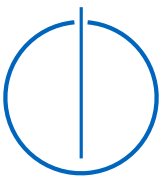
Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Transforming Data Frame Operations from Python to MLIR

Robert Imschweiler



Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Transformierung von Data-Frame-Operationen von Python zu MLIR

Transforming Data Frame Operations from Python to MLIR

Author: Robert Imschweiler
Supervisor: Prof. Dr. Thomas Neumann
Advisor: Alexis Engelke, Dr. rer. nat., Michael Jungmair, M.Sc.
Submission Date: 15. August 2022

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Robert Imschweiler

Abstract

In recent years, the analysis of large data sets has become increasingly important and a lot of tooling has been developed to support data scientists. While modern database management systems (DBMSs) have become increasingly efficient, they lack integration. The Python library pandas is one of the most popular software projects for working with data and provides excellent interoperability and convenience.
However, performance is a significant shortcoming of pandas. pandas does not perform well for large data sets and complex queries because of being implemented in Python and heavily relying on intermediate copies of the data.
We propose *mlir_pandas*, an approach for a transparent drop-in replacement for pandas, which translates the pandas operations to MLIR, an Intermediate Representation, which is then passed to LingoDB, a compiling DBMS based on MLIR.
In the performance evaluation of our implemented prototype for *mlir_pandas*, we can show an average speedup by a factor of 10 compared to pandas and a considerably reduced peak memory usage.

Contents

1 Introduction

In recent years, the amount of data to be processed and analyzed has dramatically increased. Modern business processes heavily rely on *big data* and evaluating numerous and sometimes very large data sets. An example use case is the medical and health care sector, where relevant data can help analyze complex bio-medical mechanisms or assist doctors in decision-making. The increasing size of the data and the ever more widespread use required the development of appropriate tooling.

pandas [38] is a Python library that is very common among data scientists. It provides an easy-to-use API for working with table-like structures called *data frames* and makes it possible to transform, query, or combine them in various ways. One of the major strengths of the pandas API is the interoperability: there is a rich set of conversion methods from and to data representations such as CSV or JSON. Additionally, pandas provides a built-in way to conveniently plot data using the library matplotlib. One crucial limitation of pandas is its performance. pandas works on data in-memory, and operations usually return a modified *copy* of the input data or work with intermediate copies. This limits the use of pandas, in particular for very large data sets.

With the rise of big data, efficient database management systems (DBMSs) have been considerably further developed and optimized for the increasing workload. A prevalent theory behind many DBMSs is the relational algebra model. It defines operators working on tuples organized in multi-sets. The mathematical abstraction from the data structures allows for concise optimizations within one operator and, even more importantly, for the data flow between multiple operators, which is optimized by selection pushdown or join re-ordering, for example. There are mainly two approaches for advancing the development of DBMSs: vectorizing and data-centric code generation. [18] An example for a vectorizing DBMS is DB2 Blu [34]. Compiling DBMSs with data-centric code generation are represented by systems such as Hyper [22, 24], Umbra [23] or LegoBase [19]. Finally, the efficient use of hardware parallelism has become increasingly important, for example, through multithreading.

LingoDB [17] is a modern compiling relational database management system (RDBMS). One major characteristic of LingoDB is that its engine is based on Multi-Level Intermediate Representation (MLIR) [20], a new Intermediate Representation (IR) by the LLVM project. MLIR is a compiler framework designed to provide a common IR for multiple abstraction levels to allow optimizations across the boundaries of software projects. LingoDB does not only provide an Structured Query Language (SQL) interface, but also accepts queries written in MLIR [20]. This has the advantage that queries can be built using optimizations across multiple domains.

We propose *mlir_pandas*, an approach for a transparent drop-in replacement for the pandas data frame API. We also implemented this in a Python library of the same name. *mlir_pandas* aims to leverage the similarity of the data frame API and the relational

algebra model to run relevant pandas operations in LingoDB instead of Python. To achieve this, *mlir_pandas* provides custom classes to replace the corresponding pandas classes. The *mlir_pandas* classes are completely transparent. They provide the same API as their original counterparts but internally transpile supported pandas operations to MLIR code which can then be passed to LingoDB to be compiled, optimized, and executed. The execution of the MLIR code happens lazily and is deferred until the materialized results are needed. Operations of the pandas API that are not (yet) supported transparently fall back to the original pandas implementation.

The results of our performance benchmarks on some TPC-H queries show that our implemented prototype for *mlir_pandas* provides a speedup by a factor of 10 on average compared to pandas. Also, the maximum memory consumption of pandas is more than halved on average. The overhead of *mlir_pandas* for management tasks is constant. Translating the original SQL query to MLIR and executing it directly in LingoDB still may provide better performance since it is able to build an easier-to-optimize MLIR module.

1.1 Contributions

Our contribution is *mlir_pandas*, a mapping of the pandas API to the MLIR interface of LingoDB and a transparent drop-in replacement for pandas.

1.2 Outline

The remainder of this thesis is structured as follows. Chapter 2 gives an introduction to the relevant software projects and technologies. In Chapter 3, we describe and discuss the design of *mlir_pandas*. The results of the performance evaluation and comparison are elaborated in Chapter 4. Chapter 5 shows the related work. Finally, Chapter 6 provides a summary and an outlook.

2 Background

In order to discuss the realization of the aforementioned goals of this thesis, we need to give a brief overview of some relevant topics. First, we introduce the essential aspects of pandas. Then, we will shortly describe modern DBMSs. This will be the basis for the section about LingoDB, which will be preceded by the description of MLIR. Finally, we will describe some characteristics of Python that will be useful for understanding the Python-related aspects of pandas and our approach.

2.1 Pandas

pandas [40, 21] is a Python [11] library which is widely used by data scientists. NVIDIA, for example, describes pandas as "the most popular software library for data manipulation and data analysis for the Python programming language." [2] Reasons for this are that pandas is easy to use and integrates with many other well-known libraries. For example, pandas can parse and export to data storage formats such as JSON, CSV, or XML. Additionally, it offers the possibility to plot data using matplotlib [41].

Data Types

For the internal data storage, pandas uses NumPy [33], short for "Numerical Python", which is a library providing elaborate data structures such as multidimensional arrays and matrices, as well as optimized mathematical functions. Additionally, NumPy maintains its own type system. [9] This is also where the integration of pandas and NumPy becomes immediately apparent. The supported data types of pandas are based on the type system of NumPy. [39] In addition to the standard NumPy types such as `float`, `int`, `bool`, `timedelta64[ns]` and `datetime64[ns]`, pandas enhances the datetime types with timezone awareness and adds an own string type, for example. Furthermore, pandas defines nullable types such as `Int64Dtype`. Finally, there is the special data type `object`, which can represent any Python object. For supported types, the `object` type should be avoided as it hinders special type-dependent operations not defined on the general `object` type.

The most common representations of data within pandas are the `Series` and `DataFrame` classes. They are typically sufficient for the "use cases in finance, statistics, social science, and many areas of engineering", as described by the pandas developers. [39] In the following sections, we will describe the `Series` class, the `DataFrame` class and typical operations on the `DataFrame` class.

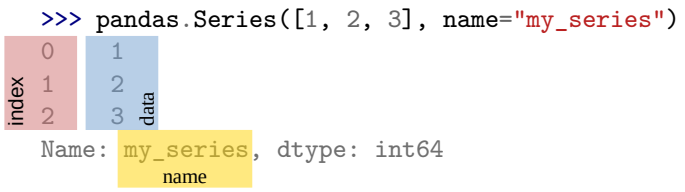


Figure 2.1: *The output format of a* `Series` *explained. The name attribute of a* `Series` *is optional.*

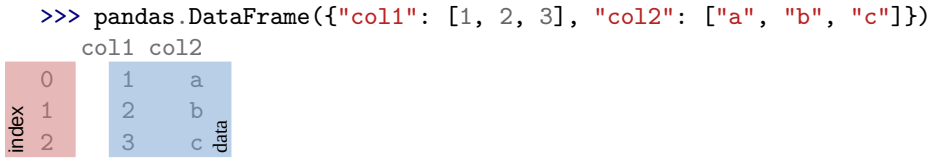


Figure 2.2: *The output format of a* `DataFrame` *explained.*

The `Series` class

A `Series` is an array-like one-dimensional data structure. Its representation is demonstrated in Figure 2.1. However, unlike a basic array or similar rather simple structures such as lists in Python, the elements of a `Series` may also be labeled and accessed similar to a dictionary. This is implemented using an index structure. The index is stored in an array-like data structure of type `Index`. Additionally, a `Series` is not only capable of storing and accessing a certain number of elements but also supports a rich set of high-level operations. See Listing 2.1 for a simple example.

The `DataFrame` class

A `DataFrame` is a tabular representation of data and can handle columns holding different data types, as known from database tables where each column has its own data type, independent of the types of the other columns. A `DataFrame` thus resembles a relation in a relational database system. Figure 2.2 shows how a `DataFrame` is presented py pandas.

However, a `DataFrame` is not limited to a simple tabular form. For example, every `DataFrame` has an index holding the row labels. By default, the row labels are an ascending numeric index value, but they do not necessarily need to be unique (except for specific operations), and they can be changed to some other representation such as string labels. The latter are a usual choice for the column labels of a `DataFrame`, which are organized in a separate object of type `Index`. The index objects can also be named, which is relevant for some `DataFrame` methods. The example in Listing 2.2 shows a `DataFrame` with numerical row labels from zero to two and strings as column labels.

