Blue Elephants Inspecting Pandas

Inspection and Execution of Machine Learning Pipelines in SQL

Maximilian E. Schüle
Technical University of Munich
m.schuele@tum.de

Luca Scalerandi
Technical University of Munich
luca.scalerandi@tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

Data preprocessing, the step of transforming data into a suitable format for training a model, rarely happens within database systems but rather in external Python libraries and thus requires extraction from the database systems first. However, database systems are tuned for efficient data access and offer aggregate functions to calculate the distribution frequencies necessary to detect the under- or overrepresentation of a certain value within the data (bias).

We argue that database systems with SQL are capable of executing machine learning pipelines as well as discovering technical biases—introduced by data preprocessing—efficiently. Therefore, we present a set of SQL queries to cover data preprocessing and data inspection: During preprocessing, we annotate the tuples with an identifier to compute the distribution frequency of columns. To inspect distribution changes, we join the preprocessed dataset with the original one on the tuple identifier and use aggregate functions to count the number of occurrences per sensitive column. This allows us to detect operations which filter out tuples and thus introduce a technical bias even for columns preprocessing has removed. To automatically generate such queries, our implementation extends the mlinspect project to transpile existing data preprocessing pipelines written in Python to SQL queries, while maintaining detailed inspection results using views or common table expressions (CTEs). The evaluation proves that a modern beyond main-memory database system, i.e. Umbra, accelerates the runtime for preprocessing and inspection. Even PostgreSQL as a disk-based database system shows similar performance for inspection to Umbra when materialising views.

1 INTRODUCTION

A preprocessing pipeline is the connective link between the validated input data and the training of a model [12]. While a machine learning pipeline covers the entire life cycle of a model (see Figure 1), a data-preprocessing pipeline only refers to the part, which puts the available data into a form for training a machine learning model. Preprocessing consists of loading, joining, and filtering the data (step 1) and transforming existing columns or replacing missing values (step 2). Finally, the processed data is then fed into the model for training (step 3).

Data preprocessing alters the original data semantically, which can lead to the "over- or under-representation of a certain group" [8]. This is called a technical bias, when caused by data preprocessing, in contrast to a pre-existing bias already contained in the data. Technical biases that a static analysis of the pipeline cannot find are particularly critical. They occur after preprocessing has modified the data, for example, when a trained model frequently misclassifies some classes but works fine for others. Manually repeating and inspecting each step in a pipeline would be time-consuming for large datasets. Even if a technical bias is found, it has to be traced back to the bias-introducing operation to fix this *data bug*. Finding data bugs automatically would not only be interesting from a debugging perspective but also proactively ensure correctness and trustworthiness.

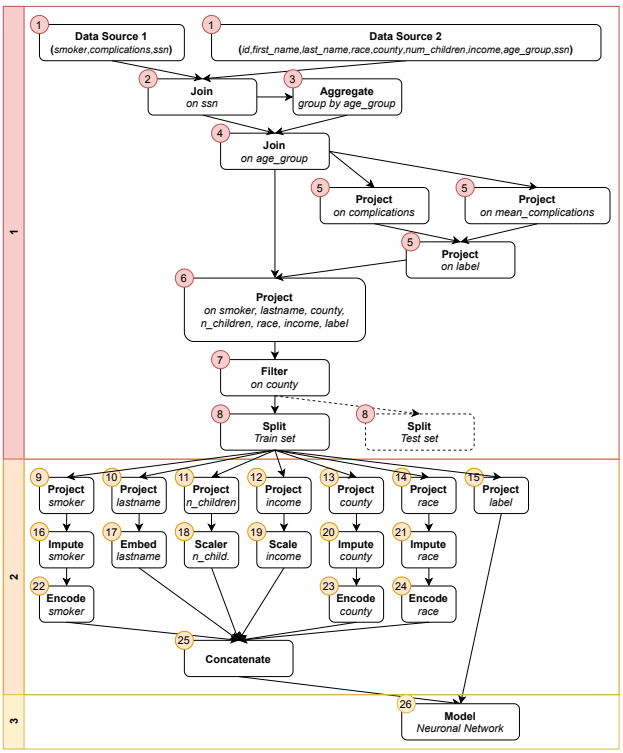

Figure 1: Dataflow of mlinspect's *healthcare* pipeline [8].

Programmatically inspecting intermediate results during execution reduces the effort to find technical biases. This includes counting the number of entries per group after each operation is performed to compute the ratio. The ratio is the relative distribution frequency of how often each sensitive group occurs within the data and should remain as constant as possible during preprocessing.

As the focus in machine learning lies on creating effective models and not on efficient data processing, the traditional wisdom is to use Python library functions not only for training a model

but also for data preprocessing. However, database systems are tuned for efficient data access and manipulation [25, 31, 32]. In a relational database, a table out of columns (attributes) and tuples, which each is a set of attribute values, represents a dataset. A synthetic tuple identifier *tid* identifies a tuple and can be used to trace tuples throughout all operations. Database systems offer a declarative language, SQL, allowing the user to specify *what* to do rather than caring about optimisation details [42]. SQL provides aggregate functions that allow grouping by a column to count the number of occurrences per value that forms a sensitive group. Common table expressions (CTEs) allow us to modularise and to structure the SQL code [39]. Modularising the code causes no additional overhead since the database system's query optimiser unnests nested subqueries before execution [21]. Moreover, modern database systems generate code for pipelined data processing [19, 38, 40, 43, 44]. This enables in-memory performance and eliminates the overhead of function calls and the need to materialise subresults. Thus, instead of delegating calls to library functions such as pandas, a database system will increase the overall performance of data preprocessing.

We argue that database systems with SQL are ideally suited for data inspection within machine learning pipelines. In our solution, one CTE/view represents one line of the original Python source code. We, therefore, present a set of SQL queries that covers the functionalities of certain Python libraries such as pandas and scikit-learn (see Figure 2). Furthermore, aggregate functions within CTEs/views will track the ratio for each group of interest to detect biases: We equip each tuple with a synthetic index (tid) to identify a tuple although the sensitive column is not part of the result. After each operation that influences the number of occurrences per tuple, e.g. joins or selections, the index allows the restoration of the sensitive column if needed to count the number of occurrences per group. This is the first study that allows inspection in SQL of columns not even present in the result by preserving the tid. To automatically generate these SQL queries, we extend the mlinspect framework [7, 8] with an SQL backend. This way we can rely on the user interface for bias detection but just generate the SQL queries in the background. Likewise, we are able to fall back on the original Python library function calls when no SQL substitute has been implemented so far. Generating SQL code from Python pipelines provides further advantages. Firstly, existing pipelines can benefit from the performance of database systems without alteration of the original code. Secondly, it allows a less verbose specification of pipelines, which is also accessible for those not too familiar with SQL. This study's specific contributions are:

- inspection of preprocessing pipelines in SQL,
- a set of SQL expressions that cover the functionalities of the Python libraries pandas and scikit-learn,
- an SQL backend for the mlinspect framework to translate Python into SQL supporting many pandas and scikit-learn operations with a focus on inspection,
- an expandable transpilation framework[1] to facilitate the addition of support for other libraries,
- executing the mlinspect inspection of pipelines while offloading all expensive computations to a database system,
- functionality to generate inspection-enabled SQL queries from pipelines written in Python without execution,
- full support for running and inspecting end-to-end pipelines including training and testing the models in Python.

[1]https://github.com/tum-db/mlinspect4sql

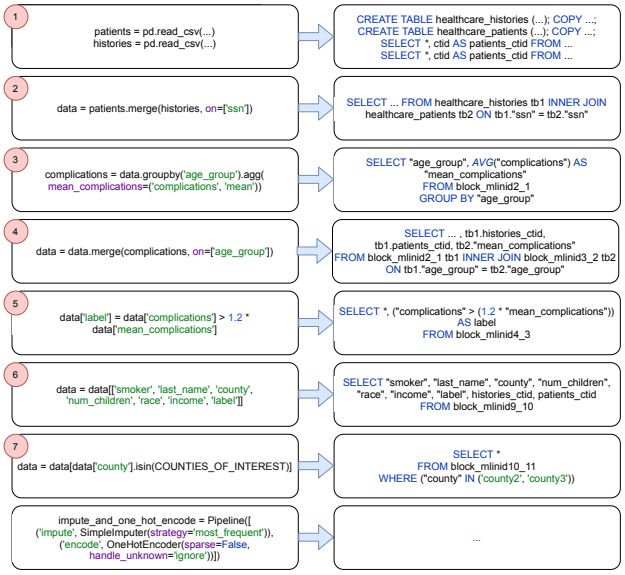

Figure 2: Functional blocks with references to Figure 1.

- and an evaluation that compares the performance gain when inspecting preprocessing pipelines within a disk-based database system, i.e., PostgreSQL [37] (*blue elephant*), and a beyond main-memory one, i.e., Umbra [16, 20, 35, 36], to Python library function calls.

This study comprises the following sections: Section 2 summarises the related work on debugging pipelines and SQL support for machine learning. Section 3 introduces the SQL expressions needed for tuple tracking and the detection of introduced biases. To generate these queries, Section 4 shows how to create an SQL backend for the mlinspect framework using monkey patching. This SQL backend further generates one CTE/view in SQL per line of the original Python code (Section 5). The evaluation in Section 6 presents the performance of selected pipelines within PostgreSQL and Umbra. Section 7 concludes this study with an outlook on further extensions.

