PySpark at Bare-Metal Speed

Master Thesis
Sabir Akhadov
October, 2017

Advisors: Dr. Ingo Müller, Prof. Gustavo Alonso
Department of Computer Science, ETH Zürich
Abstract

Data analytics has become the driving force for many industries and scientific research. More and more decisions are made based on statistical analysis of large datasets and machine learning. Big data data processing frameworks, such as Apache Spark, provide an easy-to-use out-of-the-box solution, scalable to large machine clusters. Python is the most widespread programming language in the data science field due to its simplicity and the abundance of analytical tools developed for it. Many Spark users would prefer its Python frontend in their daily work. Multiple studies indicate, however, that there is a wide gap between Spark’s performance and the best handwritten code. With this thesis we bring the functional data-flow programs’ performance closer to the bare-metal speeds and show that it is possible to write productively high performance code.
# Contents

## 1 Introduction

## 2 Background

2.1 RDBMS

2.2 Dataflow functional programming

2.2.1 Operators

2.2.2 Execution

2.3 User-defined functions

2.4 Dataflow frameworks

2.4.1 Single machine

2.4.2 Machine cluster

## 3 System Overview

3.1 Architecture

3.1.1 Pure python

3.1.2 C++ backend

3.2 UDF compilation

3.3 UDF static analysis

3.3.1 Data dependencies

3.3.2 Field properties

3.4 Code generation

## 4 Measurements

4.1 Experiment setup

4.2 Built-in operators

4.2.1 Single operator

4.2.2 Operators chain

4.2.3 Compile time
## Contents

4.3 User-defined functions ........................................... 26  
  4.3.1 Minimal salary per department ................................. 27  
4.4 Use cases ............................................................. 29  
  4.4.1 K-Means ........................................................... 30  
  4.4.2 Crime prediction .................................................. 30  

5 Related Work .......................................................... 33  
  5.1 Just-in-time query compilation .................................... 33  
  5.2 Static UDF analysis ................................................. 35  
    5.2.1 Semantics ....................................................... 35  
    5.2.2 Execution cost ................................................. 36  
  5.3 Lazy evaluation and staged programming ....................... 36  

6 Conclusion .............................................................. 39  

Bibliography .............................................................. 41
Chapter 1

Introduction

Over the last few years, data analytics has become an essential part of scientific and business development. Research in different areas such as biology [19] and healthcare [33] relies on analyzing large datasets. Retail companies base their decisions on Big Data [15]. The surge of Internet of Things is also responsible for large amounts of sensor data which has to be processed efficiently [14]. All this calls for new, efficient tools for data processing.

As a response to these developments, more and more tools have emerged for data analytics. The Hadoop ecosystem for data storage and processing provides an easy, out-of-the-box solution for data analytics. Hadoop’s MapReduce is one of the first big data processing frameworks which could scale out to large clusters. Apache Spark is a state-of-the-art big data framework which gained tremendous success due to large in-memory performance improvements compared to MapReduce and its simple functional user interface. Dataflow programming with Spark gives the users more control over the computation compared to the traditional SQL frameworks. At the same time the explicit data dependencies expressed by the users in form of a directed acyclic graph allow the query engine to distribute the computations over a large compute cluster. Users can write Spark programs in multiple programming languages, such as Scala, R and Python.

In a 2016 Spark survey [10], 76% and 91% of the responders marked the ease of programming and the performance to be the primary features they are looking for in a big data framework. However, multiple studies have shown that Spark’s execution times are far below the bare-metal performance [18]. Spark’s primary performance bottleneck is the inefficient query computation on the CPU [28]. Furthermore, as we show in our experiments, the difference in performance for Spark’s primary RDD programming interface between Python and Scala reaches a factor of 5. Spark’s primary optimization efforts have been directed towards reducing the network and I/O overhead and also towards providing a simple user interface.
1. Introduction

To address the performance issue, Spark developed a DataFrame API, which detains programmers from employing user defined functions in Python, but provides better performance and the possibility for relational optimizations by the engine. Project Tungsten [9] aims to improve the DataFrame performance by generating Java code for queries at runtime. The provided data frame solution, therefore, trades the ease of use with better performance.

User defined functions (UDF) represent the key challenge in effort to get the best performance and provide rich functional programming interface. UDFs play the central role in dataflow programming. Most query processing engines allow users to specify UDFs but treat them as black boxes. This means that no relational optimizations can be performed across operators with UDFs and no compiler optimizations such as function inlining are possible. Additionally, in cases where the host language is different from the backend, like in PySpark, the UDF code has to be executed in the host language, Python in this case. This implies a lot of additional overhead in function and data serialization.

Consider the following example dataflow task in Spark:

```python
# input1 and 2 are key-value RDDs defined earlier
# 1 task
res1 = input1.join(input2).filter(lambda t: t[0] > 0)
.map(lambda t: (t[0], t[1] * 3 + 7))
.reduceByKey(lambda t1, t2: t1 + t2).collect()
```

Both input RDDs are joined on the key column, then a filter is applied on the same column. Afterwards a map function modifies the value column and reduce by key produces the final result. There are two possible optimizations that could be carried out by the execution engine for this task. The first is to push the filter before the join to reduce the join input data. The second, depending on the join implementation, is to recognize that the output after the join operator could be grouped by the key column and to use an efficient operator for reduce by key. The first optimization could be done directly by the programmer. However, this would require their deeper understanding of relational operators and would imply writing the same code twice which is always prone to errors. Furthermore, the user may not know how far the filter could be pushed without analyzing the whole DAG for every filter. The second optimization could be indicated by the user if it is possible to add annotations to the map function. This would, however, require even more database systems knowledge and programming effort from the user.

Multiple surveys show that Python is the most used programming language for data analytics [5, 12], gaining in popularity among statisticians and data scientists over R. The same Spark survey [10] shows that between 2015 and 2016 Scala usage had dropped by 6%, while 4% more users switched to Python. Python is an interpreted scripting language that relies on efficient
extension modules for heavy numerical computations. This also means that any user written code that does not employ the built-in efficient functions will perform considerably worse.

