Making DataFrames Get A Move On

DBMS Support for DataFrame Operations

<u>Stefan Hagedorn</u>
TU Ilmenau

Steffen Kläbe
Actian / TU Ilmenau

Kai-Uwe Sattler
TU Ilmenau

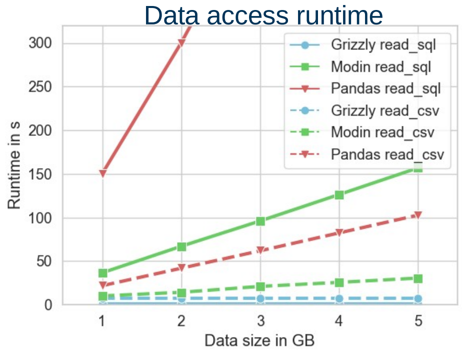
FG DB Spring Symposium March 2022
Challenges and Opportunities in the Data Cloud
HPI Potsdam

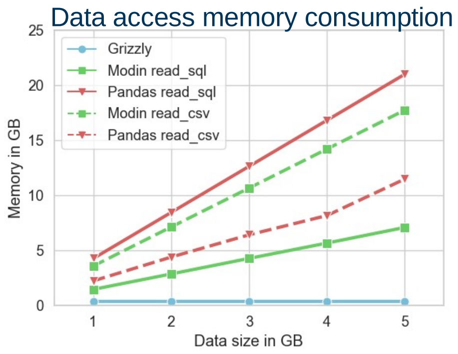
TECHNISCHE UNIVERSITÄT ILMENAU

Pandas

- eager execution of operations – limited possibilities for optimizations
- Operations create copies of intermediate results

  – high memory pressure, limited scalability
- Data is transferred to the client machine (if stored in a DBMS)
  - High transfer costs
  - DBMS only acts for data delivery
  - weak client hardware





Goal

Python

```python
import pandas as pd
import psycopg2 as pg

con = pg.connect("dbname=sales user=data \
                 password=scientist host=bigserver")

df_o = pd.read_sql_table("orders", con)
df_c = pd.read_sql_table("customers", con)

df_j = df_o.join(df_c, how='inner',
               on=['o_custkey','c_custkey'])
df_j = df_j[df_j["c_name"] == "Max Power"]
```

SQL

```sql
SELECT *
  FROM orders o INNER JOIN customers c
    ON o_custkey = c_custkey
 WHERE c_name = 'Max Power'
```

Grizzly

- drop-in replacement for Pandas

```
import pandas as pd  ⟶  import grizzly as pd
```

- **Idea**: "hollow" DataFrame replacement
  - contains metadata, no real data
- solve Panda's scalability issues
  - lazy query evaluation – enable query optimization
  - In-DBMS query execution – exploit database capabilities
  - only transfer results to client – reduce transfer costs

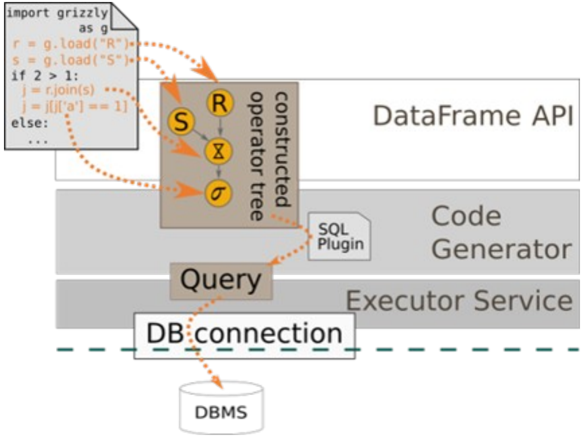
	Python Pandas	SQL
Projection	df['A'] df[['A','B']]	SELECT a FROM ... SELECT a,b FROM ...
Selection	df[df['A'] == x]	SELECT * FROM ...WHERE a = x
Join	pandas.merge(df1, df2, left_on='x', right_on='y', how='inner|outer|right|left')	SELECT * FROM df1 inner|outer|right|left join df ON df1.x = df2.y
Grouping	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() df['A'].value_counts()	SELECT min(a) FROM ... max(a)|avg(a)|count(a)|sum(a) SELECT a, count(a) FROM ... GROUP BY a
Add column	df['new'] = df['a'] + df['b']	SELECT a + b AS new FROM ...