2 RELATED WORK

This study combines research on integrating machine learning pipelines into database systems [1, 2, 5, 14, 27, 30, 47, 48, 52] with studies on debugging pipelines. Debugging machine learning pipelines is necessary to ensure fairness [23, 24, 26, 49, 50], when data transformations [29] introduce biases [13] or, more generally, to detect the cause of incorrect predictions. For example, *SliceLine* [28] proposes algorithms to investigate the effectiveness of training datasets. *Breadcrumb* [6] finds removed tuples in Spark. *Vizer* [3] is a debugging tool with SQL and Python integration that also uses Spark as backend. *MLDebugger* [17] and *BugDoc* [18] are data debuggers for *VisTrails* [4].

mlinspect [8] provides an extendable framework that, on the basis of declarative abstractions of popular data science libraries, allows insightful inspections without the need for manual code instrumentation. For these inspections, mlinspect mostly relies on the frameworks NumPy and pandas with mostly modest performance. mlinspect's functionality for pipeline inspection does not require the alteration of any existing code. This work proposes SQL queries to perform the inspections recommended by mlinspect. The mlinspect project will form the basis for implementing

the proposed transpiling and offloading of data-intensive operations to a database system, including all inspection capabilities. *Grizzly* [10, 11] aims to replace the pandas library by transpiling Python code using pandas to SQL and offloading the operations to a database system. It is capable of dealing with external data, user-defined functions and reusing intermediate results. Grizzly can emit the generated SQL code but requires a connection to a database system to work in the first place, so we decided to generate the SQL queries independently. Furthermore, this work's SQL code-generation covers data inspection and other library functions and stores each result in a view/CTE.

3 TUPLE-TRACKING AND INSPECTION

One contribution of this paper is the inspection of so-called data distribution bugs [8], which are based on pre-existing and introduced (technical) biases, when using SQL. A bias is as a systematic error leading to irrational preferences or aversions [51]. There is no general recipe to avoid introducing a technical bias in data preprocessing prior to training a model but detection is possible. For bias detection, it is necessary to monitor the ratios per sensitive column, which is the distribution frequency of a value, and the introduced ratio changes after each operation.

Biases can also be induced for columns not present in the original data or such that preprocessing has removed. Tracking the tuples explicitly detects biases even when preprocessing has removed a sensitive column. To track tuples throughout the pipeline, it is necessary to cope with possible projections when the sensitive column is not part of the result. The proposed tuple tracking allows us to restore and monitor all columns present in the original data independently of their removal inside the pipeline. An example for a common projection is one-hot-encoding, where an input is turned into a binary array. Assigning a unique identifier to each entry allows ratio changes the pipeline has introduced independently of the tuple's current representation to be discovered.

The mlinspect framework provides two checks: *NoIllegalFeatures* verifies that none of the used features in the provided dataset are contained in a blacklist of illegal feature names. *NoBiasIntroducedFor* targets pre-existing and technical biases.

The latter check is based on three inspections: *MaterializeFirstOutputRows* materialises a set of rows from the input by each operator, to easily examine the effects of the pipeline [8]. *RowLineage* provides lineage information for the resulting tuples. *HistogramForColumns* calculates the ratios for specified columns before and after each operation in the pipeline.

3.1 Tuple-Tracking and Inspection in SQL

To implement tuple tracking, an identifier for the tuples is needed. Most database systems allow access to the physical table location, which could be used as an identifier. For PostgreSQL, the `ctid` is such an identifier, although it is not suitable as a long-term identifier[2]. This is based on the possibility of garbage collection routines changing the physical location. As we extract the `ctid`, initially only once within a common table expression (Listing 1 lines 3-6), we ensure that we use consistent identifiers during the checks.

To reproduce mlinspect's bias inspections in SQL, we need to compute and compare the ratios for each sensitive column (*HistogramForColumns*). In order to compare the ratios (Listing 1 lines 11-19), it is necessary to group by each sensitive column to count the initial and current number of occurrences per value. One common table expression retrieves the original number (Listing 1 lines 7f). When measuring the current number, there are two cases. In the simple case, the result table already contains the sensitive column. In this simple case, an aggregate function counts the number of occurrences per column directly (Listing 1 lines 9f). In the event that the current table contains the tuple identifier of the original table but no longer contains the sensitive column, a join on the tuple identifier restores the requested column (Listing 2 lines 7-9). When an aggregation function is part of the preprocessing pipeline, also the tuple identifiers must also be aggregated to form an array (Listing 3 line 5). Unnesting the array allows the number of included tuples (line 8) to be computed and the sensitive column to be restored if required.

[2]https://www.postgresql.org/docs/12/ddl-system-columns.html

```
1  CREATE TABLE data (a int, s int); -- sensitive column: "s"
2  INSERT INTO data (values (1,1),(1,2));
3  WITH orig AS ( -- the original data with exposed ctid
4    SELECT ctid, a, s FROM data),
5  curr AS ( -- current representation after preprocessing
6    SELECT ctid, s FROM orig WHERE s > 1),
7  orig_count AS ( -- original count per value of column "s"
8    SELECT s, count(*) AS cnt FROM orig GROUP BY s),
9  curr_count AS ( -- current count per value of column "s"
10   SELECT s, count(*) AS cnt FROM curr GROUP BY s),
11 orig_ratio AS ( -- original ratio per value of column "s"
12   SELECT s, (cnt*1.0 / (select count(*) FROM orig)) AS ratio
13   FROM orig_count),
14 curr_ratio AS ( -- current ratio per value of column "s"
15   SELECT s, (cnt*1.0/(select sum(cnt) FROM curr_count)) AS ratio
16   FROM curr_count)
17 -- join on the sensitive column to calculate the ratio change
18 SELECT o.s, o.ratio - COALESCE(c.ratio,0) AS bias_change
19 FROM curr_ratio c RIGHT OUTER JOIN orig_ratio o ON o.s = c.s;
```

Listing 1: Ratio measurement (column present).

```
1  CREATE TABLE data (a int, s int); -- sensitive column: "s"
2  INSERT INTO data (values (1,1),(1,2));
3  WITH orig AS (...),
4  curr AS ( -- current representation after preprocessing without column "s"
5    SELECT ctid, a FROM orig WHERE s > 1),
6  orig_count AS (...),
7  curr_count AS ( -- current count per value of column "s" needs join
8    SELECT s, count(*) AS cnt FROM curr c, orig o
9    WHERE c.ctid=o.ctid GROUP BY s),
10 orig_ratio AS (...),
11 curr_ratio AS (...)
12 ...
```

Listing 2: Ratio measurement (column not present).

```
1  CREATE TABLE data (a int, s int); -- sensitive column: "s"
2  INSERT INTO data (values (1,1),(1,2));
3  WITH orig AS (...),
4  curr AS ( -- current representation (aggregated)
5    SELECT array_agg(ctid) AS ctid, s FROM orig GROUP BY s),
6  orig_count AS (...),
7  curr_count AS ( -- current count for column "s" needs unnest
8    SELECT s, count(*) AS cnt FROM (SELECT unnest(ctid) AS ctid, s
           FROM curr) c, orig o GROUP BY s),
9  orig_ratio AS (...),
10 curr_ratio AS (...)
11 ...
```

Listing 3: Ratio measurement of an aggregated value.

3.2 Technical Bias Detection

Not all operations can introduce a bias, as not all operations add or remove tuples from the dataset. This section shows how a technical bias can be introduced but also how it can be detected. We want to inspect the columns `race` and `age_group` for an introduced bias. Lines 20 to 35 (Listing 4) contain operations that introduce a bias, for example, line 35 (see Figure 3). The projection in line 33 has removed the column `age_group`, so it is

```
20 patients = pd.read_csv(os.path.join( str(get_project_root()),
21     "example_pipelines", "healthcare", "patients.csv"),
22     na_values='?')
23 histories = pd.read_csv(os.path.join( str(get_project_root()),
24     "example_pipelines", "healthcare", "histories.csv"),
25     na_values='?')
26
27 data = patients.merge(histories, on=['ssn'])
28 complications = data.groupby('age_group').agg(
29     mean_complications=('complications', 'mean'))
30 data = data.merge(complications, on=['age_group'])
31 data['label'] = \
32     data['complications'] > 1.2 * data['mean_complications']
33 data = data[['smoker', 'last_name', 'county',
34     'num_children', 'race', 'income', 'label'] ]
35 data = data[data['county'].isin(COUNTIES_OF_INTEREST)]
```

Listing 4: *healthcare* pipeline [8] (lines 20-35).

...	county	race	age_group	Track_id_Patients	Track_id_Histories
	county_1	race_3	age_group_2	1	11
	county_2	race_2	age_group_2	2	22
	county_3	race_1	age_group_1	3	33
	county_3	race_2	age_group_1	4	44
	county_2	race_2	age_group_2	5	55
	county_1	race_1	age_group_1	6	66

data = data[data['county'].isin(['county_1', 'county_2'])]

...	county	race	age_group	Track_id_Patients	Track_id_Histories
	county_1	race_3	age_group_2	1	11
	county_2	race_2	age_group_2	2	22
	county_2	race_2	age_group_2	5	55
	county_1	race_1	age_group_1	6	66

Figure 3: Example of bias-introducing operation.