The goal of this thesis is to show that it is possible to write dataflow programs productively in the Python dynamic scripting language and achieve close to the bare-metal performance at the same time. We implement our framework for the single machine use case and compare it to the existing tools available for data analytics in Python. The framework architecture is separated in the backend and frontend. The frontend provides a dataflow programming interface to the user with full UDFs support. The backend is implemented in C++. We use Numba Python library to translate the UDFs’ code to LLVM IR. The query engine generates code for individual queries at the runtime and compiles everything together into tight native machine code. We demonstrate up to two orders of magnitude performance improvement compared to Apache Spark and data analytics libraries available in Python.
Chapter 2

Background

The focus of this thesis lies on the efficient execution of queries that are written in a dynamic general purpose programming language, using a dataflow programming model. In this chapter we present the concepts which are used in the subsequent chapters.

2.1 RDBMS

In traditional Relational Database Management Systems (RDBMS) users formulate queries in a declarative Structured Query Language (SQL). SQL builds upon relational algebra with some syntactical additions which vary between different vendors. The whole query is processed and optimized by the execution engine at once. The backend execution system receives from the user enough information to perform the computation in parallel and to apply query optimizations based on rules derived from relational algebra. RDBMSs rely on the built-in functionality making the user-defined functions (UDF) difficult to write and less attractive in terms of performance.

Implementation Conventional RDBMS systems are built following the Volcano iterative model [20]. Each operator, such as scan, join, or select, has the same interface which consists of three methods used to prepare, retrieve and clean up tuples between the operators: \textit{next}, \textit{open}, and \textit{close}. The operators process tuple by tuple by calling the \textit{next} method of the upstream operators and outputting the result. This interface, while providing a simple model, has high CPU costs associated with it and comes from times when the disk speed was the primary performance concern. Processing every single row in a relation involves performing multiple expensive virtual function calls which also reduces the possibility for compiler optimizations. In the original iterator model the computation is controlled by the sink operator which calls first the \textit{next} method of its upstream operator and receives tuples one
2. Background

Figure 2.1: Dataflow operators. Each operator maps the input to 0 or more tuples of the same or different structural type.

by one. In this context we may say that the sink operator pulls the input from its sources. An alternative approach is push based, where the computation is controlled by the source nodes which push their result tuples to their downstream. This could allow for a more efficient selection operator which could be implemented without a loop. However, some operators are impossible to implement efficiently in a push based pattern, such as a merge operator on a sorted input from two upstream sources. Nonetheless, as discussed in [31], both approaches perform similarly for the majority of benchmarks consisting of complex queries.

2.2 Dataflow functional programming

Dataflow programming is a declarative model where the user specifies operations to be performed on data sets. Dataflow programming can be more expressive than relational algebra, yet the programmer conveys enough information to allow for parallel execution and query optimizations by the execution engine. The users specify a lineage graph by chaining the operators which build a directed acyclic graph (DAG). In contrast to relational algebra and SQL, the dataflow programming model also allows for any level of structural nesting in the processed datasets.
2.2.1 Operators

Dataflow operators accept first-order user-defined functions which are applied on every tuple. Each operator may output zero or any number of tuples depending on its semantics. Map operator, for example, always returns the same number of tuples it was given. Figure 2.1 lists the fundamental operators with their output cardinalities. Map, flatmap, and reduce by key operators may change the tuple type and structure, if specified by the operator UDF. Every operator takes a collection as the input and produces a collection as the output. Many programming languages such as Python, Scala, Haskell, and Java have built-in constructs for functional programming. The pure functional dataflow programming is stateless and the user-defined functions pure, i.e., their output depends solely on their input and they have no side-effects. Additionally, similarly to SQL the reduce UDFs should be associative. These constraints allow the execution engine to run the workflows in parallel by partitioning the data between threads or between individual cores in a cluster setting.

2.2.2 Execution

The functional interface, provided by Python and other programming languages, evaluates every operator eagerly. The operators work on collections and return collections. This implies computing all intermediate results which are not needed by the user. Eager evaluation consumes more memory and may result in more computations than is optimal for a particular query plan.

A better approach is letting the user specify the whole query first without evaluating every operator individually. The execution system can then pipeline individual operators and make inter-operator optimizations, as well as reduce the memory footprint to the minimum necessary for a given query.

2.3 User-defined functions

User-defined Functions (UDFs) are the central piece in the dataflow programming. Through UDFs users can customize operations performed on the data. In RDBMS, UDFs provide a possibility to extend the functionality of the existing system. They are normally implemented as stored procedures and can be used inside every SQL clause. Some RDBM systems like PostgreSQL allow to specify the UDFs in a general programming language like C. Otherwise they have to be written using extended SQL syntax. Dataflow programming frameworks generally use the host programming language to specify the UDFs.
Most of the systems treat UDFs as black boxes and call them via function pointers on every input tuple. This implies performance penalties and extra data movements from the processor registers and onto the stack. This performance leak is similar to the iterator model backend architecture.

2.4 Dataflow frameworks

In this section we briefly describe some of the existing frameworks for dataflow functional programming available in Python programming language. We look at a single machine and a cluster setting.

2.4.1 Single machine

The Python Data Analysis Library (pandas) [8] provides a dataframe interface for data analysis. Dataframes are similar to relational tables. Every operation on a dataframe is executed eagerly which results in high memory consumption and higher-than-optimal compute times. Dataframes do not support structural nesting well. To reduce total memory consumption, users can process the data in small batches, chunks, and merge the results manually. Internally Pandas uses NumPy [7], the Python numerical computation library, for all its heavy computations. NumPy is written in C and uses BLAS libraries for matrix computations. Yet, in presence of UDFs in a map operator for example, the standard Python interpreter is used, which may slow down the computation by several orders of magnitude.

