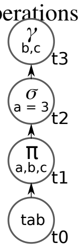
	`df['A'].value_counts()`	`SELECT a, count(a) FROM ...`
		`GROUP BY a`

Add column	`df['new'] = df['a'] + df['b']`	`SELECT a + b AS new FROM ...`

Tab. 2: Basic Pandas `DataFrame` operations and their corresponding SQL statements.

```
# 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.

Fig. 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).

is transformed into a SQL query. Although the nested query imposes some overhead to the optimizer for unnesting in the DBMS and bears the risk that it fails to produce an optimal query, we believe they are very powerful and mostly well tested, so that it is not worth it to re-implement such behavior in Grizzly. 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[8]. Grizzly produces standard SQL with vendor-specific statements to create functions or access external data as we will discuss below. The vendor-specific statements are taken from templates defined in a configuration file. By providing templates for the respective functions one can easily add support for arbitrary DBMSs. We currently support Actian Vector and PostgreSQL.

Besides the plain SQL queries, Grizzly needs to produce additional statements in order to

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

set up user-defined functions and connectors to external data sources. If the Python code uses, e.g., UDFs, this function must be created in the database system before it can be used in the query. Thus, Grizzly produces a list of so-called pre-queries. A pre-query to create a UDF is the `CREATE FUNCTION` statement including the corresponding function name, input and output parameters as well as the function body of course. For an external data source, the pre-query creates the necessary DBMS-specific connection to the data source, as described in the next section.

One design goal of the Grizzly framework is to serve as a drop-in replacement for Pandas in the future. Being under active development, we did not yet reach a state of full API compatibility. For operations that are not supported yet or can not be expressed in SQL, one might fallback to either Pandas operations by triggering the execution inside the DBMS and proceed with the intermediate result in Pandas, or exploit the Python UDF feature of modern DBMS to execute operations as described in Section 5.

4 Support for External Data Sources

In typical data analytics tasks data may be read from various formats. On the one hand, (relational) database systems are used to store and archive large data sets like company inventory data or sensor data in IoT applications. On the other hand, data may be created by hand, exported from operational systems or shared as text files like CSV or JSON. For these files it is not always necessary, intended or beneficial to import them into a database system first, as they might be only for temporary usage or need to be analyzed before loading them into the database. As a consequence, there is a gap between tables stored in a database system and plain files in the filesystem, and both sources need to be combined. In Pandas, one would need to read the data from the database as well as the text files and combine them in main memory. 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 as defined in the SQL/MED standard from 2003 [Mel+02].

As an example, PostgreSQL offers *foreign data wrappers* (FDW) to access external sources such as files, but also other database systems. A PostgreSQL distribution includes FDWs for, e.g., CSV files, files in HDFS as well as other relational and non-relational DBMSs. Own FDWs for other sources can easily be installed as extensions. Internally, the planner uses the access costs to decide for the best access path, if such information is provided by the FDW.

Besides PostgreSQL, Actian Vector offers the external table feature, which is realized using the Spark-Vector-Connector[9]. Here Vector handles external tables as meta data in the catalog with a reference to a file path in the local filesystem or HDFS. Whenever a query accesses an external table, Vector exploits the capabilities of Apache Spark to read data efficiently in parallel, leading to fast scans of external tables.

[9] `https://github.com/ActianCorp/spark-vector`

In Grizzly, we offer easy-to-use operations to access external data sources. These operations return a `DataFrame` object and are the leaves of the operator lineage graph described in Section 3, similar to ordinary database tables. Here a user has to specify the data types of the data in the text files as well as the file path. During SQL code generation, a pre-query is automatically generated that creates the external table/foreign data wrapper for each of these leaves. As the syntax of these queries might be vendor-specific, we maintain templates to create an external data source in the configuration file. The pre-queries are then appended to the pre-query list described in Section 3, so they are ensured to be executed before the actual analytical query is run. In the actual query, these tables are then referenced using their temporary names.

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 server is not a problem since such data files are rather small, compared to the amount of data stored in the database.

5 Support for Python UDFs

Another important part of data analytics is data manipulation using user-defined functions. In pandas, users can create custom functions and apply them to `DataFrames` using the `map` function. Such functions typically perform more or less complex computations to transform values or combine values of different columns. These UDFs are a major challenge when transpiling Pandas code to SQL, as 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[10]. Most DBMS support stored procedures and some of them, e.g., PostgreSQL and Actian Vector, also allow to define them using Python (language PL/Python). This way, functions defined in Python are processed by Grizzly, transferred to the DBMS and dynamically created as a (temporary) function. Note that most systems only offer scalar UDFs at the moment, which produce a single output tuple from a single input tuple. Consequently, Grizzly only supports this class of functions and does not offer any support for table UDFs, which produce an output tuple for an arbitrary number of inputs.

The actual realization of the UDF support is hereby different for different DBMS and shows some limitations that needs to be considered. First, Python UDFs are only available as a beta version in PostgreSQL and Actian Vector. 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. While this might be a problem in production systems, we argue that this should not be an issue in the scientific use cases where Python Pandas is usually used for data analytics.

[10] Using the `inspect` module: `https://docs.python.org/3/library/inspect.html`

Second, the actual architecture of running Python code in the database differs in the systems. While some systems start Python processes per-query, other systems keep processes 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 user-specific data structures in order to use it in several queries, which is of major importance when designing the model join feature in Section 6. On the contrary, keeping the Python processes alive allows to cache such a user context and use it in several queries. However, this approach violates isolation, so UDF code has to be written carefully to avoid side effects that might impact other queries.

Although the DBMS supports Python as a language for user defined code, SQL is a 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)
```