Ratio race	Ratio age_group
race_1: 0.333	age_group_1: 0.5
race_2: 0.5	age_group_2: 0.5
race_3: 0.166	

Ratio race	Ratio age_group
race_1: 0.25 (-0.083)	age_group_1: 0.25 (-0.25)
race_2: 0.5 (-0.0)	age_group_2: 0.75 (+0.25)
race_3: 0.25 (+0.084)	

Figure 4: Ratio change by the operation in line 35 for the exemplary data.

```
1  CREATE TABLE patients_51_mlinid0 ( "id" INT, "first_name" text,
       "last_name" text, "race" text, "county" text, "num_children
       " INT, "income" FLOAT, "age_group" text, "ssn" text);
2  COPY patients_51_mlinid0("id", "first_name", "last_name", "race"
       , "county", "num_children", "income", "age_group", "ssn")
       FROM 'patients.csv' WITH (DELIMITER ',', NULL '', FORMAT
       CSV, HEADER TRUE);
3  CREATE TABLE histories_52_mlinid1 ( "smoker" text, "
       complications" INT, "ssn" text);
4  COPY histories_52_mlinid1("smoker", "complications", "ssn") FROM
        'histories.csv' WITH (DELIMITER ',', NULL '', FORMAT CSV,
       HEADER TRUE);
5
6  WITH patients_51_mlinid0_ctid AS (
7   SELECT *, ctid AS patients_51_mlinid0_ctid
8   FROM patients_51_mlinid0
9  ), histories_52_mlinid1_ctid AS (
10  SELECT *, ctid AS histories_52_mlinid1_ctid
11  FROM histories_52_mlinid1
12 ), block_mlinid2_53 AS (
13  SELECT tb1."id", tb1."first_name", tb1."last_name", tb1."race",
        tb1."county", tb1."num_children", tb1."income", tb1."
       age_group", tb1."ssn", tb1.patients_51_mlinid0_ctid, tb2."
       smoker", tb2."complications", tb2.
       histories_52_mlinid1_ctid
14  FROM patients_51_mlinid0_ctid tb1 INNER JOIN
       histories_52_mlinid1_ctid tb2 ON tb1."ssn" = tb2."ssn"
15 ), block_mlinid3_54 AS (
16  SELECT array_agg(tb1.patients_51_mlinid0_ctid) as
       patients_51_mlinid0_ctid, "age_group", AVG("complications"
       ) AS "mean_complications"
17  FROM block_mlinid2_53 GROUP BY "age_group"
18 ), block_mlinid4_55 AS (
19  SELECT tb1."id", tb1."first_name", tb1."last_name", tb1."race",
        tb1."county", tb1."num_children", tb1."income", tb1."
       age_group", tb1."ssn", tb1."smoker", tb1."complications",
       tb1.histories_52_mlinid1_ctid, tb1.
       patients_51_mlinid0_ctid, tb2."mean_complications"
20  FROM block_mlinid2_53 tb1 INNER JOIN block_mlinid3_54 tb2 ON
       tb1."age_group" = tb2."age_group"
21 ), block_mlinid9_62 AS (
22  SELECT *, "complications" > 1.2 * "mean_complications" AS label
23  FROM block_mlinid4_55
24 ), block_mlinid10_63 AS (
25  SELECT "smoker", "last_name", "county", "num_children", "race",
        "income", "label", histories_52_mlinid1_ctid,
       patients_51_mlinid0_ctid
26  FROM block_mlinid9_62
27 ), block_mlinid13_66 AS (
28  SELECT * FROM block_mlinid10_63
29  WHERE "county" IN ('county2', 'county3')
30 )
31 SELECT tb_orig."age_group", count(*)
32 FROM block_mlinid13_66 tb_curr JOIN patients_51_mlinid0_ctid
       tb_orig ON tb_curr.patients_51_mlinid0_ctid=tb_orig.
       patients_51_mlinid0_ctid
33 GROUP BY tb_orig."age_group";
```

Listing 5: Inspection of column `age_group` in SQL.

not explicitly present in the table and shown in grey. Apart from line 35, the joins in line 27 or 30 could introduce a bias as well depending on a join partner. If many people of one `race` live in a removed county, a racial bias will be introduced. Far less visible is that line 35 could also introduce a bias regarding the `age_group`. This seems contradictory, as the selection in line 33 has removed the column `age_group`. But biases can be introduced for columns not present in a dataset's current representation.

Before calculating the ratio changes for each group, one has to define the threshold for which a bias becomes unacceptable, for example 25%. If the representation of a feature in the data alters by more than or equal to 25%, it is assumed that the pipeline introduced a bias. Figure 4 presents the original ratio on the left and the new ratio on the right. The parentheses in the right table show the precise changes. The ratios can be calculated as described in Section 3.1 (Listing 1, 2). The ratios of the feature `race` did not surpass the threshold of 25% and are assumed acceptable. For the feature `age_group`, the changes for both classes changed by exactly 25%. In this case, `age_group_2` is over- and `age_group_1` is underrepresented, which could lead to a model discriminating against some groups if trained with this data.

3.3 Automated Inspection in SQL

To inspect the biases each time, mlinspect measures the column ratios, whereas our extension delegates the call instead to the database system, returns the measured results and injects them into the corresponding data class (`SQLHistogramForColumns`). At this point, the default functionality of mlinspect performs the comparison of the measured ratios to the user-given threshold. When a pre-defined threshold is exceeded, a bias has been introduced and the user is notified.

Listing 5 displays the translation of Listing 4 to SQL with inspection of the column `age_group`. The first lines (lines 1-4) import the tables from a CSV file. The first two CTEs expose the `ctid` as a tuple identifier with a unique name (lines 6-11). Afterwards, one CTE represents one line of the original code, so either joins (lines 12-14, 18-20), an aggregation (lines 15-17), projections (lines 21-23, 24-26) or a condition (lines 27-29). Finally, lines 31-33 show the code to inspect the column `age_group`: First, the preserved `ctid` allows the reconstruction of this column as it was not part of the last projection. Then an aggregate function counts the number of occurrences per value, which is returned to the mlinspect framework.

```
1 inspector_result = PipelineInspector \
2      .on_pipeline_from_file("Listing3.py") \
3      .add_custom_monkey_patching_module(custom_monkeypatching) \
4      .add_check(NoBiasIntroducedFor(["age_group", "race"]))
5 inspector_result = inspector_result.execute_in_sql(
6       dbms_connector=PostgresqlConnector(dbname="db",user="user"
            ,password="123",port=5432,host="localhost"),
7       mode="VIEW", materialize=True)
```

Listing 6: Addressing the SQL target with inspection.

```
1 code = cleandoc("""
2 import pandas as pd
3 patients = pd.read_csv(r"path/to/p.csv", na_values="?")
4 histories = pd.read_csv(r"path/to/h.csv", na_values="?")
5 data = patients.merge(histories, on=['ssn'])
6 data["new_col"] = patients["children"] > 2
7 """)
8 inspector_result = PipelineInspector \
9 .on_pipeline_from_string(code) \
10 .add_check(NoBiasIntroducedFor(["race"])) \
11 .execute()
```

Listing 7: Simple pipeline with inspection call.

3.4 User Interface for the SQL Target

Code-generation to SQL is realised as an extension to the mlinspect [8] project. The users alone are able to choose the right fit for the task at hand, so no functionality is forced on them. The changes do not duplicate existing functionality or require any later development on mlinspect to be altered. The goal is to translate a productive pipeline written in Python to SQL statements without requiring the user to rewrite any of them, as is required for existing tools such as Grizzly or *pandas-to-sql*[3]. Listing 6 shows the code to inspect the attributes `race` and `age_group` from an input pipeline stored as `Listing3.py`. The first part (lines 1-4) originates from the mlinspect framework and prepares the inspection steps. The second part (lines 5-7) executes the pipeline including inspection within a database system.

3.4.1 Choice: View/CTE. The users are able to choose if the whole pipeline is held in a CTE or views should be used (line 7: `mode="CTE"/"VIEW"`). In PostgreSQL, CTEs represent an optimisation barrier, which reduces the performance[4]. When choosing CTE as the target, the previous SQL operations must be included within the `with` clause for each query, for example, once for each inspected attribute. Whereas when creating a view for each query, less SQL code has to be generated. Nevertheless, each non-materialised view runs each query on demand.

3.4.2 Materialising a View/CTE. To avoid rerunning every query on demand, the user is also able to decide if intermediate views are materialised (line 7: `materialize=True/False`). Materialisation is interesting for views/CTEs that are expensive to calculate, contain few tuples and are used frequently. For example, if a query accesses a view at least twice, the accessed view is a candidate for materialisation. This applies to all fitting parameters that are calculated for the application of scikit-learn preprocessing functions (see Section 5.2). Materialising the number of occurrences per group also accelerates the runtime for inspections when comparing the ratios before and after some operations. Materialising a view or a CTE before the first operation is generally faster, as subsequent operations depend on its result. When the user chooses to materialise, all created views/CTEs, for which recalculating can be avoided, as well as all fitting parameters are materialised. On the other hand, when the database system provides main-memory capabilities, storing big intermediate tables might not be wanted as rerunning the query with the additional step performs as well as materialisation.

The decision to materialise a view/CTE is made after translating the Python code. Our implementation allows choosing between views and CTEs and whether to materialise them. The class `SQLQueryContainer` holds all SQL queries in a list. This also allows representing an SQL query as a CTE or as a view and to materialise it when needed.

[3]https://github.com/AmirPupko/pandas-to-sql
[4]https://www.postgresql.org/docs/12/queries-with.html

```
1  from mlinspect.instrumentation._pipeline_executor import \
2     set_code_reference_call, \
3     set_code_reference_subscript, \
4     monkey_patch, undo_monkey_patch
5  monkey_patch()
6  import pandas as pd
7  patients = pd.read_csv(
8     'path/to/p.csv',
9     **set_code_reference_call(
10       3, 11, 3, 178, na_values='?'))
11 histories = pd.read_csv(
12    'path/to/h.csv',
13    **set_code_reference_call(
14       4, 12, 4, 180, na_values='?'))
15 data = patients.merge(histories,
16    **set_code_reference_call(4, 7, 4, 44, on=['ssn']))
17 data[
18    set_code_reference_subscript(5, 0, 5, 42, ('new_col'))
19 ] =
20    patients[
21       set_code_reference_subscript(
22          5, 18, 5, 38, ('children')
23       )
24    ] > 2
25 undo_monkey_patch()
```

Listing 8: Listing 7 prepared for monkey-patching.