Dask [2] on the other hand, allows to work with larger-than-memory datasets and directly out of the box. Internally Dask uses Pandas for all computations on dataframes but undertakes the computation scheduling, data partitioning, distributed execution, and streaming. All operations are executed lazily, building the task graph. Additionally, Dask provides a bag interface to parallelize computations across collections of any Python objects. This is useful to work with semi-structured data like JSON or XML files. For faster execution of queries involving shuffling between partitions, users have to convert bags to dataframes manually. Dask does not perform any query plan optimizations, such as predicate push-down, even though a full task DAG is available, thanks to the lazy evaluation.

2.4.2 Machine cluster

As previously mentioned, Dask implements a distributed scheduler to use across multiple compute nodes. It is, however, still in a beta phase with no fault tolerance built-in yet.

Apache Spark [34], on the other side, is a state of the art distributed computation framework which runs on large cloud clusters built of commodity
2.4. Dataflow frameworks

hardware. RDDs, resilient distributed datasets, is the primary data abstraction provided by Spark. It is similar to Dask bags and allows to work with arbitrarily structured data.

Apache Spark is written in Scala and integrates well with other Apache big data frameworks running on JVMs. This, however, means that if the user decides to use Python as the host language, all operations on RDDs have to be serialized and deserialized between the workers JVM and Python processes. Furthermore, UDF code is executed by the Python interpreter as specified by the user without further optimizations. Spark also provides a dataframe interface in Python. In this case, Python plays a role of just a scripting language and all of the computation is done inside the JVM. This, however, removes the possibility to use custom Python UDFs inside the operators, which is also true for the dataframe interface in Scala. The main focus of Apache Spark has always been big data processing in a cloud setting where fault tolerance and I/O costs were the primary concerns.
3.1 Architecture

In this chapter we present our novel approach to execute data flow programs most efficiently on modern hardware in the presence of user-defined functions (UDF). Our goal is to compile user queries into tight native machine code loops, maximizing data locality and making use of code vectorization. Additionally, we want to be able to inline UDFs during compilation to enable further compiler optimizations and to eliminate virtual function calls from the hot execution path. And finally, being able to analyze the UDFs’ code provides new possibilities for query plan optimization. In the following, we present two different prototypes for the proposed system. One is written completely in the Python programming language and leverages native machine code compilation with the Numba [6] and one with the backend completely rewritten in C++. Both architectures provide a similar to Apache Spark [34] functional user interface in Python (see for instance Chapter 1 programs). In particular, we implemented the following operators: join, map, filter, reduce, reduce_by_key, and cross_product.

3.1.1 Pure python

In order to better understand the frontend requirements, we have built a complete framework using only the Python functional tools such as \texttt{map()} and \texttt{filter()}. We used this experience in the subsequent iterations to adapt the architecture depicted in Figure 3.1. The main principle is to explicitly separate the supplied data flow program into stages. Every individual stage is a pipeline consisting of operators which work in a tuple-at-a-time fashion. Pipelining operators which do not require intermediate data materialization, allows the compiler to keep variables in registers longer, thus increasing the data locality and making the computation more cache-friendly. The code
3. System Overview

Figure 3.1: Initial architecture. The DAG is separated in pipelined stages. Each stage is compiled into native machine code by Numba. An executor object takes care of calling different stages and passing the input. The result from the last stage is passed back to the user.

for the individual stages is then compiled at the runtime with Numba into machine native code. In detail, the engine functions as follows:

1. The user supplies the data flow program in a form of a directed acyclic graph (DAG) of operators and their UDFs. The user also specifies the action to be performed (e.g., collect, count, reduce) and the input source.
2. The stage scheduler breaks up the DAG into stages. Stages are delimited by operators which need to materialize any of their sources.
3. For each stage, we generate an imperative Python code in the AST in-memory format. All calls to the supplied UDFs are embedded into AST.
4. The generated AST is then compiled to a native machine code with the Numba framework.
5. Finally, the executor runs the stages in the scheduled order by passing the materialized output from one stage to the next.

**Advantages:** By dividing the query DAG in separate stages explicitly, we have a more fine-grained control over the execution. Every stage can be executed as a separate task which provides an additional level of parallelism. It also makes batched data processing easier, as the input to every stage can be adjusted at the runtime for better load balancing. The whole stage is compiled by Numba into efficient, vectorized machine code. In addition, im-
implementing everything in one programming language has its own benefits. For once, as stated previously, the development effort is lower. Moreover, we do not have to take ABI compliance into account, because the execution is driven by the Python interpreter and the machine code generation is handled by Numba.

Limitations: The main limitation is that by generating Python code, we have no explicit access to the memory and hardware the way it is possible in C++. Additionally, we are limited by the Numba support of Python language. We cannot use many of the Python built-in functions, such as the dictionaries, as hash maps or a sorting function. A solution could be to add the needed functionality to Numba directly. Internally, Numba leverages LLVM framework [11] for the machine code generation. It is possible to add new functionality by providing LLVM IR intrinsics to the Numba. There are, however, further problems with the proposed architecture, which makes the solution with a separate backend more favorable. For once, with a separate backend in C++, we can integrate seamlessly existing efficient implementations for relational operators written in C++. Furthermore, by separating the backend from the frontend we are now able to provide the user interface in any programming language that can be translated to LLVM IR. And lastly, the LLVM library for C++ conveniently provides static analysis tools for LLVM IR which we can use to analyze UDFs, something that would not be possible in Python without reimplementing the static analysis library.
3. System Overview

3.1.2 C++ backend

Our current architecture comprises two separate parts, the frontend and the backend. The frontend provides the same functional user interface as in the initial architecture. It is also responsible for building the DAG, inferring the tuple types and translating Python UDFs into LLVM IR. The backend analyzes UDFs, performs query optimizations, generates the query code and compiles the final shared library object. In particular, the system works as follows:

1. The user supplies the data flow program in a form of directed acyclic graph (DAG) of operators and their UDFs. The user also specifies the action to be performed (e.g. collect, count, reduce) and the input source. This step is the same as in the first prototype architecture. We currently support the input from any Python collection, a Numpy array, or a Pandas DataFrame. In case of a Python collection, the input will first be copied into a contiguous Numpy array. At present we do not support column-store, therefore a Pandas DataFrame is also first copied into a row-oriented contiguous array.