Using the type hints and a mapping between Python and SQL types, Grizzly's code generator can produce a pre-query 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'
```

Currently, we statically map variable-sized Python types to reasonable big SQL types, which is a real limitation and should be improved in the future. The command to create the function in the system is vendor-specific and therefore taken from the config file for the selected DBMS. We then extract the name, input parameters, source code and return type using Python's `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 4, the generated code is appended to the pre-query list and executed before the actual query. The `map` operation is translated into a SQL projection creating a computed column in the actual query:

```
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.

6 Machine Learning Model Join

In Grizzly, we expand the API of Python Pandas with the functionality to apply different types of Machine Learning models to the data. In the following, we name this operation of applying a model to the data a “model join”. Instead of realizing 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 exploit the recent upcome of user-defined functions in popular database management systems and 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 in this section, as 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. Additionally, users may use publicly available, pre-trained models for various use cases. For our discussions, we assume that necessary Python modules are installed and the model files are accessible from the server running the database system. In the following, we describe the main ideas behind the model join concept as well as details for the supported model types, their characteristics and their respective runtime environments, namely PyTorch, Tensorflow, and ONNX.

6.1 Model join concept

Performing a model join on an `DataFrame` triggers the generation of a pre-query as described in Section 3, 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 hereby 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 output 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 perform the expensive loading only if necessary. (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. However, such a caching mechanism must be designed carefully under consideration of the different, vendor-specific Python UDF realizations discussed in Section 5.

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 in the UDF. 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 in Section 5, 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. 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.

6.2 Model types

Grizzly offers support for PyTorch[11], Tensorflow[12] and ONNX[13] models. 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 by creators. With A, B, C, D being lists of data types, the conversion functions have signatures $in\_conv : A \rightarrow B$ 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 5. With this, applying a model to data stored in a database can be done easily and might look like the example in Listing 1.

As the conversion functions are typically provided along with the model, users only need to write a few lines if code. It is thereby mandatory to specify the input parameter types of the `input_to_model` function as well as the output type of the `model_to_output` function. In this example, the resulting UDF would have signature `str -> str`. Running this example, Grizzly automatically generates the UDF code and triggers its creation in the DBMS before executing the actual query. The produced query along with the pre-query to setup the UDFs in PostgreSQL is shown in Listing 2.
The actual code for model application is generated from templates and varies for the different model types described in the following.