4 TRANSPILING CODE

The mlinspect project is an extendable tool for lightweight inspection of data-preprocessing pipelines written in Python [8]. Support for additional Python libraries for data preprocessing can be added through backends and monkey-patching modules. Monkey-patching is a concept based on Python's ability to extend its functionality of objects and functions at runtime [15]. Instead of modifying the user code, monkey-patching catches relevant function calls by their type and delegates the call to an alternative modified function. This approach considers functions only that are needed to support the inspection functionality. Additionally, nested calls are automatically executed in Python's default execution order. Currently, only data pipelines created with pandas[5] and scikit-learn [22] can be inspected. Scikit-learn and pandas are popular and constantly evolving libraries, so support for new functionality needs to be added over time.

Running mlinspect returns a dataflow directed acyclic graph (DAG) representing the pipeline and two dictionaries. One dictionary maps each check to its result and a second dictionary maps each DAG node to the inspection results associated with it. The dictionaries can be examined to see the checks that failed and the ones that passed.

The class `PipelineInspector` (Listing 7) initialises the setup in order to annotate the original source (`code`) for monkey-patching (Listing 8). Before execution, monkey-patching either adds the checks and inspections or replaces the whole function call as needed for an SQL backend. To deduce the table schema from the original Python code, the SQL backend runs the pipeline with a subset of the original data (e.g. ten rows) first. The main piece

[5]https://pandas.pydata.org/docs/index.html

of the transpilation is the *SQL mapping* that maps each object the original Python source code creates, called *dummy object*, to the deduced information to create the corresponding representation in SQL (*table expression*). The SQL mapping is a hash-table with the Python dummy object as key and the deduced information as the value. The dummy object is either of the `pandas.Series` or `DataFrame` type. This is better suited as a key than a variable name as multiple dummy objects share the same name in the Python source code. To ensure that each name of a view or a CTE is unique, we assign a sequential number to its name in SQL. As each line of the source code might depend on previous operations, each table expression might also depend on previous ones. Each patched function is performed once on the Python object to deduce the table expression in SQL. In this way, the next operation working on the output knows which table expression to access. Before execution, the `SQLQueryContainer` generates all table expressions and the class `SQLLogic` embeds them in a view/CTE. The SQL mapping contains, in addition, the following information to allow combining queries and tracking tuples:

(1) The Python dummy object.
(2) List of tuple identifiers (index columns) currently associated with the Python dummy object.
(3) List of non-index columns.
(4) *Execution tree* of operations that lead to the current dummy object (used to resolve joins on the same table).

The list of identifiers contains the tuple identifiers that are needed to reconstruct a sensitive column during inspection. Each identifier has a unique name (<*view/CTE-name*> + `_ctid`), which is propagated throughout all operations. In our implementation, a dictionary in `SQLHistogramForColumns` maps the original column name in pandas to the table and the tuple identifier in SQL. When the mlinspect framework needs to inspect a sensitive column, it delegates the call to our SQL backend. The backend looks up the identifier in the dictionary to generate and execute the table expression that calculates the ratios and restores the column when needed.

Each patched function returns exactly what the original would return. This guarantees that the inspection does not distort any results. The modifications are entirely generic and, hence, are independent of the already presented implementation of the mlinspect's backend and monkey-patching modules.

The query that is subsequently built is always in an executable state, because by monkey-patching it is not known where the pipeline ends. The ability to always return an executable and correct query is also suitable for debugging and testing parts of the pipeline. A class `SQLQueryContainer` collects all the operations in a list that can be translated into working queries for any statements in the pipeline at any time.

5 TRANSLATION INTO SQL

In order to translate each call of a library function, this section presents the supported set of SQL queries to run selected preprocessing pipelines.

5.1 Supported Pandas Functions

5.1.1 Read from CSV. To load data from a CSV file to SQL, the table first needs to be created. This is done by analysing the CSV file and deriving the data types based on the content. Besides integers and floats, other types—*objects* in pandas—are considered as a string. Afterwards, the data is loaded using the `COPY` command in SQL. For tuple-tracking, a column with an identifier has to be added. In PostgreSQL, this is achieved by exposing the ctid.

Some pandas operations rely on a row number, for example, row-wise multiplications of two columns of different tables (see Section 5.1.8). For these operations, it is necessary to add the column to the original data structure. When working on a CSV file, for example, a new column could be added to the file before reading and storing the content in the database system. Actually, in two of the three example datasets in mlinspect, such columns are already present, so detection capability for such cases was added as well. The new table with the added columns will be used as the source in the SQL mapping.

5.1.2 Merge/Join. The Python counterpart to SQL joins in the pandas library on `pandas.DataFrames` is called `merge` (Listing 4 line 27) and consists of the join partner (`right`), the join type (`how`), e.g. `inner` (default) or `cross`, and the column containing the join condition (`on`).

The translation consists of finding the table expression in the SQL mapping and putting the tables together in a `JOIN ... ON` block (Listing 5 line 14). The select clause explicitly lists the columns to avoid duplicates within the same table. A difference exists regarding the handling of `null` values as pandas treats `null` as a value to join. If the attribute is nullable and `null` values are not pruned, we can mimic this behaviour by adding to the join condition whether both values are `null`: `tb1.<c>=tb2.<c> or (tb1.<c> is null and tb2.<c> is null)`.

The tuple identifiers from the two input tables are all added to the result table. This is needed as a join influences the occurrence of a tuple depending on a join partner, which affects the ratio and might introduce a bias.

5.1.3 Selection and Projection. The selection and projection are subsumed together, as in pandas they are both accessed through the same function: `__getitem__`. Depending on the arguments, this function performs either operation.

If the argument is a string or a list of strings, a projection is meant where the strings name the columns.

With selections, the argument will frequently be a nested operation representing the selection condition. The only restriction is that the input evaluates to a `pandas.Series` of type bool, the criterion for keeping or removing rows. The `pandas.Series` is a single column of the `pandas.DataFrame`.

As the operations are translated line by line, it is not known beforehand whether an operation is part of the result as a column or used within a selection condition. Internally, the SQL mapping maintains an execution tree. Each caught functions extends the execution tree. After processing all operations, the complete execution tree gets traversed to create the current table expression.

5.1.4 Arithmetic/Boolean Operations (e.g., +, ∗, >, &, `not`*).* Applying monkey-patching to Listing 9 results in multiple single calls and cannot be translated into a single SQL query:

(1) The projection `data['b']` returns a new pandas object, called `pandas.Series(data_b)`
(2) The projection `data['c']` again leads to a new pandas object: `pandas.Series(data_c)`.
(3) Also the multiplication `__rmul__` (∗) with the arguments `1.2` and `pandas.Series(data_c)` returns the new object `pandas.Series(data_c_mul)`
(4) The comparison `__gt__` (>) operation with the arguments `pandas.Series(data_b)`, `pandas.Series(data_c_mul)` returns `pandas.Series(data_b_gt_data_c_mul)`

(5) Finally, the result is the assignment `__setitem__` with the arguments `data` (origin), `'a'` (new label), and as source `pandas.Series(data_b_gt_data_c)`.

Monkey-patching only delegates explicit function calls with its parameters and does not provide any additional information about the context of the function call. As a consequence, it is not known when a caught operation is part of a bigger nested call or if it stands alone. Of course, parsing the code or injecting additional parameters into the call allows us to deduce the context, but the following aspects avoid the need for additional modifications: Monkey-patching obeys the default execution order from Python. The optimiser of modern database systems eliminates subqueries that are not required to calculate the result.

One possible approach is to translate each single function call into one CTE/view, which uses the former one as input. This requires a join on the index column for operations that add a single column or depend on a modified one (Listing 10).

Our implementation reduces dependencies as it resolves joins and includes the columns of former operations when the operations affect the same table. An SQL-mapping execution tree contains all operations that lead to the current result. This way, each time an operation on the same table is resolved, the previous SQL statement is copied and the new operation is added as a column instead of using views/CTEs as input (Listing 11). If some views/CTEs are not part of the result and therefore remain unused, the database system will not execute them as optimised out. So it is not necessary to consider whether a pandas projection is part of a result table.

```
data['a'] = data['b'] > 1.2 * data['c']
```

Listing 9: Binary operation with additional assignment.

```
WITH data_b AS (
    SELECT b, ctid FROM data
), data_c AS (
    SELECT c, ctid FROM data
), data_c_mul AS (
    SELECT (c * 1.2) AS c_mul, ctid FROM data_c
), data_b_ge_data_c AS (
    SELECT (l.b > r.c_mul) AS b_ge_c_mul, l.ctid
    FROM data_b l, data_c_mul r
    WHERE l.ctid = r.ctid
), result AS (
    SELECT l.data_f, l.b, l.c, r.b_ge_c_mul AS "a", l.ctid
    FROM  data l, data_b_ge_data_c r
    WHERE l.ctid = r.ctid
)
```

Listing 10: Generated query with index columns.

```
WITH data_b AS (
    SELECT "b" FROM data
), data_c AS (
    SELECT "c" FROM data
), data_c_mul AS (
    SELECT (1.2 * "c") FROM data
), data_b_ge_data_c_mul AS (
    SELECT (("b" > (1.2 * "c"))) FROM data
), result AS (
    SELECT *, ("b" > (1.2 * "c")) AS "a" FROM data
)
```

Listing 11: Generated query (condensed).