2. The frontend then materializes the specified DAG structure in JSON format [4] by specifying a list of upstream sources for every operator. The supplied UDFs are translated to LLVM IR using Numba, and embedded into the DAG directly. The frontend is also responsible for caching the compiled library objects for different DAGs. For this a global hash value is computed that uniquely describes a particular DAG, which is then used as the cache map key.

3. The backend analyzes the UDFs’ code, infers the schema for every operator and performs optimizations on the query plan.

4. Subsequently the code generator emits the operator building code into a file which is then compiled and linked into a shared library object.

5. The control flow returns back to the frontend which calls the generated shared library with the input, receives the result and returns it to the user. If the same query is executed in a loop, the frontend will call the same generated library for every iteration with possible different input, without recompiling the query.

Advantages: With such architecture, it is fairly simple to add new programming languages to the frontend as long as there exists an LLVM IR translator for it. We are consciously relying on the Numba framework to translate the Python code into LLVM IR to reduce the development effort and laying the groundwork for using third party libraries for LLVM IR translation.
3.2 UDF compilation

**Limitations:** For every new host language for which we would like to provide a frontend, we would need to either find or implement an LLVM IR translator. In case of Numba we also had to patch LLVM IR generation a little bit to suit our needs. More on that in the subsequent section. Furthermore, using several programming languages, more tools and libraries, we increase the final software package size and potentially the deployment effort for the end users.

The biggest advantage of having the backend written in C++, however, is the low-level access to the memory, more control over compiler and the generated machine code, and the availability of high performance relational operators and libraries such as MPI and OpenMP which could potentially be used in the future.

### 3.2 UDF compilation

Numba is a JIT compiler for Python programming language. Its main focus is the fast execution of the numerical code. It has a wide support for Numpy arrays and vector operations. In the background it generates LLVM IR code and uses LLVM tools to compile it into executable code. Through its API it is possible to obtain the produced LLVM IR. Numba, however, was not built as a general Python to LLVM IR translator and currently does not support many of the built-in Python functions. Most notably, it does not yet support operations on strings directly, a limitation which carries over to our current framework as well.

Furthermore, since Numba has to infer static types for every variable, some of the valid Python constructs cannot be compiled directly with Numba. An example is adding elements to a tuple in a loop:

```python
tup = ( , )
for i in range(n_columns):
    tup += ( i , )
```

A tuple in Python is defined to be immutable and, therefore the Python interpreter has to create a new tuple object in every loop iteration thus changing its type, which works fine in a dynamically typed language. However, in order to be able to use such a construct in Numba, the forloop needs to be unrolled, which would explicitly create a new variable `tup` for every iteration.

We implemented an optimization pass over Python AST to unroll the loops and rewrite the AST before passing it on to Numba. Additionally, we perform constant propagation for all global scope variables, such as `n_columns` in the example above, if they are known to be constant between function calls. This allows Numba to cache the compiled function. Being able to modify the UDF Python code gives us new possibilities to play around unsupported Python features in Numba without directly extending the library.
3. System Overview

**ABI:** In our setting we have to make sure that the LLVM IR code produced by Numba has the same ABI as a function declaration compiled with the clang compiler. Clang follows the Intel ABI manual specification [23] for compiling the machine code. To summarize the ABI specification: structs smaller or equal than 16 bytes should be passed by their fields as flat arguments from the caller to the callee. Conversely, larger structs are passed directly in the arguments and correspond to LLVM structs in our case. Similarly, the callee should return smaller structs directly. Otherwise the caller is responsible to allocate enough memory and pass a pointer to the callee to store the result. Furthermore, arguments smaller than 8 bytes may be passed in one 8-byte variable. For example, a struct with 4 fields of 4-byte integers would be passed as two 8-byte integer arguments. The same applies to the return type. To make LLVM IR code generated by Numba correspond to C++ declaration compiled with clang we rewrote some parts of the Numba code generation. By default, Numba would always return structs directly, regardless of their size and the number of fields. We had to provide the possibility to add an additional return pointer argument to the callee and store the results in that pointer. Depending on the result struct size, either the default code generation or our modified version is used. Function parameters in Numba are always unnested and struct fields are passed as flat list of arguments. We kept the same approach as it simplifies the code analysis as well.

3.3 UDF static analysis

In order to be able to find the best execution plan for a given query we have to analyze the UDFs’ code and understand its properties to decide which optimizations are allowed. We leverage C++ LLVM library for static analysis of LLVM IR.

3.3.1 Data dependencies

One important optimization technique in query processing is operator reordering such as predicate push down. Most of the time it might be beneficial to push the filter closer to the source to reduce the input size for consecutive operators. Furthermore, in some cases the filter can be spread to different join sources as shown in Figure 3.3. We use a similar algorithm proposed by Levy et al. [24], called predicate move around. The filter is first pulled down to the sink until at least one of the columns used to compute the predicate is not in the output anymore. Once at the bottom, the predicate is pushed up to all the sources until an operator is found upon which at least one of the columns used the filter predicate depends. Therefore, to be able to reorder different operators we have to determine their data dependencies. This would not be possible with opaque UDFs. To infer data
3.3. UDF static analysis

Figure 3.3: Demonstration of the predicate move-around optimization. Filter_2 and filter_5 have no data dependencies on previous map operators and can be pushed before them closer to the source. Furthermore, thanks to the consecutive join operator, we know that the column that is being read by both predicates is the same in both join branches. The filters therefore can be replicated on both join branches.
dependencies we compute the read and write sets of columns for every operator. A similar approach was proposed by Hueske et al. [22]. The operator is in a conflict with another operator if at least one of the columns in its read set is in the write set of the another operator. Otherwise the operators may read the same column, in which case they are not in a conflict with each other and may be reordered.