PyTorch The PyTorch library is based on the Torch library, originally written in Lua. PyTorch was presented by Facebook in 2016 and has gained popularity as it enabled

[11] `https://www.pytorch.org/`
[12] `https://www.tensorflow.org/`
[13] `https://www.github.com/onnx/`

```
def input_to_model(a: str):
        ...

def model_to_output(a) -> str:
        ...

df = grizzly.read_table('tab') # load table
# apply model to every value in column 'col'
# using provided input and output conversion functions
# store model output in computed column 'classification'
df['classification'] = df['col'].apply_model("/path/to/model", input_to_model,
↪  model_to_output)
# group by e.g. predicted classes
df = df.groupby(['classification']).count()
df.show()
```

Listing 1: Python code of model join example

```
CREATE OR REPLACE FUNCTION apply_model_123(col varchar(1024))
 RETURNS varchar(1024)
 LANGUAGE plpython3u AS 'def input_to_model(a: str):
        ...

 def model_to_output(a) -> str:
        ...

  #apply model here
' parallel safe;

SELECT t2.classification, count(*) FROM (
  SELECT *, apply_model_123(t1.col) as classification FROM (
    SELECT * FROM tab t0
  ) t1
) t2 GROUP BY t2.classification
```

Listing 2: Generated SQL code of model join example

programs to utilize GPUs and integrate other famous Python libraries. In its core, PyTorch consists of various libraries for Machine Learning that have different functionality.

A trained model can be saved for later reuse. For saving, two options exist. The first option is to serialize the complete model to disk. This has the disadvantage that the model class definition must be available for the runtime when the model is loaded. In Grizzly, this would mean that users who want to use a pretrained model in their program also need the source code of the model class. The second option for storing a trained model is to store only the learned parameters. Although this option is more efficient during deserialization, the user code must explicitly create an instance of the model class. Thus, the source code of the model class must be available for the end user. Additionally, in order to instantiate the model class, users need to provide initial parameters values to the model's constructor which may be unknown and hard to set for inexperienced users.

Besides the challenges for using PyTorch with foreign models, Grizzly allows to load PyTorch models where only the learned parameters have been stored (option 2 from above).

Tensorflow Tensorflow is a another famous framework for building, training and running Machine Learning models. Tensorflow models are directed, acyclic graphs (DAGs) that are statically defined. With placeholder variables attached to nodes of the model graph, inputs and outputs can be mapped to the graph nodes. A `tensorflow.Session` object is used as the main entrance point and allows to run the model after configuring.

During the training phase, arbitrary states of the model graph can be exported as a `checkpoint`, which is a serialized format of the graph and its properties. Grizzly supports these `checkpoints` as an model type and generates code to restore the model graph as well as the `tensorflow.Session` object. However, users have to know the names of placeholders defined in the model in order to map inputs and outputs to the respective model nodes. Additionally, users can specify a vocabulary file to translate inputs to an expected model input format if necessary. As these restrictions require in-depth knowledge about the model, Grizzly offers additional possibilities to automatically generate the conversion functions. Nevertheless, this harms the ease-of-use slightly.

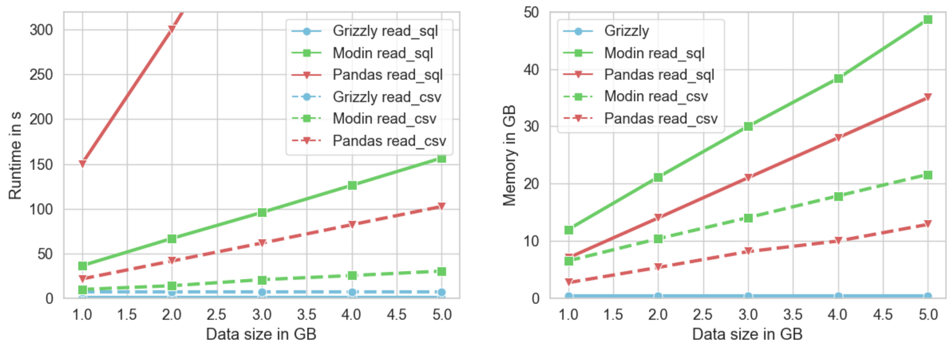
Starting with version 2, Tensorflow offers different possibilities to exchange trained models with the introduction of the Tensorflow Hub[14] library or support for the Keras[15] framework. In the future, we aim at integrating support for these formats in Grizzly.