5.1.5 Group-By and Aggregation. A group-by with an additional aggregation consists of two parts in pandas (Listing 4 lines 28f). First, the group-by function is called on the pandas table, with the columns to group by as the argument. This function returns a specific group-by object, which is only needed when an additional aggregation is called. To translate the call, the columns to group by are stored until the aggregation functions is called.

A lookup table helps to translate the aggregation functions, which in pandas are named differently to those in SQL. It is necessary to rename or reimplement some custom functions from pandas before using them in SQL. For example, the pandas function `std` results in `stddev_pop` for SQL, `mean` in `avg`.

To enable inspection of aggregated values, their corresponding identifiers also have to be aggregated to an array using `array_agg` as part of the SQL query.

5.1.6 Drop Null Values. pandas' `dropna` function allows any row containing `null` values to be dropped (see Figure 5). In SQL, a simple selection suffices. The selection condition consists of concatenated and negated `is null` blocks.

A	B	C	Tracking
null	-5	"XX"	1
null	null	null	2
3	1	"NULL"	3

dropna

A	B	C	Tracking
3	1	"NULL"	3

Figure 5: Example for table altered by the `dropna` function.

5.1.7 Replace. `replace` substitutes any value contained in a pandas object with another. By default, only whole strings are changed, but replacements by a regular expression are also possible. The SQL function `REGEXP_REPLACE` fits as a general solution, as alternatives like SQL's `REPLACE` function replaces each instance of a word in an entry with a new one. To force the replacement of whole strings, the regular expression special characters ^ and $ match the beginning of the string as well as the end (Listing 12).

```
SELECT REGEXP_REPLACE("label",'^Medium$','Low') AS "label"
FROM origin
```

Listing 12: Replace in SQL.

5.1.8 Row-Wise Operations. The term *row-wise* refers to operations, where the n-th line of one table is combined with the n-th line of another table. By default, any pandas table contains an index column, but it is not semantically part of the table. Porting tb1 and tb2 to SQL causes no problem (Listing 13), although row-wise operations on columns of different source tables require an index column. We support the combination of two tables by the row number to mimic pandas functionality. The relational model is set-oriented, whereas pandas implicitly joins two tables with the same number of rows by the row number. The tuple identifier cannot be used to indicate the row number. Therefore, the function that creates a table, e.g. `add_csv`, also adds an index column if required (`add_mlinspect_serial=True`). It represents the row number needed for row-wise operations in SQL (Listing 14).

```
tb1 = pandas.read_table("tb1.csv")
tb2 = pandas.read_table("tb2.csv")
tb1['new_column'] = tb2['column']
```

Listing 13: Example row-wise operation in Python.

```
CREATE TABLE tb1 (index_ serial, "column" text);
CREATE TABLE tb2 (index_ serial, "column" text);
COPY tb1 ("column") FROM 'tb1.csv' WITH (DELIMITER ',', NULL '',
      FORMAT CSV, HEADER TRUE);
COPY tb2 ("column") FROM 'tb2.csv' WITH (DELIMITER ',', NULL '',
      FORMAT CSV, HEADER TRUE);
SELECT tb1.index_, tb2."column" AS "new_column"
FROM tb1 INNER JOIN tb2 ON tb1.index_ = tb2.index_
```

Listing 14: Example row-wise operation in SQL.

5.2 Supported Scikit-Learn Functions

For many transformations in a preprocessing pipeline, specific parameters are calculated before the first use. For example, the *SimpleImputer* from scikit-learn (numbers 16 and 21 in Figure 1) calculates the replacement for `null` values before substitution [9]. The fitting step of a pipeline calculates these prerequisites for certain steps. The fitting needs to occur before transforming the passed data (see Figure 6). Furthermore, calculating the fitting parameters can be performed independently of transforming the data. If fitting was performed each time a transformation is applied, the results would not be consistent. For example, the substitution for `null` values could be different for the training and the test set. This means that certain values need to be calculated exactly once and be accessible in the query from then on. These values should be materialised or cached, so that they do not need to be recalculated in each step.

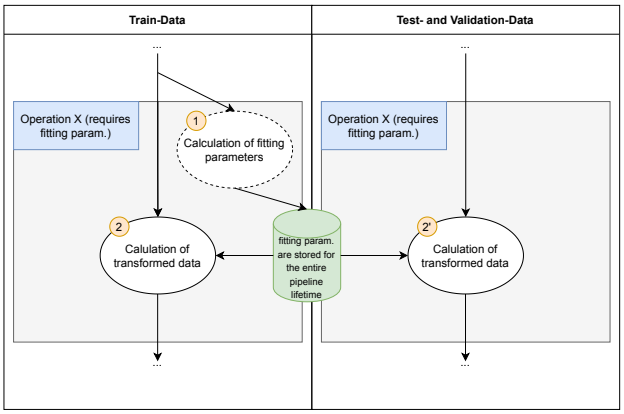

Figure 6: Relation between fitting and transforming.

5.2.1 Simple Imputer. The Simple imputer—as implemented in scikit-learn—replaces all `null` values in a table based on some metric, for example, *most_frequent*, *constant*, *mean* and *median*. The SQL query to realise an impute consists of a nested subquery that aggregates the values according to a metric for the substitute and returns a single value to replace `null` values (Listing 15). For the case *most_frequent*, the value that appears the most frequently is the substitute. The SQL function `COALESCE` assigns the first non-`null` value. As the values for substitution are calculated in the fitting step and are reused in each transformation step, their table expression is a candidate for materialisation.

```
WITH counts_help AS (
 SELECT "label", COUNT(*) AS count FROM origin GROUP BY "label"
), substitute AS (
 SELECT "label" AS most_frequent FROM counts_help
 WHERE counts_help.count = (SELECT MAX(count) FROM counts_help)
 LIMIT 1
)
SELECT COALESCE("label", (SELECT * FROM substitute)) AS "label"
FROM origin
```

Listing 15: Simple imputer in SQL.

5.2.2 One-Hot-Encoder. One-hot-encoding transforms categorical data into a binary vector. The idea is to create an array with its size equal to the number of categories out of zero values besides a single one at the position of the current category.

The function `array_fill` creates an array filled with a constant value of a length passed through a second parameter. So one-hot-encoding can be implemented by taking the number n, which represents the number of categories (e.g., by counting distinct entries or with `RANK`), and creating an array with $n-1$ zeroes followed by a one and $\#categories - n$ zeroes once again. After creating such a representation for each sensitive column, a join brings all the values together to the final table (Listing 16).

```
WITH original_label_onehot AS (
 SELECT "label", array_fill(0, ARRAY["rank"-1]) || 1 ||
       array_fill(0, ARRAY[cast((select count(distinct("label"))
       from original) AS int) - "rank"]) AS "one_hot"
 FROM (
    SELECT "label", CAST(ROW_NUMBER() OVER() AS int) AS "rank"
    FROM (select distinct("label") from original) oh
 ) one_hot_help
)
SELECT "one_hot" AS "label"
FROM original, original_label_onehot
WHERE original."label" = original_label_onehot."label"
```

Listing 16: One-hot-encoding in SQL.

5.2.3 Standard Scaler. The standard score[6] z of a sample $l \in X$ is calculated as $z(l) = \frac{l - mean(x \in X)}{stddev(x \in X)}$. The mean and standard deviation is calculated in the fitting step (Listing 17) and reused for any other transformation.

```
SELECT (("label" - (SELECT AVG("label") FROM origin)))
       / (SELECT STDDEV_POP("label") FROM origin) AS "label"
FROM origin
```

Listing 17: Standard scaler in SQL.

5.2.4 KBins Discretizer. The KBins discretizer from scikit-learn creates k intervals or bins based on the training data in the fitting step. It then assigns each input value to a specific bin and returns an encoding representing the assigned bin. The first step is to calculate the $step\_size = \frac{max(x \in X) - min(x \in X)}{k}$.

All values smaller than $min(x \in X) + step\_size$ are assigned to the first bin, which usually is 0. All values l where

$$min(x \in X) + step\_size \leq l < min(x \in X) + 2 \cdot step\_size$$

holds belong to the second bin and similarly for greater values. So the bin is calculated as

$$bin(l) = \lfloor \frac{l - min(x \in X)}{step\_size} \rfloor.$$

Values smaller than $min(x \in X)$ and bigger than $max(x \in X)$ are possible, because the training data does not necessarily provide values smaller and bigger than the testing set, hence the edge cases are handled with the help of `LEAST` and `GREATEST` (Listing 18). Different strategies for calculating the bins exist, currently only the explained strategy called *uniform* is implemented.

```
SELECT LEAST(3, GREATEST(0,
 FLOOR(("label" - (SELECT MIN("label") AS min_val from origin))
 / (((SELECT MAX("label") FROM origin) - (SELECT MIN("label")
       FROM origin)) / 4 AS step))))
 AS "label" FROM origin
```

Listing 18: KBins discretizer in SQL (4 bins).

5.2.5 Binarize. *Binarize* decodes a value as one when surpassing a threshold or as zero otherwise, which is translated into a case statement in SQL (Listing 19).

```
SELECT (CASE WHEN ("label" >= 50) THEN 1 ELSE 0 END) AS "label"
FROM origin
```

Listing 19: Binarize in SQL (threshold 50).