**Read, write sets:** We define the read and write sets as follows. The read set contains columns that are used to compute the result of the UDF. Columns which are copied to the output without change, for our purposes, are not in the read set. Write set are columns which are produced by this operator. With the help of read and write sets, we can make the data dependencies between different operators more precise than can be inferred directly from the DAG. If an operator creates a column which a subsequent operator reads, the second operator depends on the output of the first and they are said to be in write-read conflict. Having computed the sets of columns that are read or written by every operator, we can determine all data dependencies between query operators. To compute the read and write sets we start by assigning a new column type to every input field at the source. These columns are referenced globally by operator input fields. Depending on operator semantics, these columns are either propagated or a new column is generated. We can think of columns as the relational schema for every operator. We use the LLVM C++ library, which has static analysis tools for the LLVM modules to determine the read and write sets. Most importantly, it provides access to the list of uses for every variable definition. The algorithm to compute the read set for a map function is then as follows:

```plaintext
for arg in arg_list:
    arg_read = False
    for use in arg.use_list():
        if is_used_in_output(use):
            # copy the column type
            ...
        else:
            arg_read = True
    if arg_read:
        # add to the read set
        ...
```

Hereafter, in order to compute the write set, it is enough to walk over the output fields of the operator and check if they have a column type assigned. If not, we generate a new column type and add it to the operator write set. A particular input column can either be propagated directly to the output, used to compute another column, or not used at all. It is in the read set only if it is used to compute another output. If it is not used and not propagated by an operator, it is added to the dead set. The information about the dead
columns can then be propagated potentially up to the source operator to reduce the data size. This is similar to the projection push-down rule used in query optimization engines.

It is important to note, that the algorithm works only after basic static optimizations such as constant propagation and common subexpression elimination have already been performed. The variables should also be in a static single assignment form. In our case, the generated LLVM IR is already optimized by Numba’s backend. Otherwise, an extra optimization pass would be needed before applying the algorithm.

To illustrate the performed optimization we walk through the example data flow in Figure 3.3. The corresponding Python code:

```python
data1 = bc.collection([(1, 33, 3), (0, 0, 4)])  # collection_3
    .map(lambda t: (t[0]*3, t[0]))  # map_4
    .filter(lambda t: t[1] > -1)  # filter_5
    .map(lambda t: (t[1], t[0]))  # map_6
res = bc.collection([(1, 2.0), (5, 4.0)])  # collection_0
    .map(lambda t: (t[0], t[1]*3))  # map_1
    .filter(lambda t: t[0] > 1)  # filter_2
    .join(data1).collect()  # join_7
```

The operators are numbered uniquely in post-order tree traversal starting at the sink operator, join_7. The map_1 operator reads the column_1 to compute column_2 and copies column_0. Filter_2 reads column_0 to evaluate its predicate. There is no data dependency between the two operators because map_1 only copies column_0 without modifying it. Join_7 reads the first column from both its sources for join predicate. Here we can therefore infer that column_0 and column_3 have equal schema type and rename column_3 to column_0. After analyzing the right source of the join in the same way, we compute the read and write sets for that operators as well. In the first step of the predicate move around optimization we remove the filters from the DAG completely. Next, for both filters we start from the sink node, join_7 and walk up the DAG to find the first operator where any of the filter columns were created. In this case it is the collection source operators which both have column_0 in the write set. We create new filter operators and insert them between the map and collection operators. Note, however, that the argument order and number for both filters’ predicates had to be changed to accept tuples from collection source_3 which outputs three elements. We use the LLVM C++ library to modify the predicate function code. This implies changing the number of function arguments and renaming their uses. Here is an example how the optimized dataflow code could look like in Python:
3. System Overview

```
data1 = bc.collection([(1, 33), (0, 0)])  # collection_3
    .filter(lambda t: t[0] > -1)  # filter_12
    .filter(lambda t: t[0] > 1)  # filter_10
    .map(lambda t: (t[0]*3, t[0]))  # map_4
    .map(lambda t: (t[1], t[0]))  # map_6
res = bc.collection([(1, 2.0), (5, 4.0)])  # collection_0
    .filter(lambda t: t[0] > -1)  # filter_12
    .filter(lambda t: t[0] > 1)  # filter_10
    .map(lambda t: (t[0], t[1]*3))  # map_1
join(data1).collect()  # join_7
```

During the compilation, the compiler then would merge both filters and reduce the predicate to just `column_0 > 1` on both branches of the join upstreams.

3.3.2 Field properties

In addition to data dependencies, we keep track of useful column properties such as being `grouped`, `sorted`, or `unique`. The rules depend on the particular operator. For example, a cross product operator preserves the `grouped` property on one of its sources if it was already `grouped`. `Reduce by key` operator adds the `grouped` property to the key column. Propagating such properties from the source allows us to choose the best algorithm for a particular operator. Operators which involve data shuffling such as `join` or `reduce by key` have a much faster algorithm if the inputs are already sorted or grouped.

In Figure 3.4 we have an example data flow which represents one iteration of the k-means algorithm used for clustering in unsupervised machine learning. The collection source operator adds an index to every input point which makes this column field `unique`. The Cartesian product operator keeps this property on the key column used then in `reduce by key` operator. Also the `map 3` operator does not modify the key column, and it is not in its write set, so the column properties can be copied over. In `reduce by key 4` operator we can then use a more efficient operator implementation which works with a grouped key column. However, we cannot do the same for the second `reduce by key 6`, because it reduces on the column 16 which was created by `reduce by key 4` operator and does not have the grouped property.

3.4 Code generation

All operators are written as separate C++ templates. Each operator provides a simple iterator interface. During code generation we have to specify the operator order, their tuple types, and provide the UDF declarations. For code generation we employ the visitor programming pattern on the DAG operators. Every operator emits the operator definition, its upstream operators, tuple types and the UDF signature declaration if present. We kept the
3.4. Code generation

Figure 3.4: DAG for a single iteration of K-Means clustering algorithm

generated code as concise as possible, making heavy use of C++ template meta-programming and writing to a single text source file. The generated source file is compiled and linked together with UDF LLVM IR and operators’ definitions. The individual operator headers are precompiled, which reduces compile time drastically. We also pass the `-flto` flag to the clang compiler to enable inter-modular link time optimization, which allows the compiler to inline UDF code. Thanks to the static type information, the compiler is able to inline all the `next()` method calls between operators and to apply inter-modular optimizations. This results in very tight loops of vectorized native machine code and eliminates all unnecessary data movements between the operators, keeping the variables in the processor registers for as long as possible.
3. **System Overview**

*Result array allocation:* We employ a simple algorithm where the output result size is checked at every iteration and increased exponentially if needed. An alternative would be to allocate big enough blocks which could be processed in a nested loop without boundary checking at every iteration.