The `Series` class is used to represent a column of a `DataFrame` to the user and let the user interact with it. One example is the access to a column of a `DataFrame`. This operation returns a `Series` object containing the data of the specified column. But representing the contents of a column is not the only aspect where the `Series` class works in combination with a `DataFrame`. A `Series` object can also be used to select rows of a `DataFrame`. This so-called "boolean indexing" is a pandas way of implementing the

Listing 2.1: *The overloaded comparison of `Series` objects results in a `Series` of boolean values. A transformation of a `Series` can be achieved by a map operation, which applies a function on each value of the `Series`.*

```
>>> pandas.Series([1, 2, 3]) >= pandas.Series([2, 2, 2])
0    False
1     True
2     True
dtype: bool
>>> pandas.Series([5, 7, 9]).map(lambda x: x**2)
0    25
1    49
2    81
dtype: int64
```

Listing 2.2: *A `DataFrame` and the types of its elements (per column). The objects containing the row and column labels are unnamed in this case. The last line of the output means that the column labels are of type `object`.*

```
>>> pandas.DataFrame({"col1": [1, 2, 3], "col2": ["one", "two", "three"]})
   col1   col2
0     1    one
1     2    two
2     3  three
>>> pandas.DataFrame({"col1": [1,2,3], "col2": ["one", "two", "three"]}).dtypes
col1     int64
col2    object
dtype: object
```

Listing 2.3: *Selecting rows from a* `DataFrame` *using a* `Series` *of boolean values.*

```
>>> df = pandas.DataFrame({"col1": [1, 2, 3], "col2": [4, 5, 6]})
>>> df[pandas.Series([True, False, True])]
   col1  col2
0     1     4
2     3     6
```

selection of the relational algebra and is shown in Listing 2.3.

Common operations on the `DataFrame` class

Internally, pandas uses NumPy arrays to store the data of a `DataFrame`. The operations on a `DataFrame` are not lazily computed, and even in-place operations may produce intermediate copies. The order of the rows of a `DataFrame` is mostly specified since pandas uses arrays as the internal storage format and the users of the pandas API expect array-like behavior and not set semantics. A `DataFrame` can be seen as similar to a dictionary where the keys are the column labels and the values `Series` objects.

Indexing. One of the most basic operations on a `DataFrame` is accessing columns using *indexing*. Indexing behaves just as one would expect from an array- or dictionary-like structure. A `DataFrame` can be indexed using one of its column labels, which results in a `Series` representing the corresponding column. This `Series` can then again be subscripted using the row labels to access the scalars contained in the column. Additionally, a `DataFrame` can be indexed by a list of column names. This produces a `DataFrame` instead of a single `Series`. A special type of indexing is the so-called *boolean indexing*, where we use a `Series` of boolean values to select rows from a given `DataFrame`.

Arithmetic Operations. Regarding other usual operations on array-like data structures, a `DataFrame` also allows arithmetic operations such as the addition of a constant, where the constant is added to every element of the `DataFrame`, or the (element-wise) addition of two `DataFrame` objects. Furthermore, simple algorithms on linear data structures are supported, such as the `min`/`max` operations, which operate per column. It is also possible to sort a `DataFrame`, e.g. according to one or multiple columns, using the `sort_values` method.

Join. Listing 2.4 shows an inner join of two `DataFrame` objects using the `merge` method. The explicitly-called `join` method is more specialized and does not support joining `DataFrame` objects on arbitrary columns. Left and right joins preserve the order of the rows from the corresponding `DataFrame` objects. The inner join and the cross-product preserve the order of the keys from the left `DataFrame`. An outer join sorts the keys lexicographically.

Listing 2.4: *Join two `DataFrame` objects using inner join on "col1".*

```
>>> pandas.DataFrame({"col1": [1,2,3], "col2": [4,5,6]}).merge(
...     pandas.DataFrame({"col1": [1,0,3], "col2": [7,8,9]}),
...     on="col1", how="inner"
... )
   col1  col2_x  col2_y
0     1       4       7
1     3       6       9
```

Listing 2.5: *Apply different aggregation functions on the columns of a `DataFrame`.*

```
>>> pandas.DataFrame({"col1": [1,2,3], "col2": [4,5,6]}).agg(
... {"col1": 'sum', "col2": 'mean'})
col1    6.0
col2    5.0
dtype: float64
```

Aggregation. Listing 2.5 shows an example for the aggregation method of a `DataFrame`. Although the result of the `sum` function is an integer, it gets converted to a floating point value so that the `Series` holding the aggregation results can be of homogeneous type. If we had used `min`, for example, instead of `mean`, the resulting data type would be an integer type.

GroupBy. The `groupby` method shown in Listing 2.6 is able to group the entries of a `DataFrame` according to one or multiple columns. By default, it uses the group labels, so the values of *col1* in this example, as the index of the resulting `DataFrame`. In order to have a more SQL-like experience, this has to be disabled explicitly. The `groupby` operation returns an object of type `DataFrameGroupBy`. This object holds the information on the collected groups and can perform operations such as aggregation on them (*sum*, *min*, *max*, etc.). The results of such operations may then again be of type `DataFrame` or `Series`. The order of the rows within the groups is preserved. [39]

Listing 2.6: *Group by "col1" and sum the values of each group.*

```
>>> pandas.DataFrame({"col1": [1,2,1], "col2": [4,5,6]}).groupby(
... ["col1"], as_index=False).sum()
   col1  col2
0     1    10
1     2     5
```

In-place Operations. Some methods of a `DataFrame` can be executed in-place by specifying the boolean parameter *inplace* of those methods to be true instead of false (the default). It is, however, discouraged to use this feature. First, it prevents method chaining as the methods no longer have a return value. Second, the result of those operations may still be calculated as a copy internally and then just copied back to the original `DataFrame`.

2.2 Modern Data Base Systems

In contrast to pandas, a modern DBMS provides a more optimized system for handling *big data*. In this work, the term *database system* specifically refers to the relational model, so to RDBMSs. The programming language commonly used to interact with an RDBMS is SQL. The relational model builds on the theory of first-order logic / predicate logic. This means that every SQL statement has an equivalent expression in first-order logic. This equivalence facilitates optimizations by exploiting the mathematical theory behind it.

Starting from a query given by the user of an RDBMS, the query optimizer, which is part of most modern DBMSs, builds a query plan. Since SQL is a declarative language, the user does not specify *how* the query should be executed, but only *what* should be done. This allows the query optimizer to construct a query plan which performs the specified query but does not necessarily follow the user-provided sequence of operations. Among the most common optimizations applied by the query optimizer are Selection Pushdown and Join-Order Optimization, which both aim to minimize the size of intermediate results.

Another important characteristic of DBMSs is the memory or disk buffer handling. In contrast to pandas, DBMSs do not necessarily hold all the data in memory. Although it is advantageous to have as much data as possible in the main memory, current approaches are able to achieve high performance even when it becomes necessary to spill data to the disk. [23]

A seemingly uninteresting property of DBMSs is that there is no guaranteed row ordering after certain operations such as selection or join. The developers of PostgreSQL explicitly state in their documentation that the order of the rows is unspecified after a query without a final sorting. [15] This is different from pandas, where the row order may be specified, as mentioned before. The relational model and its multi-set semantics are the reason for the unspecified row order in DBMSs. Relational algebra is defined on multi-sets; this means that the entries of a relation (the rows of a table) are contained in a set that can manage duplicates.

2.3 MLIR

MLIR [20] is a new IR and part of the LLVM project [31]. *Intermediate Representation* is a term used in compiler building to describe intermediate results during the compilation process. IR code is generated by the compiler from the source program written in a high-level language such as C or C++. The generated IR is then optimized and transformed into the machine code of the target architecture. The problem with the traditional IRs is that they are tailored for one specific purpose, also called *domain*. Those IRs are

typically restricted to *one* abstraction level. There is a lack of generality for a common representation that allows for multiple abstraction levels and is domain-independent. MLIR aims to fill this gap. It provides a common syntax and establishes an extensible basis.

Data Types

MLIR is statically typed. It defines built-in types and can additionally be extended with custom ones. The built-in types range from low-level types such as integers or floating-point types of different bit widths to more abstract types such as `tuple`. For integers, there exists a special notation to designate *signless* integers, i.e. integers whose signedness depends on the operation using them. [6] Table 2.1 shows a collection of important built-in types.

Table 2.1: *Important built-in types defined by MLIR.*

Type	Description
`i1`	1-bit signless integer type to represent a boolean value.
`i64`	64-bit signless integer type.
`f64`	64-bit floating-point type.
`(<TYPES_LIST>) -> <RESULT_TYPE>`	Type for a function. It consists of the list of operand types and the function's return type.

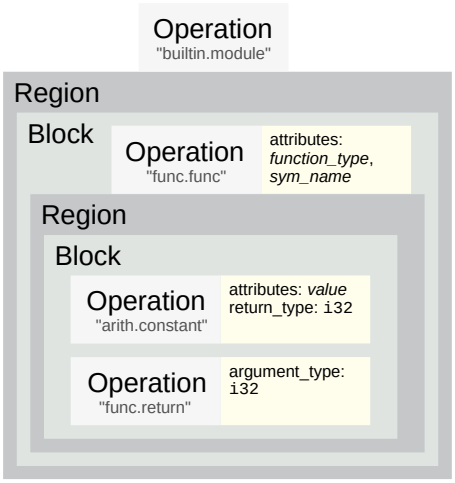
The structure of an MLIR module

The basic unit of MLIR code is an *operation*. An operation can take an arbitrary number of *operands* and returns an arbitrary number of *results*. Additionally, attributes can be attached to an operation. Operations themselves are organized in *blocks*, which are contained in *regions*. The blocks have to be labeled if there is more than one block in a region. Furthermore, each block can optionally take parameters. The parameters of the first block in a region, the *entry block*, are also the parameters of the region. Since a region can be passed to an operation similarly to an operand, it is possible to arbitrary nest operations, regions and blocks. The top-level operation of a piece of MLIR code is called a *module*. A module contains exactly one region with exactly one block. This block often holds function definitions, among other operations. Functions are operations with exactly one region (and at least one block). [20, 5] Figure 2.3a shows an example MLIR module, Figure 2.3b displays the corresponding simplified syntax tree.