[6] https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing

healthcare	compas	adult simple	adult complex
read_csv	read_csv	read_csv	read_csv
merge	*miscellaneous operations*	dropna	label_binarize
groupby	replace	label_binarize	SimpleImputer
agg	label_binarize	StandardScaler	OneHotEncoder
miscellaneous operations	SimpleImputer		StandardScaler
isin	OneHotEncoder		
SimpleImputer	KBinsDiscretizer		
StandardScaler			

Table 1: Pipelines with pandas and scikit-learn operations.

	healthcare	compas	adult
#Tuples	889	2167	9771
Generated size	10^2 to 10^6	10^2 to 10^6	10^2 to 10^6
Files	histories.csv, patients.csv	compas_test.csv, compas_train.csv	adult_test.csv, aduld_train.csv
Schema (key, sensitive column)	`histories`: {[smoker, complications, ssn]} `patients`: {[id, first_name, last_name, race, county, num_children, income, age_group, ssn]}	{[, id, name, first, last, compas_screening_date, sex, dob, age, age_cat, race, juv_fel_count, decile_score, juv_misd_count, juv_other_count, priors_count, days_b_screening_arrest, c_jail_in, c_jail_out, c_case_number, c_offense_date, c_arrest_date, c_days_from_compas, c_charge_degree, c_charge_desc, is_recid, r_case_number, r_charge_degree, r_days_from_arrest, r_offense_date, r_charge_desc, r_jail_in, r_jail_out, violent_recid, is_violent_recid, vr_case_number, vr_charge_degree, vr_offense_date, vr_charge_desc, type_of_assessment, decile_score.1, score_text, screening_date, v_type_of_assessment, v_decile_score, v_score_text, v_screening_date, in_custody, out_custody, priors_count.1, start, end, event, two_year_recid]}	{[*row-number*, age, workclass, fnlwgt, education, education-num, marital-status, occupation, relationship, race, sex, capital-gain, capital-loss, hours-per-week, native-country, income-per-year]}

Table 2: Datasets used.

6 EVALUATION

By extending the mlinspect framework by code-generation to SQL, we provide full support for existing pipelines within mlinspect while off-loading compute-intensive tasks to the database system. To measure the performance increase through the use of a database system, we compare the performance of selected pipelines using existing library functions to either common table expressions (CTEs) or materialised views.

The pipelines are taken from the mlinspect repository[7] and their names were not changed. The *healthcare* and *compas* pipeline work on the healthcare and compas dataset respectively. The pipelines called *adult simple* and *adult complex* both work on the adult dataset. Table 1 shows all the considered pipelines with the operations from pandas and the one from scikit-learn marked in grey. The benchmarks in Section 6.1 evaluate only the pandas operations and their SQL counterparts, whereas Section 6.2 also includes the scikit-learn operations and their translations into SQL. Section 6.3 presents the additional overhead when enabling inspection to detect technical biases. Finally, Section 6.4 shows the runtime of the original pipeline including training a neural network in Keras but with data preprocessing and inspection outsourced to a database system. Section 6.5 breaks down the runtime by the performed operations. Section 6.6 measures the impact of the number of sensitive columns on the runtime using the New York Taxi dataset. The depicted runtimes enclose a call to the *psycopg2* adapter to run the query, so they include optimisation, compilation and execution time within the database systems. Transpilation to SQL took about 100ms for each pipeline (see Table 3), which we added to the measurements.

The original datasets were either extended with generated mocking data or by replicating the original one until the desired size is reached (see Table 2). The used approach depends on the pipeline that is inspected, as some operations (e.g. join) will lead to big subresults for just replicated data. The first column of the compas and adult dataset each contains the pandas row numbers without a header. The mlinspect authors considered this case intentionally, as pandas recognises and handles missing headers.

In the following subsections, the performance differences are presented for all example pipelines in the mlinspect project repository. This is possible as transpilation to SQL including inspection is available without requiring any changes to the target pipeline.

System: Ubuntu 20.04.2 LTS, four cores of Intel i7-7700HQ CPU, running at 2.80 GHz clock frequency each, and 16 GiB RAM.
Software: PostgreSQL 12.7, Umbra, Python 3.8.10, Pandas 1.3.0

[7] https://github.com/stefan-grafberger/mlinspect

	pandas		+ scikit-learn		+ inspection	
	VIEW	CTE	VIEW	CTE	VIEW	CTE
healthcare	0.058s	0.050s	0.072s	0.074s	0.134s	0.079s
compas	0.076s	0.080s	0.114s	0.113s	0.114s	0.120s
adult simple	0.017s	0.018s	0.037s	0.035s	0.036s	0.035s
adult complex	0.029s	0.031s	0.049s	0.047s	0.062s	0.050s

Table 3: Transpilation time to SQL.

6.1 Pandas-to-SQL Performance

We first compare the performance of the operations in pandas only to the SQL translations. For each pipeline, all code up to the last line containing pandas code was translated to SQL and benchmarked against the original pipeline code up to the same line (see Figure 7a). For the comparison with pandas code, no view/CTE in SQL needs to be materialised, as all parts of the query are only executed once. The pipelines *adult simple* and *adult complex* have a negligibly small pandas part, which loads the data only.

We first discuss the performance of the pandas operations and their SQL translations for the *healthcare* pipeline. Umbra outperforms the operations in pandas by up to a factor of ten using CTEs and unmaterialised views (*VIEW*). PostgreSQL underperformed pandas with CTEs, but is able to outperform pandas with larger datasets using views by about a factor of two. It is interesting that the performance using CTEs differs from that using views within PostgreSQL. The reason is that the query for CTE did not contain any `NOT MATERIALIZED` statements to avoid materialising the intermediate CTEs. All CTEs—materialised or not—are accessed only once resulting in the inefficiency of materialising unnecessary data. If `NOT MATERIALIZED` was added to all CTEs, the performance would be closer to the VIEW one.

The second benchmark is based on the *compas* pipeline. This pipeline differs from the first one, which splits one dataset into train and test sets, as it uses two different datasets for training and testing. The way the SQL translation is implemented, only the parts contributing to the result of the pipeline are executed, or alternatively the most recent call. Code not contributing to the pipeline, like prints or operations on unused tables, are skipped. For a fair comparison, we only execute the part of the compas pipeline handling the training data. The excluded lines would load the unused test set and apply a simple projection. Umbra outperformed pandas by up to a factor of five. PostgreSQL outperforms pandas as well but by a lesser margin than Umbra.

These results show that a modern database systems as Umbra performs data preprocessing faster than using dataframes in pandas. With unmaterialised views, PostgreSQL preprocess data faster than pandas dataframes.

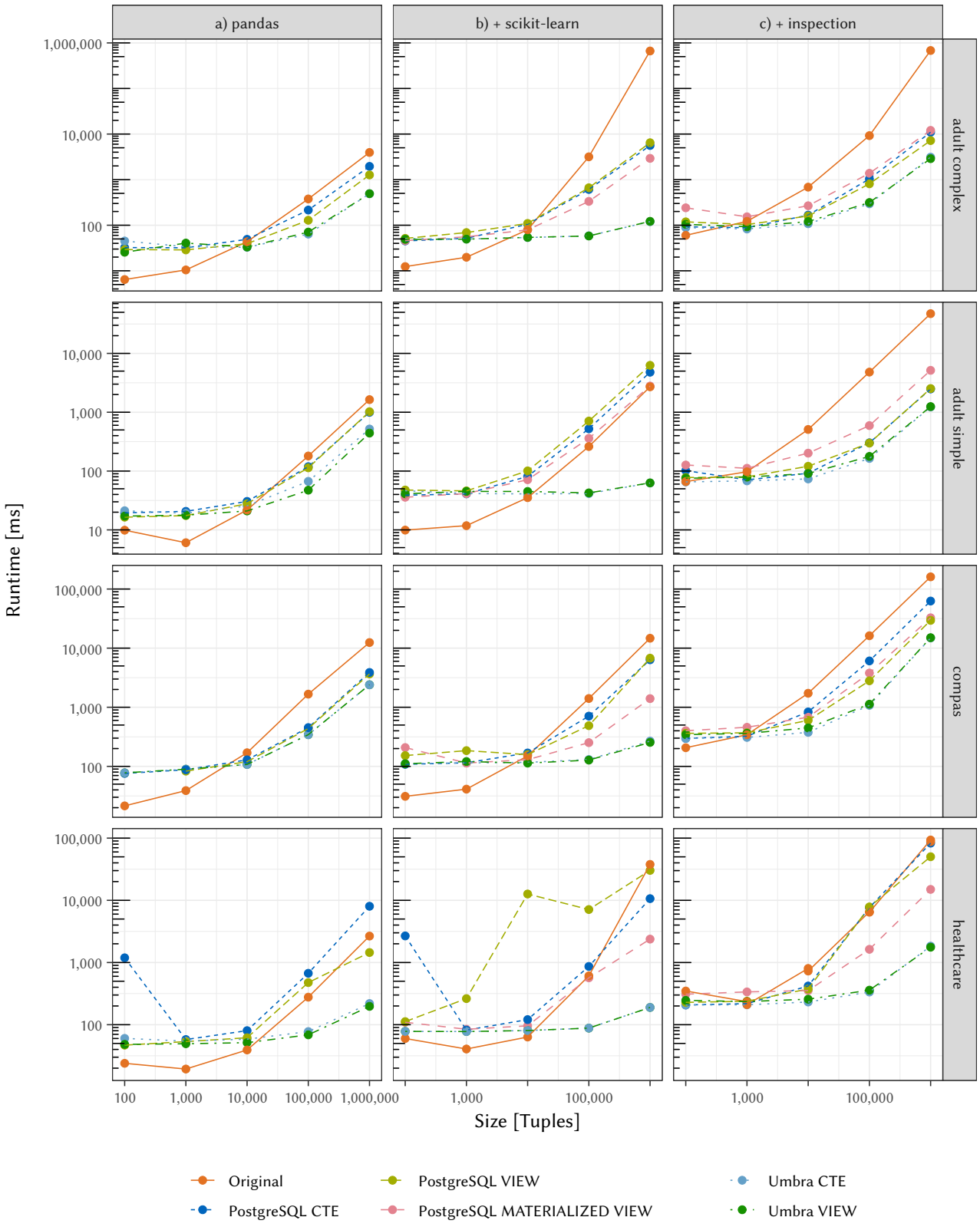

Figure 7: Runtime results for a) the operations in pandas, b) plus the operations in scikit-learn and c) plus inspection.