ONNX ONNX is a portable and self-contained format for model exchange, that is able to be executed with different runtime backends like Tensorflow, PyTorch or the onnxruntime[16]. The self-containment and the portability of models makes the ONNX format easy to use, which meets the design goals of Grizzly. In the generated model code, we rely on the onnxruntime as execution backend in order to be independent from Tensorflow or PyTorch. A broad collection of pre-trained models along with their conversion functions is available in the Model Zoo[17].

7 Evaluation

In this Section, we compare our proposed Grizzly framework against Pandas version and Modin, the current state-of-the-art framework for distributed processing of Pandas scripts, as the other related systems presented in Section 2 do not offer all evaluated features. We present different experiments for data access as well as applying a machine learning model

[14] `https://www.tensorflow.org/hub`
[15] `https://www.keras.io/`
[16] `https://www.github.com/microsoft/onnxruntime`
[17] `https://www.github.com/onnx/models`



(a) Execution time (b) Main memory consumption

Fig. 3: CPU and RAM consumption for Pandas, Modin and our proposed Grizzly framework.

in a model join. Our experiments were run on a server consisting of a Intel(R) Xeon(R) CPU E5-2630 with 24 threads at 2.30 GHz and 128 GB RAM. This server runs Actian Vector 6.0 in a docker container, Python 3.6 and Pandas 1.1.1. Additionally we used Modin version 0.8 and experimentally configured it to the best of our knowledge, resulting in using Ray as the backend, 12 cores and out-of-core execution. For fairness, we ran the Pandas/Modin experiments on the same machine. As this reduces the transfer costs when reading tables from the database server, this assumption is always in favor of Pandas and Modin.

During our experiments, we discovered a bug in the parallel `read_sql` implementation of Modin, which produces wrong results for partitioned databases. The developers confirmed the issue and planned a fix for the next release. However, we used the parallel `read_sql` in order to not penalize Modin in the experiments, not considering the wrong results. This assumption is therefore also in favor of the Modin results.

7.1 Data access scalability

In this first experiment, we want to prove our initial consideration of Pandas' bad scalability with a minimal example use case. We used Actian Vector as the underlying database system and ran a query that scans data with varying size from a table or a `csv` file and performs a `min` operation on a column to reduce the result size. This way, we can compare the basic `read_sql` and `read_csv` operations of Pandas and Modin against Grizzly.

Figure 3 shows the execution time as well as the memory consumption of the evaluated query. For `sql table` access, Pandas shows an enormous runtime, linearly growing to 800 s for a data set size of 5 GB. Modin is significantly faster than Pandas and scales better. However, Grizzly is able to answer this query in a constant, sub-second runtime, as only the result has to be transfered to the client instead of the full dataset. Additionally, this query can be answered by querying small materialized aggregates [Moe98], which are used by default as an additional index structure in Vector. In comparison to Pandas/Modin, this shows that with Grizzly queries can also benefit from index structures and other techniques to accelerate query processing used in database systems. For `csv` access, we can observe a similar behavior of Modin being faster and scaling better than Pandas due to its parallel `read_csv` implementation. Grizzly again shows a nearly constant runtime slightly higher than the `sql table` access but faster than Modin and Pandas. The main reason for this is that Grizzly uses the external table feature of Actian Vector, which is based on Apache Spark. As a consequence, the runtime is composed of the fixed Spark delays like startup or cleanup [WK15] and a variable runtime for reading the file in parallel. As data sized are small here, the fixed Spark delays dominate the runtime.