Generic vs "pretty" format The syntax shown in Figure 2.3c is the generic format. It is uniform and meant for interoperability between different code generators and parsers. However, MLIR allows the definition and use of custom formats for easier reading. Typically, this would look like in Figure 2.3a.

```
module {
  func.func @main() -> i32 {
    %0 = arith.constant 42 : i32
    return %0 : i32
  }
}
```

(a) *A small MLIR module in "pretty" format.*



(b) *A simplified syntax tree for the code in Figure 2.3a.*

```
"builtin.module"() ({
  "func.func"() ({
    %0 = "arith.constant"() {value = 42 : i32} : () -> i32
    "func.return"(%0) : (i32) -> ()
  }) {function_type = () -> i32, sym_name = "main"} : () -> ()
}) : () -> ()
```

(c) *The code from Figure 2.3a in generic format.*

Figure 2.3: *A small MLIR module in different representations.*

MLIR Dialects

MLIR allows the definition of custom abstraction levels, called *dialects*. A dialect may consist of attributes, operations, and types. It defines a new namespace, which is always separated from the actual symbols by a single dot. Thus, defining a custom `tuple` type within a dialect called `relalg` would result in the symbol `!relalg.tuple`. The `!` prefix indicates a custom dialect type. The definitions of a dialect represent the functionality of the domain for which the dialect has been implemented.

There are several builtin dialects; among others, these are the explicitly-called `builtin` dialect, the `func` dialect and the `llvm` dialect. Important parts of the `builtin` dialect are core types such as the ones presented in Table 2.1 and the `builtin.module` operation which is to be used as top-level operation for an MLIR module. The `func` dialect provides high-level operations for defining, calling or returning from a function. Finally, the `llvm` dialect represents the LLVM IR within MLIR. The semantics of the dialect operations are expected to match the corresponding operations of the LLVM IR exactly. [4]

2.4 LingoDB

LingoDB [17] offers a relational database system capable of building and optimizing queries and executing them on data managed in memory. LingoDB is characterized by the fact that it provides an interface to MLIR and uses MLIR internally for optimizing and lowering. It implements the domain-specific custom MLIR dialects `db`, `dsa`, `relalg` and `util`. These dialects cover different abstraction levels and thus allow a *layered* query compilation, where the query is lowered and optimized step-by-step over multiple abstraction levels.

The `relalg` dialect. `relalg` is the most high-level dialect defined by LingoDB. It represents *rel*ational *alg*ebra operations such as selection, join or projection in a declarative manner, similar to SQL. They operate on the abstract data type `relalg.tuplestream`, which is built on the production and consumption of elements of type `relalg.tuple`. The `relalg` dialect also defines custom attributes. The most important ones are `relalg.columndef` and `relalg.columnref`. The first one is used to define a new column; the latter is needed to reference an existing column. Especially noteworthy is the fact that every column is not only defined by its name but also by a separate namespace, called *scope*. Therefore, the same column name may exist multiple times in a tuplestream as long as the scope of each occurrence is different.

The `db` dialect. The `db` dialect is an imperative abstraction of types and operations related to a database. This includes the data types `db.string` or `db.timestamp` and arithmetic or binary operations on them, for example. The data types do not allow null as a value, but there is also the special type wrapper `db.nullable` which enables null as a value for the wrapped type.

Supported Operations

Table 2.2 shows some important operations of the `relalg` dialect defined by LingoDB. The `relalg` operations represent the corresponding operations of the relational algebra. In order to pass predicates to an operation, a region can be attached to some of the operations. In the case of `relalg.selection`, for example, the specified region operates on a `!relalg.tuple`. It determines whether this tuple should be part of the resulting tuplestream or not. Finally, there are operations such as `relalg.map` which add columns to the tuples of a tuplestream. In this context, the scope of a column is important to avoid name collisions.

Table 2.2: *Important dialect operations defined by LingoDB.*

Operation	Description
`relalg.aggregation`	Group a tuplestream and apply functions on each group.
`relalg.basetable`	Load the given table and produce a tuplestream.
`relalg.join`	Perform an inner join on two tuplestreams.
`relalg.limit`	Limit the size of a tuplestream.
`relalg.map`	Compute new columns and add them to a tuplestream.
`relalg.materialize`	Materialize a tuplestream.
`relalg.projection`	Remove columns from a tuplestream.
`relalg.selection`	Select rows from a tuplestream.
`relalg.sort`	Sort a tuplestream according to the specified attributes.

Ordering Guarantees. As already described, pandas operates on arrays with a fixed order of elements maintained implicitly by every operation. A RDBMS has no such implicit row order because of the set semantics of relational algebra. LingoDB does not differ in that from the general RDBMS mentioned above. The row order is unspecified after `relalg.join`, `relalg.map`, `relalg.projection` and `relalg.selection`.

Compiling an MLIR module

In order to compile an MLIR module, LingoDB starts with the optimization. The optimization phase covers Simplification, Query Unnesting, Selection Pushdown, Join-Order Optimization, and physical optimization [17]. Those are the usual optimizations already found in modern database systems. After the optimization pass, the declarative `relalg` dialect entities used in the MLIR module get lowered to the imperative `db` dialect. The `db` dialect can then step-by-step be lowered to the low-level `llvm` dialect. By using stepwise lowering, each dialect can be created having reusability in mind. Furthermore, optimizations such as eliminating null-checks or Common Subexpression Elimination (CSE) can be applied at the most suitable abstraction level.

The resulting LLVM module will finally be optimized by the LLVM toolchain using standard compiler procedures such as Inlining, Control-Flow based optimizations, or Dead

Listing 2.7: *Python code with type hints.* `None` *means that* `f` *returns nothing.*

```
def f(name: str, number: int) -> None:
    foo: float = math.pi * number**2
    # ...
```

Code Elimination. In the end, the LLVM module gets compiled to machine code and can be executed by the LingoDB runtime.

2.5 Notes on Python

Since Python is easy to learn and fast to prototype, it is one of the most used programming languages nowadays. However, more complexity is available, which can enhance code quality or accomplish more complicated tasks. We will briefly describe some of those features in the following subsections.

Static Typing

Although being a dynamically typed language, Python provides regular support for static typing since version 3.5. [36, 37] Officially, the static typing additions for Python are called "type hints" or "type annotations" as types are **not** enforced during runtime. They only support static analysis tools such as mypy [32]. Listing 2.7 shows an example of a statically typed Python function. The built-in support for type hints can be extended by importing the module `typing` from the Python standard library. It provides additional types such as subscriptable types, e.g. `List[int]` or `Tuple[str, str]`, or special types such as `Any`, which does not impose any type restrictions.

Using static typing, we have to pay attention to the covariance of types. This is important for container types such as `List`. Roughly speaking, "covariance" allows using an instance of a subclass of the specified type. `List` is an "invariant" type. Therefore, a function taking a parameter of type `List[Class]` will be incompatible with an argument of type `List[SubClass]`. If the function uses covariance, its parameter type may be declared as `Sequence[Class]`. This would allow an argument of type `List[Class]` as well as one of type `List[SubClass]`.

Metaclasses

In Python, everything is an object. This implies that classes are objects, too. Therefore, the class objects need to be instantiated. Per default, this is done by the `type` class, which is the default metaclass for every Python class. However, the creation of a class object can be customized by specifying another metaclass. Listing 2.8 shows a simple example of using a metaclass.

Listing 2.8: *Add the class attribute "id" using a custom metaclass.*

```
class MetaA(type):
    def __new__(cls, name: str, bases: Tuple[Type[Any], ...], classdict:
    ↪  Dict[str, Any], **kwargs: Any) -> Type[Any]:
        obj: Type[Any] = type.__new__(cls, name, bases, classdict, **kwargs)
        setattr(obj, "id", 42)
        return obj

class A(metaclass=MetaA):
    pass

print(A.id) # prints "42"
```

Double Underscore Attributes/Methods

Probably, the most common double underscore method is `__init__`, the constructor of a class in Python. There are, however, far more default double underscore methods that can be added or overwritten in order to add or customize the functionality of a class. In Listing 2.9, the + operator has been overwritten using the `__add__` method and the conversion of an object of class `A` to a string is controlled by the `__str__` method.

The `__getattr__` and `__setattr__` methods are a central part of the *pythonic* attribute mechanism. The getattr method is called when the programmer is trying to get the value of a non-existent attribute of an object. This case may be handled by throwing a suitable exception (the default) or using a custom procedure and forwarding the attribute access to another object, for example. Similarly, the setattr mechanism is accessed when the programmer wants to assign a value to a non-existent attribute of an object. Per default, the attribute will be added to the object and assigned the given value.

Comparable to the operator overloading and the attribute handling mechanism, indexing can be customized using the `__getitem__` and the `__setitem__` methods. Taking a dictionary as an example, the getitem method is called when the programmer retrieves a value from the dictionary using a key. The setitem method will be used when the programmer wants to set a key to a specified value in the dictionary.

Finally, it should be explicitly mentioned that methods are also objects and can have double underscore attributes. One ubiquitous example is `__doc__`, which may contain a method's docstring.