6.2 Pandas/Scikit-Learn-to-SQL Performance

In practise, it will be highly unlikely to find a pipeline just made of pandas operations [53]. This subsection evaluates the performance of the entire preprocessing pipeline but without training or inspection. This additionally includes operations from the scikit-learn library. Scikit-learn calculates the fitting parameters, which subsequent steps in the pipeline access afterwards. As monkey-patching rewrites each individual function call, this generates one table expression for each fitting parameter although each parameter is based on the same tuples. So materialising the table expressions needed for the fitting parameters accelerates the PostgreSQL runs with VIEW as this avoids redundant executions.

PostgreSQL outperforms pandas in most of the runs (see Figure 7b). For the *healthcare* pipeline, Umbra outperformed the original one by up to a factor of 150 and even the run in PostgreSQL with materialised views by up to a factor of 10. The performance gain with Umbra for 10^6 tuples varied between a factor of 40 (*adult simple*) and $5 \cdot 10^3$ (*adult complex*). Since the measurements include the transpilation time to generate the SQL queries, the plots indicate that the additional overhead is amortised from at least 10^4 tuples. To conclude, database systems show their full potential when preprocessing large datasets. Next, we will show how inspection influences the runtime.

6.3 Inspection Performance

This subsection discusses the performance when data inspection for the pipelines is enabled. Here, the materialisation of intermediate results becomes even more important as the inspection is applied after each operation. n inspection steps lead to n executions of the first operation, $n-1$ executions of the second operation and so on.

With an increasing number of tuples, all database systems execute the additional inspection steps faster (see Figure 7c). For 10^6 tuples, PostgreSQL outperformed the original inspection run with pandas/scikit-learn by between a factor of five (*compas*) and almost 100 (*adult complex*). Umbra accelerated inspection on the same datasets by a factor of 7 (*compas*) and 200 (*adult complex*) compared to the original run. To summarise, our approach with SQL queries accelerates inspection of pipelines out of pandas and scikit-learn operations on datasets of size 10^4 onwards.

6.4 End-to-End Performance Comparison

To show that this approach can substitute whole end-to-end preprocessing pipelines and integrates well with training, this subsection benchmarks the whole pipeline with the original datasets of mlinspect (9771 tuples for the adult, 889 tuples for the healthcare and 2167 tuples for the compas dataset). This will show the advanced state of the implementation and its potential when transferring and transforming data as investigated in the next subsection in more detail. Both pipelines on the adult dataset predict the income (less or greater than/equal to \$50k), the *healthcare* pipeline computes a score as the probability of complications and the *compas* pipeline uses logistic regression to predict a score (high/low). The correctness is verified by comparing the equality of the intermediate results (see Figure 9, Table 4). The accuracies of the trained model for the *healthcare* and the *adult complex* pipeline differ slightly because of the stochastic nature of the train/test split and the training (see Table 5).

For the presented end-to-end runs, the adult and the compas datasets are large enough that the corresponding pipelines benefit

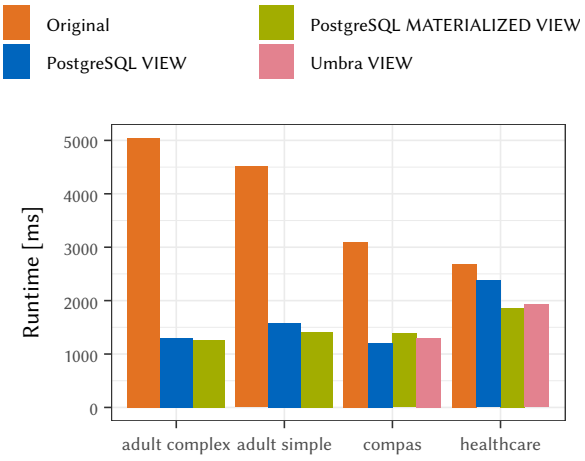

Figure 8: End-to-end performance comparison.

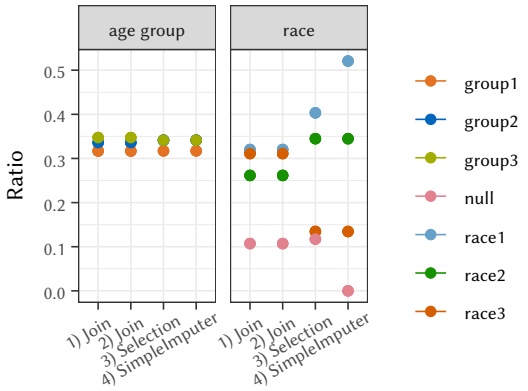

Figure 9: Ratio changes during preprocessing (*healthcare*).

Attribute	Before	After
Female	0.192117	0.191105
Male	0.807883	0.808895
African-American	0.512577	0.516099
Asian	0.00455536	0.00509613
Caucasian	0.338087	0.338661
Hispanic	0.0895227	0.0826963
Native American	0.00277283	0.00185314
Other	0.0524856	0.0555942

(a) *compas*.

Attribute	Before	After
Amer-Indian-Eskimo	0.00956476	0.00936392
Asian-Pac-Islander	0.0323359	0.0301253
Black	0.0945068	0.0925514
Other	0.00864338	0.00794514
White	0.854949	0.860014

(b) *adult simple*.

Table 4: Ratios before/after preprocessing.

	avg	median	min	max
adult simple	0.8779	0.8779	0.8779	0.8779
adult complex	0.7624	0.7622	0.7620	0.7632
healthcare	0.9068	0.9041	0.8767	0.9589
compas	0.8079	0.8079	0.8079	0.8079

Table 5: Model accuracy measurements (5 runs).

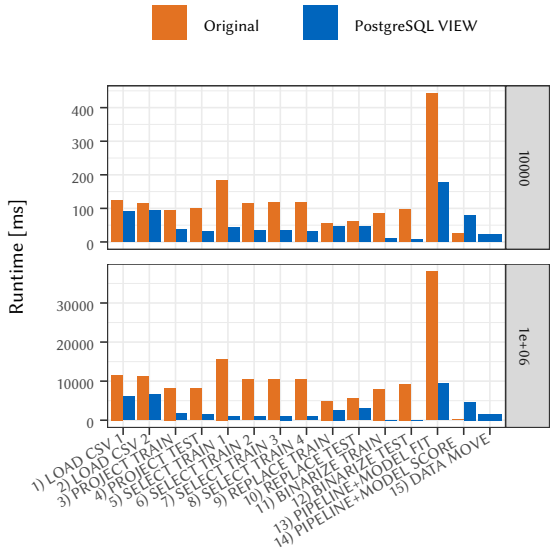

Figure 10: Operation level performance (*compas* pipeline).

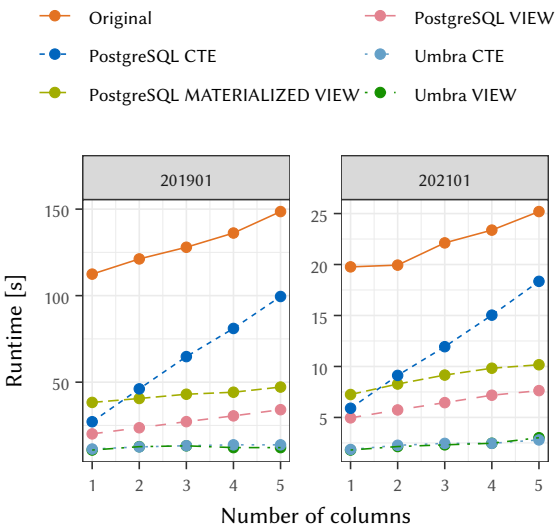

Figure 11: Runtime depending on the number of columns.

from accelerated preprocessing (see Figure 8). For the *healthcare* pipeline, PostgreSQL and the default pipeline seem to perform about the same. The reason is that the training time for the model is significantly higher here, which results in a smaller impact from the accelerated preprocessing. Processing within the database systems requires extraction, the transforming and loading of data for subsequent model training. The transformation consists of a cast into a matrix representation (NumPy array) to feed the model. This step will be skipped if the data already lies with the database system and the training happens there as well, as proposed for Umbra and HyPer [33, 34, 41, 45, 46].

6.5 Operation Level Performance

System (this experiment only): Ubuntu 20.04.2 LTS, AMD Ryzen 7 4800H, 8 cores (16 threads), 2.9 GHz, 32 GB of DDR4 RAM (3200 MHz), 512 GB of SSD (1x M.2 2280 SSD via PCI Express 3.0 x4).

To investigate how and where the speedups come from, Figure 10 shows the *compas* pipeline broken down into each operation. The two plots show the pipeline with replicated data for two different input sizes (10^4/10^6 tuples).

When inspecting the runtimes for the different pipelines, most operations are performed faster in PostgreSQL than in Python. The binarize operations took about 10 ms for PostgreSQL (a simple case statement in SQL) and therefore are hardly visible within the plots. The scoring step to calculate the accuracy accesses all tuples and is therefore time-consuming within a disk-based database system. The time PostgreSQL loses due to additional data loading and extraction could be eliminated when all computations happen inside database systems.