Regarding memory consumption, Pandas again scales very poorly and memory consumption increases very fast, with the `read_sql_table` consuming more memory than the `read_csv` operation for the same data size. As a result, the memory consumption might exceed the available RAM of a client machine even for a small dataset size. In comparison, Modin consumes even more memory than Pandas for `read_sql` and `read_csv` respectively, potentially caused by the multiple worker threads. The memory consumption of Grizzly is mainly impacted by the result size, which is very small due to the choice of the query. However, this is only half of the truth, as Grizzly shifts the actual work to the DBMS which also consumes memory. Nevertheless, modern DBMS are designed for high scalability and are able to handle out-of-memory cases with buffer eviction strategies in the bufferpool or disk-spilling operators for database operators with a high memory consumption. Therefore, this is the ideal environment to run complex queries, as it is not limited to the available memory of the machine. Note that Modin also supports out-of-memory situations by disk-spilling. Another advantage of Grizzly is that the DBMS can run on a remote machine while the actual Grizzly script is executed from a client machine, which is then allowed to have an arbitrary hardware configuration while still being able to run complex analytics.

7.2 Combining data sources

An important task of data analysis is combining data from different data sources. We investigated a typical use case, namely joining flat files with existing database tables. We base our example on the popular TPC-H benchmark dataset [BNE14] on scale factor SF100, 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. The evaluated Python scripts are similar except the data access methods. While Pandas and Modin use (parallel) `read_sql` 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.

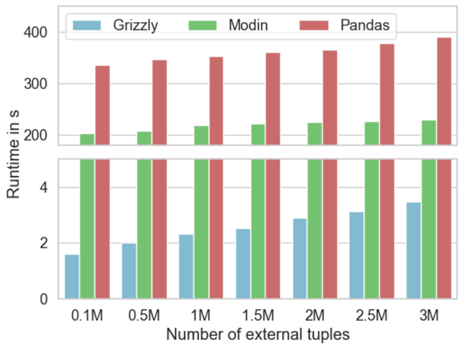


Fig. 4: Query runtime with database tables and external sources

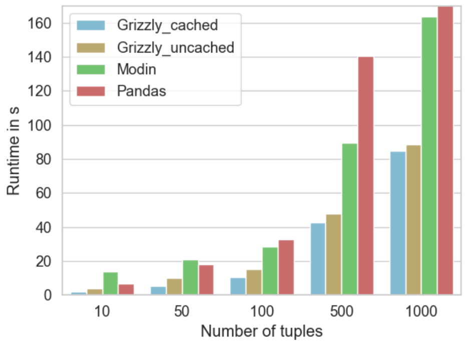


Fig. 5: Runtime for model join query.

For the experiment, we varied the number of tuples in the flat files. The runtime results in Figure 4 show that Grizzly achieves a significantly better runtime than Pandas and Modin. Additionally, it shows that Pandas and Modin suffer from a bad `read_sql` performance, as the runtime is already quite slow for small number of external tuples. Regarding scalability we can observe that runtime in Pandas grows faster with increasing number of external tuples than in Grizzly, caused by the fast processing of external tables in Actian Vector. Overall we can conclude that Grizzly significantly outperforms Python Pandas and Modin in this experiment and offers the possibility to process significantly larger datasets.

7.3 Model Join

There are various applications and use cases where machine learning models can be applied. As an example use case, we applied a sentiment analysis to a string column. We therefore used the IMDB dataset [Maa+11], which contains movie reviews, and applied the state-of-the-art RoBERTa model [Liu+19] with ONNX. The investigated query applies the model to the review column and groups on the output sentiment afterwards, counting positive and negative review sentiments. In Grizzly, we therefore use model join feature described in Section 6, while we handcrafted the function to apply the model for Modin and Pandas and invoked the function over the `map` function on the `DataFrames` after reading from the database.