Listing 2.9: *Customize addition and printing of objects of class* `A`.

```
class A():
    def __init__(self, value: int = 42) -> None:
        self.value: int = value

    def __add__(self, other: "A") -> "A":
        return A(self.value + other.value)

    def __str__(self) -> str:
        return f"value: {self.value}"

print(A() + A()) # prints "value: 84"
```

3 Design

In this chapter, we will describe an approach for a mapping of pandas operations to MLIR operations and a transparent drop-in replacement for pandas that provides its own `DataFrame` class, as well as implementations for other classes of the pandas API where a `DataFrame` operation requires it.

3.1 Motivation

pandas is a very interoperable and flexible Python library. It can read data from various representations, perform a great variety of operations on it, and export it again to standard data formats such as CSV or JSON. However, pandas is implemented in Python and may not be an appropriate choice when a more decent performance is required. Especially the storage management and the data transfer *between* several operations is not optimized. First, pandas operates on data *in memory*. Therefore, it is not possible to work with intermediate data sets larger than the memory capacity of the executing machine. This is made more problematic by the fact that most operations on `DataFrame` objects return a modified copy. Even in-place operations may work with intermediate copies [39]. As a result, the performance of pandas is not optimal when there is the need to handle large data sets. The pandas developers themselves note that it might be "worth considering *not using pandas*. pandas isn't the right tool for all situations. If you're working with very large data sets and a tool like PostgreSQL fits your needs, then you should probably be using that." [39]

In contrast, modern RDBMSs provide a very optimized data flow and are crafted to handle very large data sets. However, they do not necessarily have an easy-to-use and flexible interface with a wide range of options to interoperate with standard data formats or powerful python libraries such as NumPy or matplotlib.

Thus, this approach aims to provide the pandas API and let a RDBMS do the heavy lifting in the background.

Comparison with SQL

The resemblance of the `DataFrame` class and usual database systems clearly emerges from the comparison of `DataFrame` operations and common SQL operations, as done in Figure 3.1. [39]

Selection. As shown in Figure 3.1a, the selection is made in pandas using indexing. Depending on whether there is only a single column label specified or a list of column labels, the result will either be of type `Series` or `DataFrame`. A conditional selection,

which can be implemented in SQL using the `where` clause, corresponds to the boolean indexing of a `DataFrame`, as demonstrated in Figure 3.1b.

Join. The most general method to join two `DataFrame` objects in pandas is the `merge` method. Although there is an explicitly-called `join` method, the `merge` method is used internally there, too. Figure 3.1c illustrates the usage of the `merge` method. Note that the results are not exactly equivalent. In contrast to SQL, pandas specifies the order of the rows after the join operation. Left outer joins, for example, preserve the key order. Full outer joins sort the keys lexicographically. [39] The parameter *how* can be used to describe the type of the join (*left*, *right*, *outer*, *inner*, and *cross* for the cross product). The parameters *left_on* and *right_on* can be used to specify the columns to join on. Note that the `merge` method only supports equi-joins. Theta-joins may be implemented using a combination of merge and other operations, for example. Figure 3.1d illustrates this. Semi- and Anti-joins can be done in pandas using the `isin` method or an appropriate merge with a subsequent selection, for example.

Aggregation. A pandas `DataFrame` supports aggregation as shown by the example in Figure 3.1e. The `aggregate` method (or its alias `agg`) can accept keyworded arguments denoting a mapping of column labels to the names of aggregation functions such as *sum*, *min* or *count*. The compatibility of pandas and SQL is not completely given with this operation since pandas does not use the new column names as column names of the resulting `DataFrame` but as row labels. The operation in Figure 3.1e would thus yield a 2x2 `DataFrame` where the columns stay the same and the missing values are padded with NaN, which additionally converts the data type of the columns from integer to float.

GroupBy. The `groupby` method of a pandas `DataFrame` supports grouping by one or multiple column labels and returns a `DataFrameGroupBy` object. In contrast to SQL, the order of the rows within the groups is always preserved. By default, the group keys themselves are sorted. In order to prevent the group keys from becoming indices instead of columns and thus lose the SQL-like behavior, we need to set *as_index* to false. [39] We can then call, for example, the aggregation method on the returned `DataFrameGroupBy` object. Figure 3.1f shows a simple example. This works similarly to the aggregation method of a `DataFrame` described above. However, the result format of the aggregation method of the `DataFrameGroupBy` class matches the output format of the SQL query.

3.2 Approach: a LingoDB-based Drop-in Replacement for Pandas

In this chapter, we present our approach for *mlir_pandas*, a transparent drop-in replacement for pandas. We also implemented a prototype as a Python library which can be imported instead of pandas.

mlir_pandas provides its own `DataFrame` class and the auxiliary classes `Series` and `DataFrameGroupBy`. Instead of directly executing the methods called on those objects, the *mlir_pandas* objects incrementally build an MLIR module, which will be compiled

```sql
select col1, col2 from df
```

```python
df[["col1", "col2"]]
```

(a) *Select* col1 *from the data frame.*

```sql
select * from df
where col1 < 5 and col2 + col1 > 6
```

```python
df[(df["col1"] < 5) &
    ((df["col2"] + df["col1"]) > 6)]
```

(b) *Select all records from* df *where the value of* col1 *is greater than* 5 *and the value of* col2 *is smaller than* 3*.*

```sql
select *
from df join df2 where col1 = col3
```

```python
df.merge(df2, how="inner",
    left_on=["col1"],
    right_on=["col3"])
```

(c) *Inner join of* df *and* df2*. Assume* df2 *to have the two columns* col3 *and* col4 *of type integer.*

```sql
select *
from df join df2 where col1 > col3
```

```python
tmp = df.merge(df2, how="cross")
tmp[tmp["col1"] > tmp["col3"]]
```

(d) *Theta-Join of* df *(*col1*,* col2*) and* df2 *(*col3*,* col4*) on the condition that the value in* col1 *is greater than the value in* col3*.*

```sql
select sum(col1) as colX,
    max(col2) as colY from df
```

```python
df.aggregate(colX=("col1", "sum"),
    colY=("col2", "max"))
```

(e) *Calculate the sum of* col1 *and get the maximum element of* col2*.*

```sql
select col1, count(col2) as colX
from df group by col1
```

```python
df.groupby("col1", as_index=False)\
    .aggregate(colX=("col2", "count"))
```

(f) *Group by* col1 *and get the group keys along with the corresponding group size. (The sort order is ignored here.)*

Figure 3.1: *Operations in SQL (left) and their counterparts in Python using pandas (right). Assume the table / data frame to contain the two columns* col1 *and* col2 *of type integer.*

and executed by LingoDB. The materialization will be done lazily. Thus, every supported operation is added to the MLIR module until the data finally needs to be materialized.

We choose MLIR instead of SQL because MLIR is a compiler framework that many other libraries can use. Therefore, it allows for optimization across project boundaries, for example, when working with tensor types. LingoDB provides an implementation of MLIR dialects needed to transpile data frame operations to relational algebra, a well-known and suitable theory of handling manipulations of data frames.

Implementing the pandas `DataFrame` API

The implementation of the pandas `DataFrame` API should be a drop-in replacement for the original API. This means that the API from the point of view of the API user stays the same. API stability is a fundamental property for software acceptance and adoption. [1]

In order to be a drop-in replacement for the pandas `DataFrame` class, the *mlir_pandas* `DataFrame` class needs to provide the same attributes (methods are callable attributes). Additionally, the methods need to handle all the arguments of the original methods, and the return values need to be represented correctly. Since we do not override all methods or attributes, we need to be able to transparently forward attribute accesses to the original pandas implementation. This implies that we have to be able to provide and access an instance of the original `DataFrame` class when needed.

Our primary focus in this work is the `DataFrame` class. We additionally need to implement a wrapper for the `Series` class to be able to handle boolean indexing, for example. Furthermore, we provide our own `DataFrameGroupBy` class so that the result of the *groupby* operation of a `DataFrame` does not need to be materialized but can still be used to execute operations on it lazily.

All those wrapper classes include at least one reference to an object holding the actual data, which needs to be loaded to LingoDB when executing the constructed MLIR module. The data object usually is an instance of the corresponding original pandas class but may also be a pyarrow table. Using a pyarrow table as internal storage has the advantage that the data does not need to be converted when it is loaded or received from LingoDB, which uses pyarrow tables as exchange format. On the other hand, an instance of the original pandas class is needed for the transparent fallback mechanism.

The *mlir_pandas* classes themselves can be regarded as *views* on a data frame. The views are restricted to specific columns of the data frame. Thus, for each object, we store the identifiers of the columns this object is responsible for. This management of *partial* data frames is also supported by the `relalg.materialize` MLIR operation, which does not need to materialize all columns of a tuple stream and can materialize the specified columns only. This enables our approach of constructing multiple different *mlir_pandas* objects based on the same initial tuple stream.

Building an MLIR module

For every new *mlir_pandas* object (e.g. `DataFrame` or `Series`), we create an MLIR function. This function accumulates the supported operations so they can be executed

when the materialized result is needed. In that case, the function is inserted as *main* function into a new MLIR module.

Many operations on a `DataFrame` return a modified copy. However, in this approach, we do not materialize on those occasions. Instead, we create a new instance of the corresponding *mlir_pandas* class which points to the same original data object(s) as the previous instance and continues to build the same MLIR module. However, sharing a reference to a common MLIR module does not imply that the participating *mlir_pandas* objects need to operate on the same MLIR values. They can still follow a different execution path through the MLIR module.