*Discussion:* In contrast to our first, pure Python approach, we do not define the stages imperatively in the current version. We can, however, still make the necessary code modifications to the operator templates to allow, for example, batch processing between different operators or, to switch, between push and pull based interface.
Chapter 4

Measurements

4.1 Experiment setup

We compared our framework with existing tools for the Python programming language, that are widely used by data scientists for data analysis and machine learning. We also look at how our framework performs in comparison to handwritten C++ code for different queries. We run all of our benchmarks on a laptop running on Intel i7-3520M, 2.9GHz processor with 16GB of RAM. All benchmarks are executed on a single core with Hyper-threading turned off. For code compilation, we use the clang-4.0 compiler toolkit and the gold linker for linking with compiler flags `-O3 -ffast-math -flto`.

4.2 Built-in operators

In a first step, we limit ourselves to using only the built-in functions provided by data flow frameworks in Python. Pandas is the most widespread data analytics library for Python which uses NumPy arrays for data storage and computations. NumPy array operations are written in C++. We also look at Apache Spark, the most popular framework for big data. Spark provides a DataFrame interface that allows to specify SQL operations in Python. DataFrames are fully executed on a Java Virtual Machine (JVM), eliminating as much overhead from different programming languages as possible. A greater performance penalty occurs when data has to be moved between the JVM process and the Python interpreter. To reduce this as much as possible, we use the built-in `rand` function for our synthetic benchmarks to generate data directly in the JVM. However, the result data still has to be serialized and returned back to the user Python process. We therefore run the same benchmarks using the native Scala client for Spark.
4. Measurements

Figure 4.1: Individual operator performance compared to handwritten C++ code. NumPy does not have built-in join or reduce by key operators.

<table>
<thead>
<tr>
<th>Host</th>
<th>sum</th>
<th>map</th>
<th>filter</th>
<th>join</th>
<th>rbk</th>
<th>map.filter</th>
<th>map.filter.join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3.1</td>
<td>272.7</td>
<td>138.1</td>
<td>169.2</td>
<td>234.2</td>
<td>125.1</td>
<td>135.1</td>
</tr>
<tr>
<td>Scala</td>
<td>3.0</td>
<td>50.2</td>
<td>25.4</td>
<td>27.8</td>
<td>45.0</td>
<td>25.1</td>
<td>21.8</td>
</tr>
<tr>
<td>JIT</td>
<td>0.09</td>
<td>0.3</td>
<td>0.7</td>
<td>0.41</td>
<td>0.43</td>
<td>0.39</td>
<td>0.21</td>
</tr>
</tbody>
</table>

Table 4.1: Spark’s and our framework’s runtime in seconds for different queries on one core.

4.2.1 Single operator

Figure 4.1 shows the performance of single operators for Python libraries and our framework. With this benchmark, we want to eliminate the operator pipelining advantage of our framework and compare single operations only. For single operations our framework generates the same imperative machine code as C++ and the measured run times are the same. For sum, map, filter, and reduce by key we used 1GB of random double arrays as the input. For the join, two arrays of size 32KB and 16MB were used.

Results Apache Spark was not implemented with CPU and single-core efficiency as a primary goal and because of its inferior numerical performance and much higher run times, we put the Spark results in a separate Table 4.1 for better visibility. Summation is a basic numerical operation and all frameworks show similar performance. Spark also shows very similar
4.2. Built-in operators

Figure 4.2: Chained operators performance.

performance between Python and Scala versions.

Map operators also has similar performance between Pandas and our framework. The serialization overhead for the result data makes Spark Python code around 5 times slower than the Scala version.

Pandas filter operation first builds a bitmap vector for the input with the supplied predicate. The bitmap is then used to filter the input. This additional materialization cost results in runtime penalty. In contrast to our framework and Pandas, Spark’s filter is faster than its map operation. This indicates again numerical inefficiency.

Join and reduce by key are around 3 times faster with JIT compilation than executing with Pandas. One of the reasons is the specialized hash function and hash map implementations generated at the runtime by our framework.

4.2.2 Operators chain

Next we looked at several operators chained in one query. Figure 4.2 shows the results. Thanks to operators pipelining, JIT compilation produces much more efficient code. In the case of a map-join, the map function is applied to the non-key field of the bigger relation in the join. The compiler is then able to move the map computation after the join prob phase. The same strategy
4. Measurements

Figure 4.3: Query compilation time.

applies in case of reduce by key, where the map function is applied to the non-key field of the relation. The produce native machine code achieves a factor of 2-4 times better performance than NumPy and Pandas.

4.2.3 Compile time

Figure 4.3 shows compilation time for different operators. Join and reduce by key take longer to compile because of the template functions needed to extract the tuple key and compute its hash value to use in a hash map. Compilation time spans between 200 and 600 milliseconds for larger operator chains. To reduce the compilation time, we compiled operator headers in a precompiled header file. Compilation time does not depend on the input size and every query is compiled exactly once which is essential if the same query is executed multiple times with different input. As we can see in Figure 4.4, the compilation time fraction decreases drastically for larger inputs, making it insignificant for larger input sizes.

4.3 User-defined functions

In this section we demonstrate the performance improvement in the presence of UDFs. For this we implemented two different queries in Pandas and our framework
4.3. User-defined functions

4.3.1 Minimal salary per department

The first query computes minimal salary per department and should return the employee id as well. It is not a trivial computation to write in SQL and is easier to express in a functional way. We benchmarked two different versions for Pandas as well as our functional interface. We provide the source code below.