Grizzly follows the **Query-Shipping Paradigm**,
bringing the Python query to the database.

TECHNISCHE UNIVERSITÄT ILMENAU

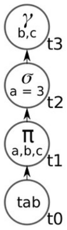
Grizzly architecture

- DataFrame operations are categorized as *transformations* and *actions*
- ***Transformations***
  - are collected (lazy evaluation)
  - e.g. Scans, Filters, Projections, Joins, …
- ***Actions***
  - trigger code generation for collected transformations
  - e.g. print, execute
- Query is sent to the DBMS, results are fetched and sent to the client




```
# 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
```

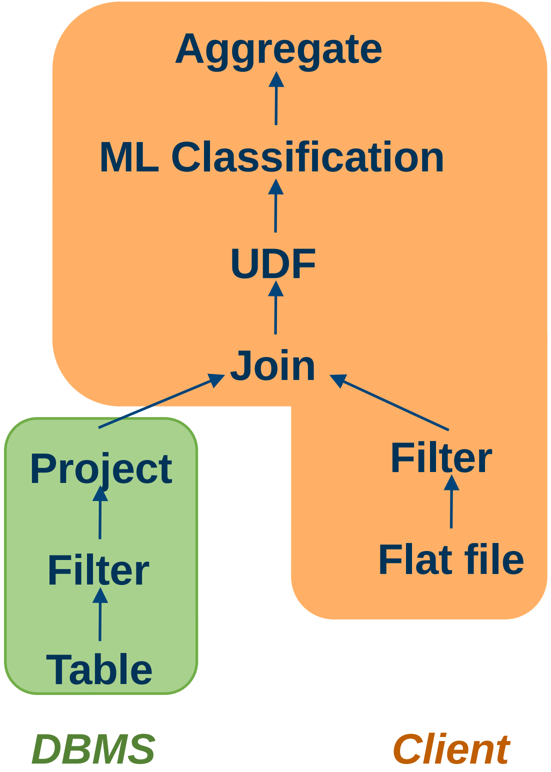
(c) Produced SQL query.

TECHNISCHE UNIVERSITÄT
ILMENAU

Grizzly: Extensions

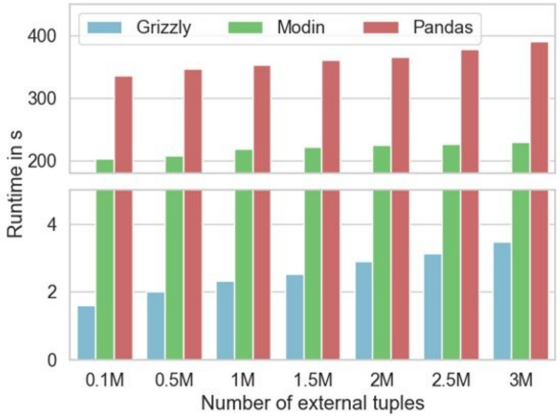
- Modern data analytics consist of more than simple queries
- Problem: every unsupported operation is a "Pipeline breaker"
  - intermediate result needs to be materialized on the client
  - local processing (Pandas, manually)
- subsequent operations can't be executed in the DBMS

  (although they could)



Heterogeneous Data Sources

- process external data inside the DBMS without loading
- special load function in Grizzly
- DBMS provided connectors
  - PostgreSQL: *foreign data wrappers*
  - Actian Vector: *external table*
  - MonetDB: `CREATE TABLE FROM LOADER`
- vendor-specific code: "pre-query" added before actual SQL query
- **WIP:** client-local processing using DuckDB, MonetDB/e, Arrow, etc.