When we need to work with multiple *mlir_pandas* objects, we combine their MLIR modules into a single module again. This new single module receives a new *main* function. The *main* functions of the previous individual modules are copied and inserted as auxiliary functions in the new module. They are called at the beginning of the new *main* function to receive their last state, which is set as their return value. The function calls are later inlined during the optimization phase of the compilation of the MLIR module. This allows the return values of the function calls to have types such as `relalg.tuplestream`, which is an internal concept of LingoDB's `relalg` dialect and would not support materialization. Figure 3.2 shows the layout of two combined `DataFrame` objects. Note that it is crucial to set the symbol visibility [7] of the auxiliary functions to private. Otherwise, it would not be possible to lower them since their return types are only internal concepts of LingoDB that are not defined in a general context.

In order to be able to work with multiple *mlir_pandas* objects, we always provide an internal index column when the data is loaded to LingoDB. Depending on the types of the original data objects, this either is the original index if a data object is a pandas object or a newly generated increasing integer sequence if a data object is a pyarrow table. The internal index column allows joining independent tuplestreams, which would not be possible otherwise since RDBMSs do not guarantee a particular row ordering.

3.3 Mapping Pandas Operations to MLIR

The pandas operations get mapped to MLIR operations lazily and across multiple *mlir_pandas* objects. The *mlir_pandas* classes override supported pandas operations. The MLIR module will only be materialized when needed or when a currently unsupported operation needs to be executed. In this work, we will focus on supporting the most intuitive and convenient methods of a pandas `DataFrame`.

Join

Being able to join two tuple streams on MLIR level is the base requirement for many other operations. We implement the `merge` operation of the `DataFrame` class such that it is possible to join a `DataFrame` and another `DataFrame` or a `Series` on arbitrary columns. For this, we make use of the join operations such as `relalg.join` defined by LingoDB's `relalg` dialect. Since the pandas `merge` method only supports equi-joins, the comparison of the specified columns will always be a comparison for equality, performed by `db.compare`. The column values are received using `relalg.getcol`. A simple example

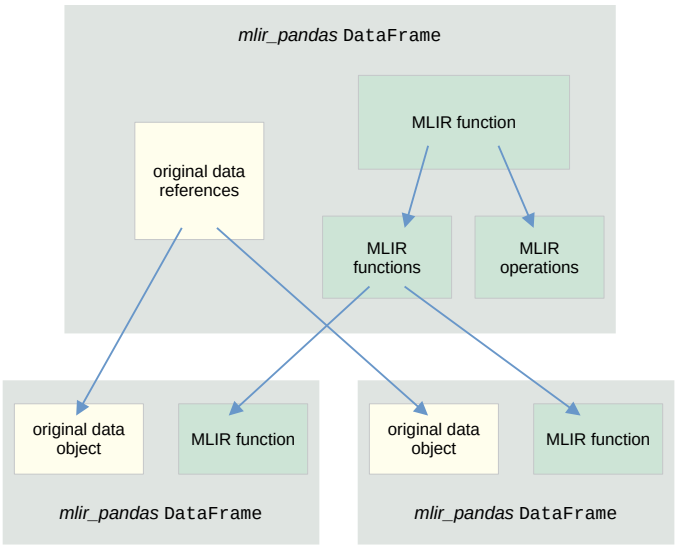


Figure 3.2: *A schema of an* mlir_pandas ***DataFrame*** *which emerged from the combination of two other* mlir_pandas ***DataFrame*** *objects. The new* ***DataFrame*** *references the original data objects of the previous* ***DataFrame*** *objects and also copied their MLIR functions.*

is shown in Figure 3.3. Our implementation currently does not handle naming collisions, but they can be trivially circumvented with a rename operation.

If an outer join needs to be accomplished, we add a sort operation to comply with the pandas API, which requires the keys to be sorted after the operation.

Indexing / Selection

Indexing is implemented in pandas by the `__getitem__` method, which is a special method of Python classes and will be called when the corresponding object is subscripted. Indexing a `DataFrame` is similar to a selection in SQL, as shown in Figure 3.1. Our approach currently handles two cases. The simple case is that the `__getitem__` method is called with a column label or a list of column labels. Supporting this operation does not require any MLIR operations. We need to copy the existing *mlir_pandas* object to a new one which is a view on the same tuple stream but restricted to the specified columns. With this, we implement a `Series` which is a (restricted) *view* on the given `DataFrame`. In the second case, the `__getitem__` method is called with a boolean `Series` as argument. Consequently, boolean indexing needs to be performed. This is a selection where the boolean values in the provided `Series` form the condition of the selection. In order to be able to select the corresponding rows of the original data frame, we may first need to join the data frame and the `Series` in case that the `Series` did not originate from this `DataFrame` and is a completely independent object. The join happens using an inner join on the internal index column. Afterwards, we use `relalg.selection` with a predicate computing the truth value of the `Series` for every row in the tuple stream. Figure 3.4 shows an example of boolean indexing.

```
%0 = relalg.basetable @df1 {columns = [index, col1, col2, col3]}
%1 = relalg.basetable @df2 {columns = [index, col4, col5, col6]}
%2 = relalg.join %0, %1 {
    ^bb0 (%arg_row: !relalg.tuple):
        %3 = relalg.getcol %arg_row @df1::col1
        %4 = relalg.getcol %arg_row @df1::col2
        %5 = relalg.getcol %arg_row @df2::col4
        %6 = relalg.getcol %arg_row @df2::col5
        %7 = db.compare eq %3, %5
        %8 = db.compare eq %4, %6
        %9 = db.and %8, %9
        relalg.return %9
}
```

(a) *Pseudo-MLIR code.*

```
df1 = pandas.DataFrame({'col1': ..., 'col2': ..., 'col3': ...})
df2 = pandas.DataFrame({'col4': ..., 'col5': ..., 'col6': ...})
df1.merge(df2, how='inner', left_on=['col1', 'col2'], right_on=['col4', 'col5'])
```

(b) *Corresponding pandas pseudo-code.*

Figure 3.3: *Comparison of pseudo-MLIR code and pandas pseudo-code for joining two* `DataFrame` *objects on the predicate* `col1 == col4` *and* `col2 == col5`.

```
%0 = relalg.basetable @df {columns = [index, col1, col2]}
%1 = relalg.basetable @series {columns = [index, value]}
%2 = relalg.join %0, %1 on @df::index = @series::index
%3 = relalg.selection %2 where @series::value = true
```

(a) *Pseudo-MLIR code.*

```
df = pandas.DataFrame({'col1': ..., 'col2': ...})
series = pandas.Series([True, False, ...], name='value')
df[series]
```

(b) *Corresponding pandas pseudo-code.*

Figure 3.4: *Comparison of pseudo-MLIR code and pandas pseudo-code for boolean indexing using a* `Series` *object which does not originate from the same* `DataFrame`*, so which is not already included in the same tuple stream.*

```
%0 = relalg.basetable @series {columns = [index, value]}
%1 = relalg.map %0 {
    %2 = db.constant 5
    %3 = relalg.getcol @series::value
    %4 = db.compare gte %3, %2
    relalg.return %4
}
```

(a) *Pseudo-MLIR code.*

```
series = pandas.Series([True, False, ...], name='value')
series >= 5
```

(b) *Corresponding pandas pseudo-code.*

Figure 3.5: *Comparison of pseudo-MLIR code and pandas pseudo-code for comparing a* `Series` *to the constant* 5.

Arithmetic Operations

All arithmetic operations will result in a `relalg.map` operation which adds a new column to an existing `DataFrame`. The return value of the arithmetic operations described here will thus always be a new `Series` object, which is a *view* restricted to this new column in the original `DataFrame`.

In order to create boolean `Series` objects, we need to be able to compare a `Series` with another `Series` or a constant. We implement the corresponding special Python methods such as `__ge__` for the greater-equal comparison. In case a `Series` should be compared to another `Series`, the two tuple streams will be joined if the two `Series` objects do not originate from the same `DataFrame` and are not already contained in the same tuple stream. Afterwards, a `relalg.map` operation will be performed to execute the specified comparison on the two columns belonging to the two participating `Series` instances. In case a `Series` should be compared to a constant, such as illustrated by Figure 3.5, it is possible to directly apply the `relalg.map` operation. In both cases, the region attached to `relalg.map` will use the `db.compare` operation in order to compare the `Series` values per row, or the `Series` values with the given constant respectively. In both cases, a new column holding the comparison result gets added to the tuple stream. Thus, as a final step, we create a new `Series` object that is a view of the new column and can be returned as the result of the comparison.

Furthermore, we can operate on boolean `Series` using other boolean operators. The bit-wise *and*, the bit-wise *or* and the bit-wise negation are implemented by overriding the corresponding special Python methods `__and__`, `__or__` and `__invert__`. They are implemented in the same way as already described. The only difference is that we use the corresponding MLIR operation, so `db.and`, `db.or` and `db.not` instead of `db.compare`.

For non-boolean arithmetic operations such as addition of a `Series` and another `Series` or a constant, we use the same approach with appropriate arithmetic operation such as `relalg.add` for addition or `relalg.mul` for multiplication.

```
%0 = relalg.basetable @df {columns = [index, col1, col2]}
%1 = relalg.aggregation %0 {group_by_cols =[], computed_cols  = [colX, colY]} {
    %2 = relalg.aggrfn sum @df::col1
    %3 = relalg.aggrfn min @df::col2
    relalg.return %2, %3
}
```

(a) *Pseudo-MLIR code. The* `group_by_cols` *attribute is empty since there is no grouping in this example.*

```
df = pandas.DataFrame({'col1': ..., 'col2': ...})
df.aggregate(colX=('col1', 'sum'), colY=('col2', 'min'))
```

(b) *Corresponding pandas pseudo-code.*

Figure 3.6: *Comparison of pseudo-MLIR code and pandas pseudo-code for an aggregation of a* `DataFrame`*. The operation computes the sum for* col1 *and the minimum for* col2*.*

Aggregation

The aggregation on a pandas `DataFrame` can be done using the `aggregate` method, or its alias `agg`. They can take keyworded arguments which map new column labels to old column labels and aggregation functions such as *sum*, *min* or *count*. For the *mlir_pandas* classes, the aggregation directly translates to the `relalg.aggregation` MLIR operation. This operation takes a region where we can execute the `relalg.aggrfn` MLIR operation on the specified columns with the corresponding aggregation function, as implemented in Figure 3.6. The results returned by the region are stored as the attributes of the resulting tuple stream.