### SQL way

```python
dep_mins = dep_emp_sal.drop('e_id', axis=1)
dep_mins = dep_mins.groupby('d_id').min().reset_index()
result = pd.merge(dep_mins, dep_emp_sal, on=['d_id', 'sal'])
```

### Pandas with a UDF

```python
idx = dep_emp_sal.groupby('d_id').apply(lambda df: df.sal.argmin())
result = dep_emp_sal.ix[idx, :]
```

### JITQ

```python
result = bc.collection(dep_emp_sal).reduce_by_key(lambda t1, t2: t1 if t1[1] < t2[1] else t2).collect()
```

The first two Pandas versions do not give equal results if there are several employees with the minimal salary per department. We therefore generated the test data in such a way that it is not the case. Figure 4.5 shows the runtime for different versions. In order to see how the input size affects the performance we looked at different numbers of departments and employees. The number of departments corresponds to the number of groups. We can clearly see that Pandas performance with a UDF is linearly dependent on
Figure 4.5: Runtime for computing argmin depending on the number of employees and departments as powers of two.
the number of groups, that is the number of times the UDF is called. For larger number of groups, JIT compilation is 62 times faster than Pandas with a UDF.

Pandas performance in SQL interface shows a completely different picture. Because of the join operator, the runtime is more dominated by the input size and does not change much for different numbers of groups. Our framework still outperforms it by 3-10 times.

A spike in the runtime at $2^{18}$ groups for our framework is due to the fact that the hashmap used in reduce by key operator does not fit into the L3 cache (4MB) anymore.

With this experiment we have shown that Pandas performs poorly in the presence of UDFs and that we are able to generate efficient code for the same use case.

### 4.4 Use cases

In the following benchmarks we show the performance gains from applying query optimizations to dataflow programs. We look at two different use cases. First, implementing the K-Means algorithm in dataflow programming and second, a feature engineering workflow for crime prediction.

![Figure 4.6: K-Means performance for different number of features.](image-url)
4. Measurements

4.4.1 K-Means

Figure 4.6 shows the runtime of the K-Means algorithm with varying number of features. The complete dataflow plan is shown in Figure 3.4. We compare our method with and without static analysis with Scikit, Python state-of-the-art library for machine learning. Without static analysis, reduce by key on the points, clusters array is implemented in the standard way, which involves building a hash map on the input point, even though they are already grouped. This results in more than two times the performance loss. Compared to Scikit, our framework, with static analysis turned on, reaches the same performance for smaller feature sizes. However, the performance starts to degrade as the tuple width increases. Scikit uses NumPy internally for vector operations and has to materialize some of the intermediate results in memory during the computation. This results in around 50% more memory usage than with our framework. Memory can be a limiting factor for efficient machine learning on a single machine.

With this result we have demonstrated, that we can achieve the same performance as tuned low-level machine learning library with a dozen lines of Python code. This shows that the ease of use and high performance should not be mutually exclusive.

4.4.2 Crime prediction

In this benchmark we look at a typical workflow in data science/machine learning. We have several data sources saved in the .csv file format and want to merge them in one dataframe for further analysis. Additionally, we want to modify and add features. Our main features file contains several million rows, which does not allow us to hold all the intermediate data in memory. We therefore use the Dask Python library for distributed data analytics. Dask divides the input data automatically in chunks and allows for working with larger than memory datasets.

Figure 4.7 shows the dataflow graph for this query. Through static analysis the query optimizer finds that Filter_7 may be pushed up to the source of the ‘land_no_use’ column. To assess the improvements gained with static analysis, we also modify the original Python code for the task and plot this version as Dask opt. in Figure 4.8. The features_raw[‘land_no_use’] < 1 predicate has 45% selectivity. Dask opt. shows a 50% improvement in the performance. Eliminating additional feature creation, further decreases the computation time by 50%. This shows that performing computations across multiple columns has a high performance penalty in Pandas which employs column store. With JITQ we achieve the best performance with the original query and only a slight improvement when eliminating the new feature computations. The Python Dask code for the experiment dataflow task is presented below.
4.4. Use cases

Figure 4.7: Dataflow graph for the crime prediction task. Query optimizer pushes the filter_7 up closer to the source reducing the input to the following join operators.

Figure 4.8: Data analytics performance for the crime prediction task.
4. Measurements

```python
import dask.dataframe as dd

### reading data

# convert temperature to Fahrenheit
features_raw['temp'] = features_raw['temp'] * 1.8 + 32

# add new features
features_raw['weather1'] = 3.2 * features_raw['temp'] ** 3 + 7.5 * features_raw['hum'] ** 3 + 2.3 * features_raw['discomf'] ** 3 + 5.3 * features_raw['daylight'] ** 3 + 8.6 * features_raw['moon'] ** 3
features_raw['weather2'] = 3.2 * features_raw['temp'] ** 2 + 7.5 * features_raw['hum'] ** 2 + 2.3 * features_raw['discomf'] ** 2 + 5.3 * features_raw['daylight'] ** 2 + 8.6 * features_raw['moon'] ** 2
features_raw['weather3'] = 3.2 * features_raw['temp'] ** 4 + 7.5 * features_raw['hum'] ** 4 + 2.3 * features_raw['discomf'] ** 4 + 5.3 * features_raw['daylight'] ** 4 + 8.6 * features_raw['moon'] ** 4

features_raw = dd.merge(features_raw, static_feat, on='grid_id')
features_raw = dd.merge(features_raw, census, on='grid_id')
features_raw = features_raw[features_raw['land_no_use'] < 1]

result = features_raw.compute()
```
Chapter 5

Related Work

The data base systems community aims to make the data processing as efficient as possible, given the current hardware. For a long time the major bottleneck for distributed query processing was the disk and network speed. It just took too much time to load the data from disk into memory, to consider optimizing for anything else. The modern machines, however, dispose of large memory, which makes data processing and shuffling completely in memory possible. The spinning hard drives have been replaced by faster flash storage and the network link runs over 100 Gigabit Ethernet connection. One of the most important challenges, therefore, is to optimize the CPU performance. Most of the popular frameworks for big data processing are optimized for slow networks and cheap hardware with fault-tolerance in mind. They are designed to run on large clusters of middle range computers and do not scale down well to smaller clusters of more powerful machines, and therefore cannot take full advantage of modern hardware capabilities [26]. We can identify two major research directions which aim at solving the identified problems: just-in-time query compilation and static analysis of user defined functions.