Overall, a significant increase in performance was achieved throughout all pipelines. This is especially promising as the examples were not artificially constructed or simplified. Instead, realistic pipelines were tested. The price for data loading and transformation is a worthwhile one compared to the other performance gains. The performance of Umbra paired with the additional benefits of training the models in the database system appears to be a promising approach worth pursuing.

6.6 Varying the Number of Columns

To measure the impact of the number of inspected columns on the runtime, we vary the number of sensitive columns. We use one month—January 2019 (7,667,793 tuples) and January 2021 (cleaned: 1,271,414 tuples)—of the New York Taxi dataset[8], where we perform one operation (selection: `passenger_count > 1`) only. We initially inspect the column *passenger_count* only and expand the inspection stepwise to encompass the following columns: *trip_distance*, *PULocationID*, *DOLocationID* and *payment_type*. Figure 11 shows the runtime depending on the number of inspected columns. All database systems execute the inspections faster than pandas. In PostgreSQL, the runtime of the CTE version increases linearly with the number of inspected columns, which is expected as the whole query runs once for each column. Nevertheless, the view version performs better as views are part of the holistic query optimisation in PostgreSQL. In Umbra, both versions show similar performance.

7 CONCLUSION

This paper has presented SQL queries for bias detection caused by data preprocessing. By exposing the tuple identifier throughout all operations, we were able to track the distribution frequency within columns that were even removed during preprocessing. In order to automatically generate these queries, we extended the mlinspect framework with a backend that replaces calls to Python library functions with SQL queries using monkey-patching. Our backend translated each line of the original Python source code into one CTE/view. One CTE or view per sensitive column calculated the ratios needed to detect introduced biases. Our evaluation revealed that materialised views in PostgreSQL could improve the overall preprocessing and inspection performance for larger datasets. The performance could be further improved using Umbra with in-memory performance even without the need to materialise intermediate results. An extension to support further library functions such as for training a model would eliminate the remaining need for final data transfer and thus further improve the performance.

[8] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

REFERENCES

[1] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. www.cidrdb.org.

[2] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768.

[3] Mike Brachmann, Carlos Bautista, Sonia Castelo, Su Feng, Juliana Freire, Boris Glavic, Oliver Kennedy, Heiko Mueller, Rémi Rampin, William Spoth, and Ying Yang. 2019. Data Debugging and Exploration with Vizier. In *SIGMOD Conference*. ACM, 1877–1880.

[4] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: visualization meets data management. In *SIGMOD Conference*. ACM, 745–747.

[5] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD Conference*. ACM, 1701–1716.

[6] Ralf Diestelkämper, Seokki Lee, Boris Glavic, and Melanie Herschel. 2021. Debugging Missing Answers for Spark Queries over Nested Data with Breadcrumb. *Proc. VLDB Endow.* 14, 12 (2021), 2731–2734.

[7] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *SIGMOD Conference*. ACM, 2736–2739.

[8] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. 2021. Lightweight Inspection of Data Preprocessing in Native Machine Learning Pipelines. In *CIDR*. www.cidrdb.org.

[9] G Hackeling. 2017. *Mastering Machine Learning with scikit-learn*. Packt Publishing. https://books.google.de/books?id=9-ZDDwAAQBAJ

[10] Stefan Hagedorn. 2020. When sweet and cute isn't enough anymore: Solving scalability issues in Python Pandas with Grizzly. In *CIDR*. www.cidrdb.org.

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

[12] H Hapke and C Nelson. 2020. *Building Machine Learning Pipelines*. O'Reilly Media. https://books.google.de/books?id=H6%5C_wDwAAQBAJ

[13] Melanie Herschel and Hanno Eichelberger. 2012. The nautilus analyzer: understanding and debugging data transformations. In *CIKM*. ACM, 2731–2733.

[14] Nina C. Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. 2017. HyPerInsight: Data Exploration Deep Inside HyPer. In *CIKM*. ACM, 2467–2470.

[15] John Hunt. 2019. Monkey Patching and Attribute Lookup. In *A Beginners Guide to Python 3 Programming*. Springer International Publishing, Cham, 325–336. https://doi.org/10.1007/978-3-030-20290-3_28

[16] Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. 2021. Umbra as a Time Machine. In *BTW (LNI, Vol. P-311)*. Gesellschaft für Informatik, Bonn, 123–132.

[17] Raoni Lourenço, Juliana Freire, and Dennis E. Shasha. 2019. Debugging Machine Learning Pipelines. In *DEEM@SIGMOD*. ACM, 3:1–3:10.

[18] Raoni Lourenço, Juliana Freire, and Dennis E. Shasha. 2020. BugDoc: Algorithms to Debug Computational Processes. In *SIGMOD Conference*. ACM, 463–478.

[19] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[20] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[21] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI, Vol. P-241)*. GI, 383–402.

[22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[23] Evaggelia Pitoura, Georgia Koutrika, and Kostas Stefanidis. 2020. Fairness in Rankings and Recommenders. In *EDBT*. OpenProceedings.org, 651–654.

[24] Romila Pradhan, Jiongli Zhu, Boris Glavic, and Babak Salimi. 2021. Interpretable Data-Based Explanations for Fairness Debugging. *CoRR* abs/2112.09745 (2021).

[25] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. 2021. One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA. In *ADMS@VLDB*. 17–26.

[26] Yuji Roh, Kangwook Lee, Steven Euijong Whang, and Changho Suh. 2021. FairBatch: Batch Selection for Model Fairness. In *ICLR*. OpenReview.net.

[27] Ibrahim Sabek and Mohamed F. Mokbel. 2021. Machine Learning Meets Big Spatial Data (Revised). In *MDM*. IEEE, 5–8.

[28] Svetlana Sagadeeva and Matthias Boehm. 2021. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In *SIGMOD Conference*. ACM, 2290–2299.

[29] Nico Schäfer and Sebastian Michel. 2020. Partially Materializable Delta Trees for Efficient Data Wrangling of Semi-Structured Contents. In *EDBT*. OpenProceedings.org, 399–402.

[30] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD Conference*. ACM, 1642–1659.

[31] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2021. B^2-Tree: Cache-Friendly String Indexing within B-Trees. In *BTW (LNI, Vol. P-311)*. Gesellschaft für Informatik, Bonn, 39–58.

[32] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2022. B^2-Tree: Page-Based String Indexing in Concurrent Environments. *Datenbank-Spektrum* 22, 1 (2022), 11–22.

[33] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*. ACM, 7:1–7:4.

[34] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT*. OpenProceedings.org, 562–565.

[35] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2021. ArrayQL for Linear Algebra within Umbra. In *SSDBM*. ACM, 193–196.

[36] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2022. ArrayQL Integration into Code-Generating Database Systems. In *EDBT*. OpenProceedings.org, 1:40–1:51.

[37] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM, 6:1–6:12.

[38] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *SSDBM*. ACM, 169–180.

[39] Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. 2022. Recursive SQL for Data Mining. In *SSDBM*. ACM.

[40] Maximilian E. Schüle, Alex Kulikov, Alfons Kemper, and Thomas Neumann. 2020. ARTful Skyline Computation for In-Memory Database Systems. In *ADBIS (Short Papers) (Communications in Computer and Information Science, Vol. 1259)*. Springer, 3–12.

[41] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM*. ACM, 25–36.

[42] Maximilian E. Schüle, Linnea Passing, Alfons Kemper, and Thomas Neumann. 2019. Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme. In *BTW (LNI, Vol. P-289)*. Gesellschaft für Informatik, Bonn, 107–126.

[43] Maximilian E. Schüle, Pascal Schliski, Thomas Hutzelmann, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. 2017. Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. *Proc. VLDB Endow.* 10, 12 (2017), 1921–1924.

[44] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. 2021. TardisDB: Extending SQL to Support Versioning. In *SIGMOD Conference*. ACM, 2775–2778.

[45] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW (LNI, Vol. P-289)*. Gesellschaft für Informatik, Bonn, 247–266.

[46] Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, and Thomas Neumann. 2022. LLVM code optimisation for automatic differentiation: when forward and reverse mode lead in the same direction. In *DEEM@SIGMOD*. ACM, 5:1–5:4.

[47] Maximilian E. Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. The Power of SQL Lambda Functions. In *EDBT*. OpenProceedings.org, 534–537.

[48] Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. 2021. An Intermediate Representation for Hybrid Database and Machine Learning Workloads. *Proc. VLDB Endow.* 14, 12 (2021), 2831–2834.

[49] Haipei Sun, Yiding Yang, Yanying Li, Huihui Liu, Xinchao Wang, and Wendy Hui Wang. 2021. Automating Fairness Configurations for Machine Learning. In *WWW (Companion Volume)*. ACM / IW3C2, 193–201.

[50] Ki Hyun Tae and Steven Euijong Whang. 2021. Slice Tuner: A Selective Data Acquisition Framework for Accurate and Fair Machine Learning Models. In *SIGMOD Conference*. ACM, 1771–1783.

[51] Matthew Welsh and Steve Begg. 2016. What have we learned? Insights from a decade of bias research. *The APPEA Journal* 56, 1 (2016), 435–450.

[52] Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation. *Proc. VLDB Endow.* 14, 12 (2021), 2715–2718.

[53] Doris Xin, Litian Ma, Shuchen Song, and Aditya G. Parameswaran. 2018. How Developers Iterate on Machine Learning Workflows - A Survey of the Applied Machine Learning Literature. *CoRR* abs/1803.10311 (2018).