After the aggregation, we have to materialize, no matter if the aggregation has been called on a `Series`, `DataFrame` or `DataFrameGroupBy` object. In order to comply with the particular output format of the corresponding original pandas implementations, it is always necessary to post-process the result computed by LingoDB and convert it to the correct shape using the original pandas methods. In the case of a `DataFrameGroupBy` object, the shape of the result returned by LingoDB would match the pandas implementation. However, we still need to materialize because the internal index column got lost during the aggregation.

GroupBy

pandas provides the `groupby` method for `DataFrame` objects. This operation returns a `DataFrameGroupBy` object. *mlir_pandas* provides an own `DataFrameGroupBy` class, which is, as the other *mlir_pandas* classes, just a view of a tuple stream. Additionally, an instance of the `DataFrameGroupBy` class stores the columns which should be used for grouping in the user-specified order. Thus, when the `groupby` method is called on an *mlir_pandas* `DataFrame` object, no MLIR operation is performed. It just returns a properly configured *mlir_pandas* `DataFrameGroupBy` instance. On this instance, the API user can perform aggregation methods, for example. They are implemented as shown in Figure 3.6, the

only difference is that the attribute `group_by_cols` of the `relalg.aggregation` operation additionally receives the list of group keys in the correct order.

Since the group keys get sorted after the aggregation in pandas, we also save whether a sort operation should be executed on the group keys after the aggregation.

3.4 Handling Pandas Peculiarities

The approach proposed in this work aims to strictly follow the pandas API so that the created *mlir_pandas* module can be used as a proper transparent drop-in replacement for pandas without requiring the user to change their code or their understanding of the API usage. This design decision leads to challenges, which we will outline in the following subsections.

Row ordering

As already discussed, pandas has a row ordering which is specified after most operations. This leads to the necessity of adding an internal index column in our implementation of *mlir_pandas* every time data is loaded to LingoDB. If the data is a pandas `DataFrame`, we use its index as the index column. The index of a `DataFrame` object can hold objects of any type, and they do not need to be unique, which we currently ignore in our prototype. If we work on a pyarrow table as a data object, we add a new index column by creating a sequence of unique integer ids of increasing value. The internal index column becomes necessary when the columns of two *mlir_pandas* objects need to be merged into one tuple stream, for example. This would not be possible otherwise, even if LingoDB had a guaranteed row ordering.

Additionally, the original indices of the `DataFrame` objects need to be kept so that they can be restored after the materialized result of the MLIR module has been received from LingoDB. Adding a new index would violate the compatibility with the pandas API since the index values are bound to the rows. Thus, the index is changed by a selection, for example, as shown in Listing 3.1.

Finally, to guarantee row ordering, we need to sort the data according to the index column before each materialization. Currently, our prototype for *mlir_pandas* does not implement this and still relies on the fact that the current LingoDB implementation *has* guaranteed row ordering since it does not yet implement multithreading.

In-place methods

Some pandas `DataFrame` API methods can optionally be executed with in-place semantics. This can be achieved in *mlir_pandas* by adding the necessary operations to the MLIR module of the current *mlir_pandas* object instead of a copy.

However, there are cases where the pandas API does not define whether an operation is performed in-place or not. A prominent example is indexing. [39] Even for a simple case such as selecting a single column from a `DataFrame` *df*, it is not specified whether a *view* or a *copy* of the corresponding part of the `DataFrame` is returned. For the implementation of *mlir_pandas*, we always worked with copy semantics.

Listing 3.1: *The index of a pandas* `DataFrame` *is updated by a selection. The selection removed the second row, and as a result, the corresponding index value* 1 *is removed, too.*

```
>>> df = pd.DataFrame({"col1": [1,2,3], "col2": [1,1,3]})
>>> df
   col1  col2
0     1     1
1     2     1
2     3     3
>>> df[df["col1"] <= df["col2"]]
   col1  col2
0     1     1
2     3     3
```

Boolean Indexing

Conditional selection is implemented in the pandas API using boolean selection, as already described. For our approach, it is a difference whether the boolean `Series` used to index the current `DataFrame` originated from the same `DataFrame` or is completely independent. In the latter case, we first need to merge the columns of the current `DataFrame` and the `Series` by performing a join operation on the index columns. If the `Series` is not independent, it originated from an arithmetic operation on the `DataFrame` and thus is just a column in its tuple stream. In that case, it is possible to directly perform the selection based on the value in the corresponding column.

3.5 Implementation Details

We use the composite pattern to implement the classes of the pandas library which we override in our *mlir_pandas* library. Every class works as a wrapper around the original pandas class. This allows us to incrementally override methods without losing the full functionality of the original class. Additionally, we leverage the *getattr*/*setattr* mechanism of Python. As soon as the user of our class tries to access a method or an attribute that we do not yet directly support, we catch the failed attribute access in the `__getattr__` method, materialize the instance of the original class, update it by executing the lazily accumulated MLIR operations, and then forward the attribute access to the resulting object.

The results of attribute accesses or operations are handled by a separate mechanism in *mlir_pandas*, which wraps objects of classes that have a counterpart in *mlir_pandas* in an instance of the corresponding *mlir_pandas* class. This way, we ensure that we do not accidentally leave our class hierarchy and unintentionally fall back to the classes of the pandas API.

Finally, we need to provide a reasonable integration of our custom classes with standard development tools such as autocompletion functionality in Integrated Development Environments (IDEs) or static analysis frameworks. Since we do not override *all* attributes

of the original classes and may not provide the same docstrings or method signatures, those tools would not be able to assist the API user correctly. In order to solve this, we leverage the constant `TYPE_CHECKING` which is defined by the `typing` module of the Python standard library. This constant indicates whether the code is currently analyzed by static analysis tools. If this is the case, *mlir_pandas* dynamically chooses not to export its custom classes but their original counterparts of the pandas library. Functionality-wise, this does not make any difference since it does not affect the execution during runtime.

4 Evaluation

We evaluate the performance of *mlir_pandas* against pandas and the direct execution of an already completed MLIR module in LingoDB. We use some of the well-known TPC-H queries for the benchmarks. As an intermediate storage format for the tables which need to be used in the queries, we use pyarrow tables [10]. They provide easy-to-use support for converting from and to pandas, and they are the data exchange format currently used by LingoDB's Python interface.

4.1 Targets

The benchmarks are implemented in Python. Every benchmark is executed as a function in a separate process. The benchmark for the pre-built MLIR module passes this module directly to LingoDB. The benchmarks for pandas and *mlir_pandas* execute the same function containing the query implemented with the pandas API. The pandas operations used to implement the SQL queries for pandas are selected so that they are completely supported by the current implementation of *mlir_pandas* and do not require the fallback to the original pandas implementation. The initial data is a pyarrow table which is completely read into memory. For the pandas and *mlir_pandas* targets, the columns of the table are casted so that their types can be represented as explicit pandas types and not just as the generic object type. This implies, for example, that decimals need to be converted to a 64-bit floating point type since pandas currently has no built-in decimal type.

MLIR The MLIR module for this benchmark is created by translating the corresponding SQL query using LingoDB's `sql-to-mlir` tool and inserting it into the Python code as string literal which can then directly be loaded to LingoDB. The result is converted into a pandas `DataFrame`. If necessary, additional computations are executed on this `DataFrame`, for example, when the result is a table / data frame with a single value and we need to extract this single value in order to return it as a plain scalar. All additional steps are included in the performance measurement.

pandas Before the benchmark for pandas is executed, the given pyarrow table is casted to ensure that all types are directly supported by pandas. Then, the table is converted to a pandas `DataFrame`. This `DataFrame` is passed to the function that will be measured.

mlir_pandas We benchmark *mlir_pandas* with the exact same query code as pandas. However, *mlir_pandas* can operate directly on the casted pyarrow table. *mlir_pandas* can use the pyarrow table internally as data instead of a pandas object. The internal

data representation is only changed to a pandas `DataFrame` if this is required for the transparent fallback mechanism.

4.2 Setup

Our performance benchmarks are executed on a cloud server with eight dedicated vCPUs (4 Intel Xeon at 2 threads per core) on a KVM-based, fully virtualized x86-64 platform running Fedora 36 with Linux Kernel 5.18.11, GCC 12.1.1, and Python 3.10.6. LingoDB is used with commit `91c529d`[1] and LLVM commit `471cc82`[2]. pandas is used in version 1.4.2.

4.3 Results

Table 4.1 shows the results of our benchmarks. Figure 4.1a and Figure 4.1b additionally display the results for scale factor 2 in chart form. *mlir_pandas* manages to provide a speedup compared to pandas by a factor of up to around 14 and over 10 on average. The additional management and conversion overhead of *mlir_pandas* over the pure runtime of LingoDB is up to 0.29 seconds or up to 47 percent, with around 0.13 seconds or 9 percent on average. The pure runtime of LingoDB for the queries submitted by *mlir_pandas* is on average approximately twice as slow as the pure runtime of LingoDB for the original MLIR queries.