Use Case Example: Join of table and external file & grouping

Here: Join customer data (table) with daily orders (flat file) and aggregate

Python:

```
df = grizzly.read_external_files("filename.csv",
  colDefs=["a:int, b:str, c:float"], hasHeader=True, delimiter='|')
```

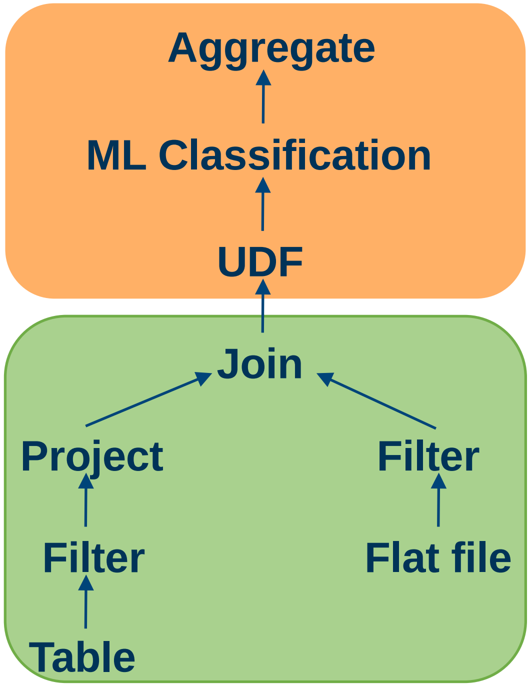
SQL:

```
DROP TABLE IF EXISTS temp_ext_tablet1;

CREATE EXTERNAL TABLE
 temp_ext_tablet1(a int, b VARCHAR(1024), c float)
 USING SPARK WITH REFERENCE='filename.csv',
 OPTIONS=('delimiter'='|');

SELECT * FROM temp_ext_tablet1 t1
```

Pre-queries

Query

TECHNISCHE UNIVERSITÄT ILMENAU


Aggregate
ML Classification
UDF
Join
Project
Filter
Filter
Flat file
Table

User-defined Functions

- create UDFs inside the DBMS using pre-queries
  - Some DBMS support Python
  - Compilation to
    - native code
    - PL/SQL (WIP)
- Python is weakly typed, SQL is strongly typed: type hints required

Python:

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

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

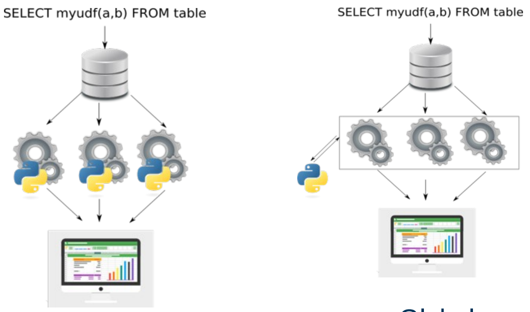
SQL:

```
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'
```

Pre-queries

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

Query



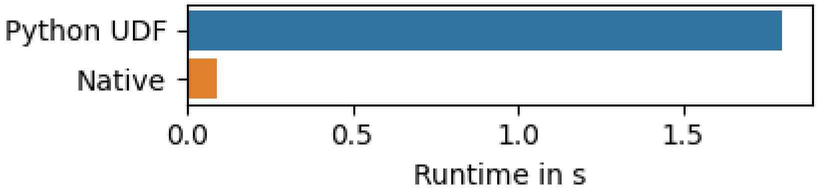
Worker-local interpreter

Global interpreter

- per Query: isolation, no caching
- long running: weak isolation, but allows caching

TECHNISCHE UNIVERSITÄT ILMENAU

UDFs: Optimization



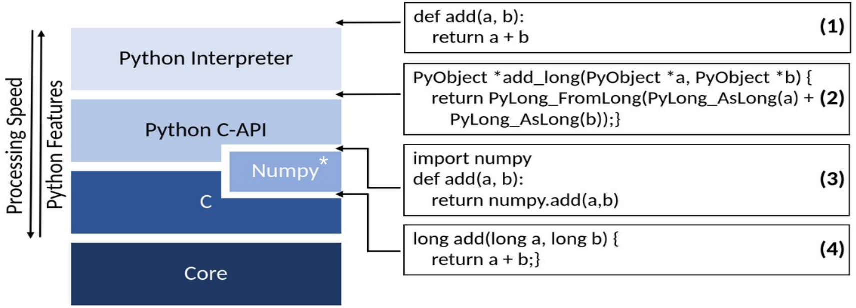
UDF vs. DBMS-native modulo operator on 8M tuples