Figure 5 shows the resulting runtimes for different number of review tuples. First, we can observe that Grizzly significantly outperforms Modin and Pandas in terms of runtime and scalability. For increasing data size Modin is also significantly faster and scales better than Pandas, while showing some overhead over Pandas for the very small data sets, as parallelism here introduces more overhead than benefit. While Modin achieves a speedup by splitting the `DataFrames` into partitions and applying the machine learning model in parallel, Grizzly achieves the performance improvement by applying the model directly in the database system. In Actian Vector, we used a parallel UDF feature and configured it for 12 parallel UDF workers, showing the best results for our setup in this experiment.

Additionally, with the Python implementation of Actian Vector, which keeps the Python interpreters alive between queries, it is possible to reuse a cached model from a former query, leading to an additional performance gain of around 5 seconds.

7.4 Resume

In our evaluation, we proved the superiority of our proposed Grizzly framework over Pandas and Modin, the existing state-of-the-art distributed `DataFrame` framework. In different experiments for data access, the combination of different data sources and the application of pre-trained machine learning models to a dataset we showed that Grizzly has significantly better query performance as well as scalability, as most operations are executed directly in the database system. This is reinforced by the fact that query performance benefits from index structures of the DBMS, while these indexes do not have any impact when only reading data in a Pandas script and performing operations on the client. Furthermore, Grizzly enables complex data analysis tasks even on client machines without powerful hardware by pushing operations towards database servers. Consequently, queries are not limited to client memory, but are executed inside database systems that are designed to handle out-of-memory situations.

8 Conclusion

In this paper we presented Grizzly, a scalable and high-performant data analytics framework that offers a similar `DataFrame` API like the popular Pandas framework. As Pandas faces some severe scalability problems in terms of memory consumption and performance, Grizzly transpiles the easy-to-write Pandas code into SQL in order to push complexity to arbitrary database systems. This way, query execution is done in a scalable and highly optimized environment that is able to handle large datasets and complex queries. Additionally, by pushing queries towards remote database systems, complex analytics can be performed on client machines without extensive hardware requirements.

We extended the Pandas `DataFrame` API with several features in order to perform typical data analytics challenges in an efficient and easy-to-use way. First, Grizzly provides support for external data sources by exploiting the respective feature of several database systems and automatically generating code to create necessary tables or wrappers. This way, different sources like database tables or flat files can be combined and processed directly inside the DBMS instead of loading them into the client, improving performance and scalability. Second, Grizzly makes use of the recent upcome of Python user-defined functions in database systems in order to transparently push the execution of such functions on `DataFrames` into the DBMS. Third, applying pre-trained Machine Learning models to data is supported by Grizzly by automatically generating UDF code for Tensorflow, PyTorch or ONNX models. Pushing these operations to the database system not only increases performance and scalability, but also enables efficient processing of further operations on the model outputs, as they still remain in the database system. This allows a seamless integration of Machine Learning functionalities towards the vision of ML systems [Rat+19].

In our evaluation, we compared our proposed Grizzly framework to Pandas and Modin, the current state-of-the-art framework for distributed execution of Pandas-like DataFrames. In our experiments on data access scalability, combining different data sources, and applying Machine Learning models we proved that Grizzly significantly outperforms both systems while offering higher scalability and an easy-to-use API.

For the execution of UDFs and the application of Machine Learning models we rely on the Python UDF realization of database vendors. Most of them released Python UDFs only as a beta version until now as they faced security issues as well as performance issues. In the future, we monitor the development of this feature and aim at actively removing UDF execution performance as a choke point of query performance.

Additionally, we plan to investigate a different way of handling heterogeneous data sources in the future. Users may use hybrid warehouse approaches consisting of cloud database instances as well as on-premise instances to ensure data privacy and data security. This challenges a frontend framework like Grizzly to query multiple database instances and combine the results on the client side. In a conceptual view, we plan to extend Grizzly with an embedded query engine, e.g. DuckDB [RM19] or MonetDBLite [RM18], and a query optimizer and compare it against existing solutions like the Avalanche hybrid data warehouse[18] or Polystores [Gad+16]. In the query graphs that Grizzly is based on, join operations between different database instances can then be seen as "pipeline breakers", where data needs to be fetched from both join sides and combined locally. Using the query optimizer, computation effort needs to be pushed into the database instances as much as possible to reduce intermediate result sizes, transfer costs and the client-side processing costs.