The difference regarding the pure runtime of LingoDB between *mlir_pandas* and the pre-computed original MLIR are most probably a result of the incremental building process of the MLIR module within *mlir_pandas*. When the original SQL query is translated to MLIR by the `sql-to-mlir` tool, the whole query is known. For query 5, where the difference is most noticeable, the original MLIR module uses, among other operations, a single selection after a series of cross products, which get optimized away, followed by one map operation. The MLIR module produced by *mlir_pandas* contains five joins, three selections and eight map operations. This is due to the fact that *mlir_pandas* cannot know the whole query in advance and is currently naively building the MLIR module incrementally while encountering the pandas operations. Another factor for the significant difference between the LingoDB runtime values of *mlir_pandas* and the original MLIR module for query 5 is that *mlir_pandas* currently needs to perform an intermediate materialization after the aggregation and before the final sorting.

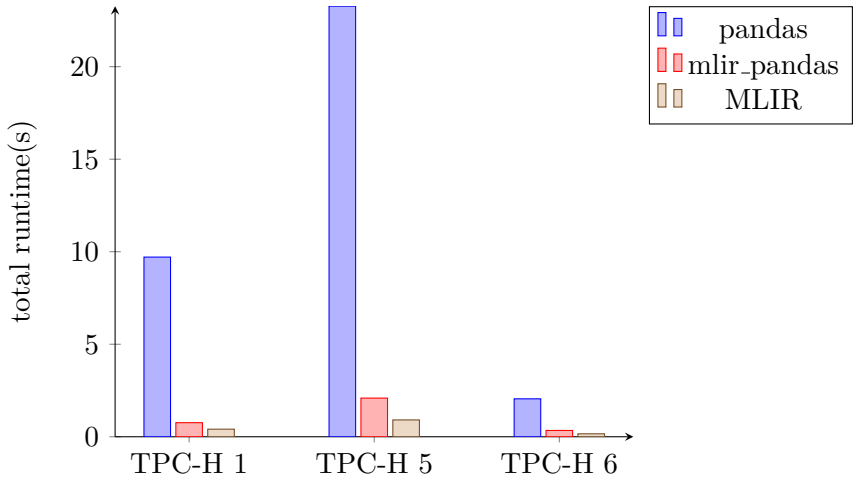
The management overhead of *mlir_pandas*, so the difference between the total runtime and the runtime of LingoDB added with the time for the conversions, takes on average around 7 percent of the total runtime. In absolute values, it is up to 0.07 seconds or about 0.05 seconds on average. Most importantly, the management overhead of *mlir_pandas* seems to be constant and does not depend on the scale factor of the input tables.

Finally, the memory overhead of pandas, resulting from the generation of intermediate data copies, could be drastically reduced by *mlir_pandas*. The maximum memory consumed by *mlir_pandas* is on average approximately 44 percent of the peak memory consumption

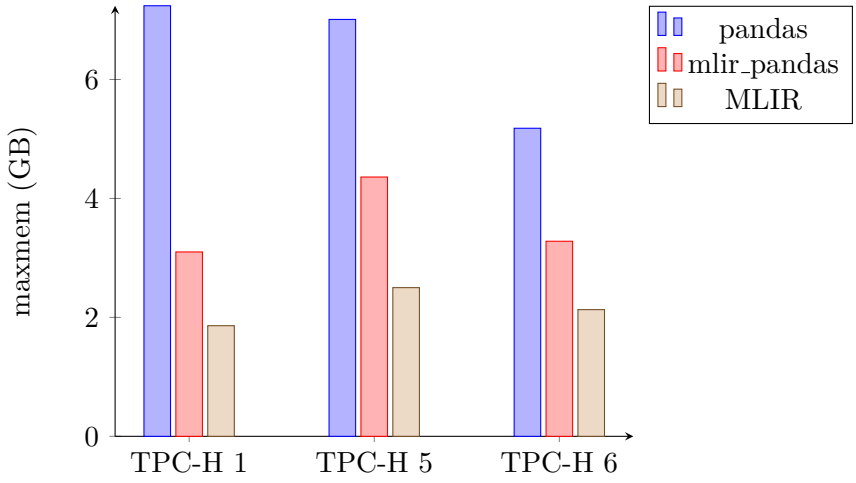
[1] `https://github.com/ro-i/lingo-db/tree/91c529d13bc46a5b8cc60c42618d14f328db1fb1`

[2] `https://github.com/jungmair/llvm-project/tree/471cc823031b8b0950be4b6e2245e63eafcfcaf3`

of pandas. Executing the original MLIR module still requires less memory, which seems reasonable, considering that the MLIR module built by *mlir_pandas* still does not achieve the same level of optimization as the original MLIR module.



(a) *Total runtime of pandas,* mlir_pandas *and a pre-built MLIR module on the TPC-H queries 1, 5 and 6 for scale factor 2.*



(b) *Peak memory consumption of pandas,* mlir_pandas *and a pre-built MLIR module running the TPC-H queries 1, 5 and 6 for scale factor 2.*

Table 4.1: *Results of TPC-H Queries 1, 5 and 6 with scale factor* 1, 2 *and* 5*, executed by pandas,* mlir_pandas *and an MLIR module pre-built by translating the original SQL query using LingoDB's* `sql-to-mlir` *tool.* total duration *denotes the total runtime of the respective function;* rundb *shows the runtime of LingoDB;* conversions *report the accumulated duration of all necessary conversion between pyarrow tables and data frames, as well as the time taken by loading the given tables to LingoDB;* maxmem *shows the maximum amount of memory used by the subprocess which executed the corresponding benchmark function. If* mlir_pandas *needed to materialize more than once,* rundb *and* conversions *are the summed up values of all materializations.*

(a) *Results for scale factor* 1.

Benchmark	Target	Measurement			
		total duration (s)	rundb (s)	conversions (s)	maxmem (GB)
TPC-H 1	pandas	5.19	—	—	3.66
	mlir_pandas	0.49	0.41	0.05	1.61
	MLIR	0.24	0.22	0.02	0.97
TPC-H 5	pandas	10.14	—	—	3.63
	mlir_pandas	1.05	0.96	0.04	2.16
	MLIR	1.34	1.33	0.01	1.63
TPC-H 6	pandas	1.02	—	—	2.49
	mlir_pandas	0.17	0.11	0.03	1.63
	MLIR	0.09	0.09	0.01	1.01

(b) *Results for scale factor* 2.

Benchmark	Target	Measurement			
		total duration (s)	rundb (s)	conversions (s)	maxmem (GB)
TPC-H 1	pandas	9.71	—	—	7.24
	mlir_pandas	0.76	0.66	0.06	3.10
	MLIR	0.41	0.39	0.02	1.86
TPC-H 5	pandas	23.27	—	—	7.01
	mlir_pandas	2.09	1.98	0.06	4.36
	MLIR	0.91	0.90	0.01	2.50
TPC-H 6	pandas	2.05	—	—	5.18
	mlir_pandas	0.34	0.18	0.11	3.28
	MLIR	0.16	0.15	0.01	2.13

(c) *Results for scale factor* 5.

Benchmark	Target	Measurement			
		total duration (s)	rundb (s)	conversions (s)	maxmem (GB)
TPC-H 1	pandas	23.65	—	—	17.97
	mlir_pandas	1.62	1.48	0.10	7.56
	MLIR	0.78	0.76	0.01	4.47
TPC-H 5	pandas	69.36	—	—	17.40
	mlir_pandas	6.48	6.30	0.11	11.11
	MLIR	1.34	1.33	0.01	5.65
TPC-H 6	pandas	4.71	—	—	12.67
	mlir_pandas	0.70	0.41	0.25	7.62
	MLIR	0.33	0.33	0.01	4.76

4.4 Discussion

The measurements show that our approach can accelerate pandas code considerably while not sacrificing the extensive and easy-to-use pandas API. Even for queries such as query 5, where currently an intermediate materialization is necessary, we can record a speedup by the factor of 10. In absolute values, this becomes increasingly valuable for larger data sets, as shown in Table 4.2c, where up to around 63 seconds could be saved for query 5.

However, the benchmarks also show that there is still more potential for optimization since we do not yet achieve comparable performance to the original MLIR module derived from the SQL query.

5 Related Work

There are many approaches which try to enhance the performance of pandas by transpiling the pandas operation to other representations. Some leverage MLIR as intermediate representation not exclusively for pandas, but also for other libraries with the goal to enable cross-library optimizations.

Among the general differences of the various approaches, one of the most important is to which language or representation the original operations are transpiled, so for example to MLIR or to SQL. Furthermore, from the performance point of view, we need to consider whether the queries are evaluated (mostly) lazily, or in a direct way. Finally, it should be distinguished between approaches with the possibility of remote execution and approaches which require the data on the machine which executes the program. Especially for the second case, it is convenient if the approach supports data larger than the memory capacity of the executing machine.