Opportunities for Optimizations

- compile Python to native code
- vectorized execution
- parallel execution

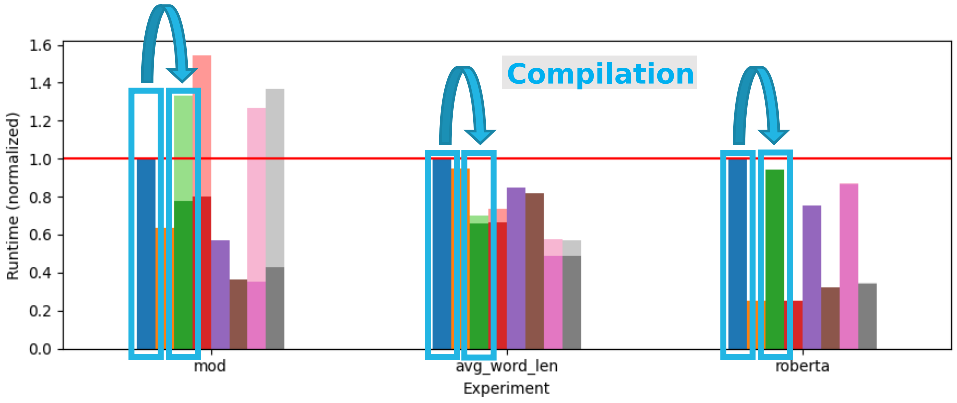
Compilation:

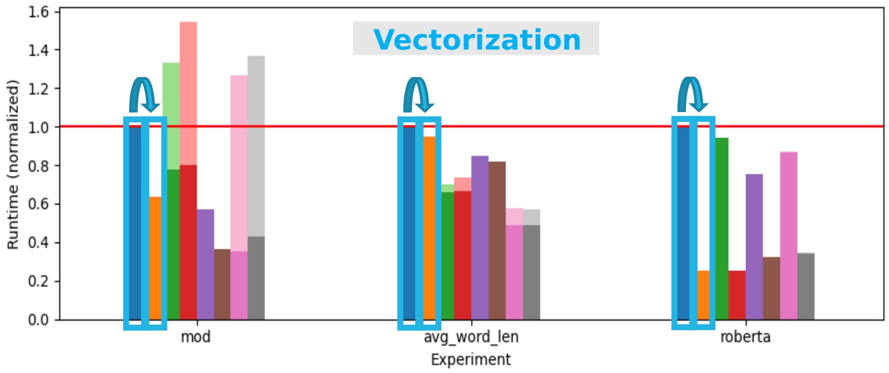
- dynamically integrate compiled code into running engine
- Challenges/requirements
  - transparent to user
  - external code/modules
  - types: SQL vs. Python
  - Safety: prevent insecure operations

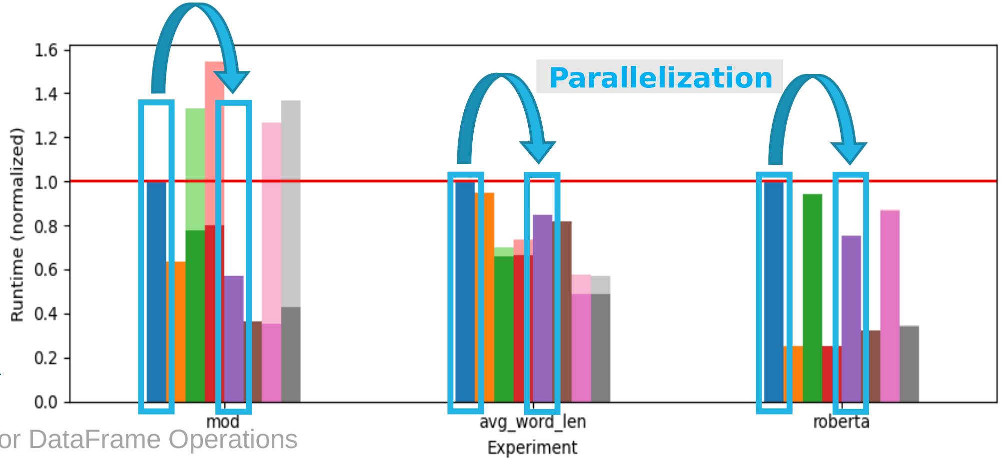


* M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. SSDBM, 2016.