Grizzly can be used in Jupyter Notebooks, where operations are typically performed incrementally. This incremental way of computation offers possibilities to further optimize the Grizzly workflow by, e.g., exploiting materialized views for intermediate results.

References

[Moe98] Guido Moerkotte. "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing". In: *PVLDB*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 476–487.

[Kos00] Donald Kossmann. "The State of the Art in Distributed Query Processing". In: *ACM Comput. Surv.* 32.4 (Dec. 2000), pp. 422–469. ISSN: 0360-0300.

[Mel+02] Jim Melton et al. "SQL/MED - A Status Report". In: *SIGMOD Rec.* 31.3 (2002), pp. 81–89.

[ZHY09] Yi Zhang, Herodotos Herodotou, and Jun Yang. "RIOT: I/O efficient numerical computing without SQL". In: *CIDR*. 2009.

[18] `https://www.actian.com/analytic-database/avalanche/`

[Maa+11] Andrew L. Maas et al. “Learning Word Vectors for Sentiment Analysis”. In: *Proc. of the 49th Annual Meeting of the Assoc. for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA, 2011, pp. 142–150.

[Zah+12] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *USENIX*. 2012.

[BNE14] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. en. In: *Performance Characterization and Benchmarking*. Springer, 2014, pp. 61–76.

[WK15] K. Wang and M. M. H. Khan. “Performance Prediction for Apache Spark Platform”. In: *HPCC/CSS/ICESS*. 2015, pp. 166–173.

[Gad+16] Vijay Gadepally et al. “The BigDAWG Polystore System and Architecture”. In: *HPEC* (Sept. 2016), pp. 1–6.

[DDK18] Joseph Vinish D’Silva, Florestan D. De Moor, and Bettina Kemme. “AIDA - Abstraction for advanced in database analytics”. In: *VLDB* 11.11 (2018), pp. 1400–1413. ISSN: 21508097.

[Mor+18] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. en. In: *arXiv:1712.05889 [cs, stat]* (Sept. 2018). arXiv: 1712.05889.

[RM18] Mark Raasveldt and Hannes Mühleisen. *MonetDBLite: An Embedded Analytical Database*. 2018. arXiv: `1805.08520 [cs.DB]`.

[Raa+18] Mark Raasveldt et al. “Deep Integration of Machine Learning Into Column Stores”. In: *EDBT*. OpenProceedings.org, 2018, pp. 473–476.

[Liu+19] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: `1907.11692 [cs.CL]`.

[RM19] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. en. In: *SIGMOD*. Amsterdam, Netherlands: ACM Press, 2019, pp. 1981–1984. ISBN: 978-1-4503-5643-5.

[Rat+19] A. Ratner et al. “SysML: The New Frontier of Machine Learning Systems”. In: *CoRR* abs/1904.03257 (2019). _eprint: 1904.03257.

[SC19] Phanwadee Sinthong and Michael J. Carey. “AFrame: Extending DataFrames for Large-Scale Modern Data Analysis”. In: *Big Data*. Dec. 2019, pp. 359–371.

[Hag20] Stefan Hagedorn. “When sweet and cute isn’t enough anymore: Solving scalability issues in Python Pandas with Grizzly”. In: *CIDR*. 2020.

[Pet+20] Devin Petersohn et al. “Towards Scalable Dataframe Systems”. en. In: *arXiv:2001.00888 [cs]* (June 2020). arXiv: 2001.00888.

[HK21] Stefan Hagedorn and Steffen Kläbe. “Putting Pandas in a Box”. In: *CIDR*. 2021.