Weld Weld [25, 27, 26] aims for cross-library optimization by collecting operations from various libraries written in different languages and transpiling them to a common intermediate representation, the Weld IR. The IR module is then compiled, optimized and executed when a materialization becomes necessary. It is important to note that Weld is also a runtime environment. In contrast to our approach, Weld does not make use of an existing runtime, such LingoDB in our case, but directly defines and implements the necessary IR operations and data types and the optimizations on them itself. As a result, the system is not limited to the interface of a particular runtime implementation and can flexibly adapt to different scenarios on different abstraction levels. On the other hand, this implies the effort to support new libraries since Weld may have to adapt their runtime implementation, too: "Porting whole libraries to use Weld is time-consuming, so we evaluate the impact of porting just a few widely used operators." [26]

Grizzly Grizzly [16] transpiles pandas operations to SQL statements which are then executed in an external DBMS. The pandas operations are collected lazily, they are only materialized when needed. Grizzly can also handle User-Defined Functions (UDF) by applying Python reflection techniques and leveraging Python type hints to deduce the data types of the actually dynamically typed language. Additionally, Grizzly distinguishes "Data Shipping" and "Query Shipping". While pandas has a very powerful API, it requires the data to reside on the machine where the pandas code is executed. In contrast, DBMS are known for being hosted on servers which are independent of the machine where the SQL statements are run. Grizzly enables the combination of both paradigms and allows the programmer to write pandas and execute pandas code on their machines while the generated SQL statements are run in potentially remote DBMS.

mlinspect mlinspect [14, 13, 12] is a library which analyzes existing data preprocessing code for Machine Learning (ML) pipelines. This code is often written using pandas. mlinspect transforms the Python operations to an IR which is then used to build a Directed Acyclic Graph (DAG) in which the operations are mapped to relational algebra operators. mlinspect acts similar to a debugger. It instruments the original Python code with the collected information and metadata and thus allows to find so-called "data distribution bugs", so issues related to the flow and the handling of the data in the preprocessing pipeline. Thus, mlinspect needs the similar translation techniques as our approach, but does not aim to be a more performant drop-in replacement for the Python implementation of the pandas API.

Modin Modin [29, 28, 30] aims to be a drop-in replacement for pandas. It is designed especially for large data sets, but claims to also improve performance up to a factor of 4 on a laptop with 4 cores. [8] Modin currently makes use of Ray [35] or Dask [3] in order to parallelize the workload and be able to distribute it over a multiple machines / a cluster. Therefore, an important characteristic of Modin is the *decomposition* of data frames such that their computation can be distributed. The developers solve this by defining the data frame operations with a set of core operators, which are designed and implemented to be parallelized. [28] In contrast to our work, Modin does not make use of existing RDBMS and also does not aim to be a RDBMS itself. The creators of Modin do not consider the relational algebra model to be suitable to handle the pandas API. They think that the relational algebra is too restricted to particular operators and lacks the required flexibility. Additionally, they identify the unspecified row ordering as an important deviation of the relational algebra model from the data frame model. [30]

6 Summary & Outlook

As the applications of *big data* have been considerably growing for many years, efficient DBMSs have been developed and improved. For many use cases, however, data scientists choose the Python library pandas as an easy-to-use way of working with their data. It provides a rich set of interoperability and convenience methods. However, while pandas has found widespread use, it suffers from the performance of the Python implementation, especially regarding large data sets. On the side of the modern compiling DBMSs, LingoDB emerged. LingoDB provides not only an interface for SQL, but also for queries written in MLIR. MLIR is a new IR under the umbrella of the LLVM project and designed to span multiple abstraction levels, providing support for cross-domain compiling and optimizing.

We presented *mlir_pandas*, an approach aiming to combine the advantages of pandas and modern DBMSs. *mlir_pandas* is designed to be a transparent drop-in replacement for pandas, translating the pandas operations to MLIR operations and lazily materializing the results when necessary.

Our implemented prototype showed a performance gain by a factor of 10 on average compared to pandas when executing some of the standard TPC-H queries. We could also show how the maximum memory consumption could be considerably decreased by not working with intermediate copies of the data—in contrast to pandas.

6.1 Outlook

While the performance of *mlir_pandas* showed the potential of our approach, we could also identify possibilities for further improvement.

On the one hand, this may include optimizing the incremental building process of the internal MLIR module.

Furthermore, the pandas types system needs to be enhanced such that decimals, for example, are natively supported and not just represented as generic objects. This would allow *mlir_pandas* to handle those types, which in principle would already be supportable.

Finally, an approach for supporting UDF and the analysis of given expressions for `DataFrame` methods such as `query` or `filter` would help to further improve the performance. Once able to build an expression tree for given string arguments, for example, the `query` method would allow to easily perform a selection according to multiple predicates at once.

Acronyms

CSE Common Subexpression Elimination.

DAG Directed Acyclic Graph.

DBMS database management system.

IDE Integrated Development Environment.

IR Intermediate Representation.

ML Machine Learning.

MLIR Multi-Level Intermediate Representation.

RDBMS relational database management system.

SQL Structured Query Language.

UDF User-Defined Functions.

List of Figures

List of Tables

Listings

Bibliography

[1] Jonathan Corbet. *On breaking things*. Nov. 2010. URL: https://lwn.net/Articles/416821/ (accessed 2022-07-24).

[2] NVIDIA Corporation. *What is pandas Python? — Data Science — NVIDIA Glossary*. URL: https://www.nvidia.com/en-us/glossary/data-science/pandas-python/ (accessed 2022-05-22).

[3] Dask core developers. *Dask — Scale the Python tools you love*. URL: https://www.dask.org/ (accessed 2022-08-11).

[4] The MLIR developers. *Dialects - MLIR*. URL: https://mlir.llvm.org/docs/Dialects/ (accessed 2022-07-15).

[5] The MLIR developers. *MLIR Language Reference - MLIR*. URL: https://mlir.llvm.org/docs/LangRef/ (accessed 2022-06-05).

[6] The MLIR developers. *MLIR Rationale — Integer signedness semantics - MLIR*. URL: https://mlir.llvm.org/docs/Rationale/Rationale/#integer-signedness-semantics (accessed 2022-07-03).

[7] The MLIR developers. *Symbols and Symbol Tables - MLIR*. URL: https://mlir.llvm.org/docs/SymbolsAndSymbolTables/#symbol-visibility (accessed 2022-08-14).

[8] The Modin developers. *GitHub - modin-project/modin: Faster pandas, even on your laptop*. URL: https://github.com/modin-project/modin/tree/3f985ed6864cc1b5b587094d75ca5b2695e4139f#faster-pandas-even-on-your-laptop (accessed 2022-08-11).

[9] The NumPy Developers. *Scalars — NumPy v1.23 Manual*. URL: https://numpy.org/doc/stable/reference/arrays.scalars.html#arrays-scalars-built-in (accessed 2022-07-02).

[10] Apache Software Foundation. *pyarrow.Table — Apache Arrow v9.0.0*. URL: https://arrow.apache.org/docs/python/generated/pyarrow.Table.html (accessed 2022-08-07).

[11] Python Software Foundation. *Python (Website)*. URL: https://www.python.org/ (accessed 2022-06-04).

[12] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. "Data distribution debugging in machine learning pipelines". In: *The VLDB Journal* (Jan. 2022).

[13] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. "MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. 2021, pp. 2736–2739.

[14] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. In: CIDR, 2021.

[15] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 14: 7.5. Sorting Rows (ORDER BY)*. URL: `https://www.postgresql.org/docs/current/queries-order.html` (accessed 2022-07-03).

[16] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. "Putting Pandas in a Box". In: CIDR, Jan. 2021.

[17] Michael Jungmair, André Kohn, and Jana Giceva. "Designing an Open Framework for Query Optimization and Compilation". In: vol. 15. 11. PVLDB, 2022.

[18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask". In: *Proc. VLDB Endow.* 11.13 (Sept. 2018), pp. 2209–2222.

[19] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-Level Language". In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 853–864.

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14.

[21] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. 2010, pp. 56–61.

[22] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550.

[23] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance". In: CIDR, 2020.

[24] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. 2015, pp. 677–689.

[25] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. *Weld: Rethinking the Interface Between Data-Intensive Applications*. 2017.

[26] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. "Evaluating End-to-End Optimization for Data Analytics Applications in Weld". In: *Proc. VLDB Endow.* 11.9 (May 2018), pp. 1002–1015.

[27] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. "Weld: A Common Runtime for High Performance Data Analytics". In: CIDR, 2017.

[28] Devin Petersohn. "Dataframe Systems: Theory, Architecture, and Implementation". PhD thesis. University of California, Berkeley, Aug. 2021.

[29] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. "Towards Scalable Dataframe Systems". In: vol. 13. 11. PVLDB, 2020, pp. 2033–2046.

[30] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. "Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System". In: *Proc. VLDB Endow.* 15.3 (Nov. 2021), pp. 739–751.

[31] The LLVM project. *The LLVM Compiler Infrastructure Project.* URL: `https://llvm.org/` (accessed 2022-06-05).

[32] The Mypy project. *mypy - Optional Static Typing for Python.* URL: `http://mypy-lang.org/` (accessed 2022-07-03).

[33] The NumPy project. *NumPy.* URL: `https://numpy.org/` (accessed 2022-06-25).

[34] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. "DB2 with BLU Acceleration: So Much More than Just a Column Store". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1080–1091.

[35] Ray. *Productionizing and scaling Python ML workloads simply — Ray.* URL: `https://www.ray.io/` (accessed 2022-08-11).

[36] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints — peps.python.org.* URL: `https://peps.python.org/pep-0484/` (accessed 2022-07-03).

[37] Guido van Rossum and Ivan Levkivskyi. *PEP 483 – The Theory of Type Hints — peps.python.org.* URL: `https://peps.python.org/pep-0483/` (accessed 2022-07-03).

[38] The pandas development team. *pandas - Python Data Analysis Library.* URL: `https://pandas.pydata.org/` (accessed 2022-08-11).

[39] The pandas development team. *pandas documentation — pandas 1.4.3 documentation.* URL: `https://pandas.pydata.org/docs/` (accessed 2022-06-25).

[40] The pandas development team. *pandas-dev/pandas: Pandas 1.4.2*. Version v1.4.2. Apr. 2022.

[41] The Matplotlib development team. *Matplotlib — Visualization with Python*. URL: `https://matplotlib.org/` (accessed 2022-06-25).