UDFs: Optimization

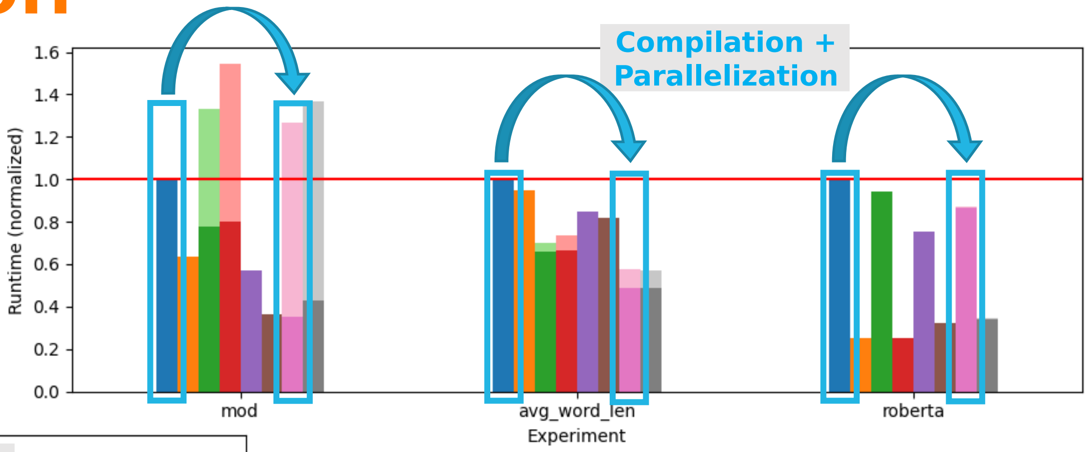


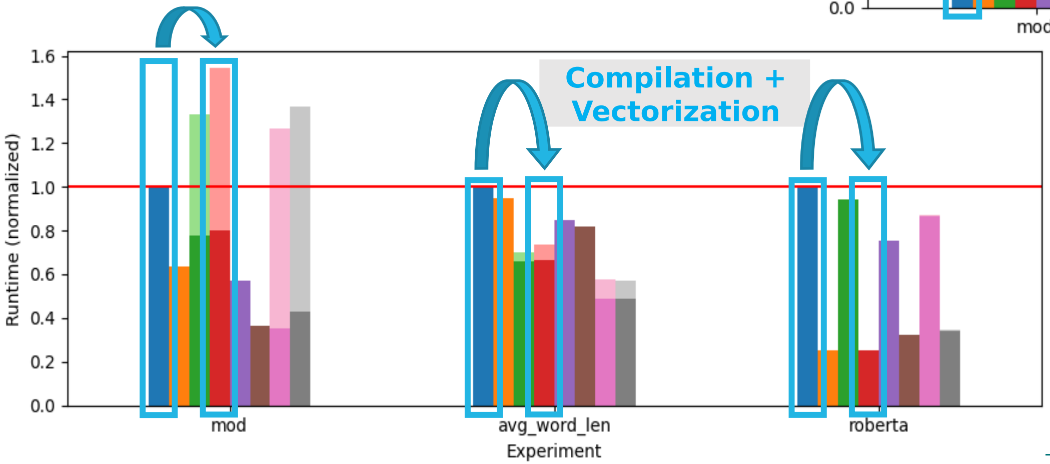


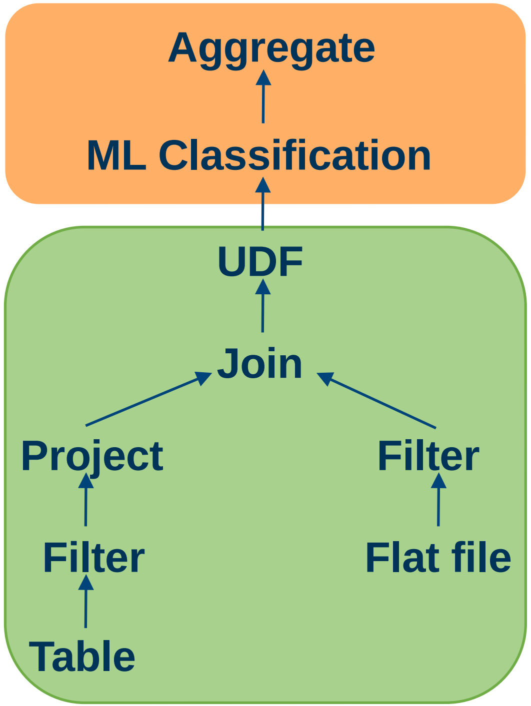


Actian Vector results (normalized)

UDFs: Optimization





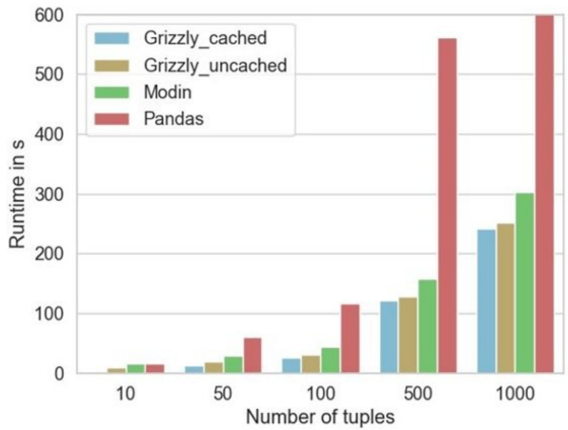


TECHNISCHE UNIVERSITÄT ILMENAU

ML Model Join

- Apply pre-trained ML models on data inside DBMS
  - Focus on (deep) neural networks
- Supported model types: Tensorflow, PyTorch, ONNX
- Generate code for model loading and application and ship as UDF
- Basic steps:
  1. Load and cache model
  2. Perform input conversions
  3. Run the model inference
  4. Apply output conversions

```
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()
```



Use Case: Apply existing ML model to table
Here: Sentiment analysis on movie review table & grouping on sentiment

ML Model Join Improvements

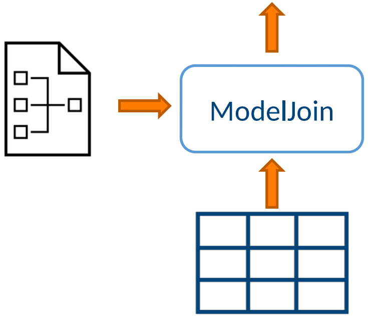
- UDFs are quite slow
- Compilation difficult with external libraries
- **Idea:** native DBMS support for ML representation (Neural Networks)
  - feed-forward / dense layers
  - RNNs

Currently Under Review

```
SELECT * FROM table t MODEL JOIN model m
```

Layer_in	Node_in	Layer	Node	Weight	Bias
…	…	…	…	…	…
1	0	2	0	0.3	0.1
1	1	2	0	0.2	**0.1**

Relational Representation: ML-to-SQL



Native ModelJoin operator


Aggregate
ML Classification
UDF
Join
Project
Filter
Filter
Flat file
Table
DBMS

Summary & Current Work

- Grizzly transpiles DataFrame operations to SQL to fully leverage DBMS capabilities
- User-friendly DataFrame API extensions to push advanced analytics to the DBMS:
    - Flat file joins
    - User-defined functions
    - Model join with pretrained ML models

- Feature completeness: use Pandas for Fallback
- Connection to multiple DBMS
    - local in-memory DBMS/Arrow
    - Optimizer rules
- Handling of operations that cannot be expressed in SQL (fallback to UDFs)
- Deeper integration of ML models into query execution
- Materialize computed columns / results

https://github.com/dbis-ilm/grizzly