5.1 Just-in-time query compilation

Conventional database management systems come from the times when the disk and network I/O took by far the most of the processing time for a given query. The actual query computation time does not matter as much as long as just reading the data for any query takes the most time. The traditional volcano iterator model [20] equips every operator with a fairly simple interface, \texttt{open()}, \texttt{next()}, and \texttt{close()}. To compute its output, every operator performs a virtual function call to its upstream operators’ \texttt{next()} method for every tuple. While providing an easy to implement interface, the iterator model produces many virtual function calls and makes inter-procedural
5. Related Work

optimizations and function inlining impossible. A major advantage of the compiled queries is the availability of type info at runtime which eliminates the virtual function calls on the one hand and allows for higher code specialization for the given query and hardware on the other hand.

Modern system make use of just-in-time compilation to reduce the query computation time. The HyPer project [27] generates imperative the LLVM Intermediate Representation (IR) [11] assembly code from optimized relational queries. Their primary goal is to keep the data in CPU registers as long as possible for all consecutive operations and to avoid unnecessary materialization. The most important part of their method is dividing individual queries into stages which are compiled together in one tight loop with the result being materialized. They further define pipeline breakers as operators which trigger materialization and represent boundaries between stages. However, one can decide to break the pipelining earlier and materialize the intermediate results if the number of instruction in a stage becomes too big to fit into L1 instruction cache. Similar to the MonetDB [25] column-store approach, where the columns are processed one at a time in tight forloops, gathering operators in stages allows for many compiler optimizations not possible before, such as vectorization, strength reduction and loop unrolling. Furthermore, executing stages in tight loops improves data locality and reduces cache misses and also the instruction cache misses. To combat excessive intermediate result materialization the X100 project [35] further refines the MonetDB execution model by processing the column tuples in batches. When it comes to performance, HyPer compiled queries were 2-3 times faster for Q1-Q5 TPC-H benchmark than MonetDB and up to two orders of magnitude faster than a conventional RDBMS.

Most notably, by generating and compiling LLVM IR, in contrast to C++ code, HyPer could reduce the compilation times by up to two orders of magnitude to just under 50ms per query. In our experiments, using precompiled C++ headers, we could achieve compile times of under 500ms for all queries. However, generating LLVM IR assembly makes code maintenance and extensions more expensive in terms of developer effort.

Similarly to our approach, Tupleware [16, 17] framework generates C++ code for a given query plan on one side and translates UDFs to LLVM IR code on the other side. The query plan and the UDFs are compiled and linked together, using the clang compilation tools. Similarly to HyPer, Tupleware generates imperative C++ code for individual queries making code maintenance harder. Furthermore, in order to support UDFs written in the Python programming language, the authors implement their own Python to LLVM IR compiler which is again associated with higher development costs and makes supporting new front-end languages more difficult. Overall, Tupleware performance comes close to that of HyPer on single node for
SQL workloads and outperforms Apache Spark in a cluster setting for both, machine learning and SQL workloads.

High Performance Analytics Toolkit (HPAT) takes a different approach on efficient query execution [32]. It leverages Julia programming language and the ParallelAccelerator package [13] developed by Intel to generate MPI/OpenMP C code. The main goal of the project is to generate high performance distributed code automatically from scripting languages to keep the user productivity and the code performance high. In contrast to our method, programmers have to write queries using vector operations instead of dataflow operators such as map. The HPAT’s backend uses than different heuristics to find the best way to partition the data and to distribute the tasks. On the one hand, this can give more flexibility to the users. But it also means that the heuristics might fail to find the best execution possible, if a new pattern is introduced by the user. The authors evaluated their method against Apache Spark and handwritten C++ MPI/OpenMP code on different machine learning algorithms. Compared to Apache Spark, HPAT performs up to three orders of magnitude faster on the Cori supercomputer [1] and up to 250 times faster on Amazon AWS and is only 2-4 times slower than the hand-written C++ code. Spark’s pure performance on the supercomputer is explained by the master node being the bottleneck in the computation. Currently, HPAT is also working on a Python frontend using Numba and LLVM for code generation [3].

### 5.2 Static UDF analysis

A major advantage of just-in-time compilation is the possibility to adapt the generated code to different parameters known only at runtime. By looking inside user defined functions we can infer many important properties of the whole query.

#### 5.2.1 Semantics

In order to enable operator reordering, it is important to define data dependencies between them. Hueske et al [22] provide a notion of read- and write-sets for every operator similarly to what we have discussed previously. Having computed these sets for every operator, they can infer data dependencies between operators. The authors propose using a static code analysis framework for Java bytecode in their case, to compute read and write sets. Such a framework should provide control flow graph and an access to uses and definitions of a variable. This is similar to what we could achieve with the LLVM C++ library. In contrast to our method, the paper focuses on operator reorderings and therefore does not discuss further applications for static analysis such as detecting and eliminating unused columns across
operators, as discussed in Section 3.3.1. Furthermore, by keeping track of column properties in the intermediate operators, we can pick the best suited algorithm for shuffle operators. According to its authors, the proposed techniques could successfully find operator reorderings in experimental TCP-H queries that resulted in performance improvements of up to one order of magnitude. Moreover, the authors were able to find optimizations that algebraic optimizers could not.

5.2.2 Execution cost

Operator reordering decisions require an estimate of the execution cost for every operator. Knowing that a filter operator can be pushed up to the source does not necessarily imply that it is a good idea to do so. It actually may be cheaper to put the filter after a join or another filter, if the supplied filter predicate function is computationally heavy. Hellerstein et al. further propose to consider using memoization technique to cache the result for given arguments, if the input domain is known to be small enough or if only a small subset of all possible values is used [21].

Dursun et al. estimate the CPU cost of individual UDFs by counting the instructions in the function body [17]. In the presence of loops, they also try to approximate the possible number of iterations. Furthermore, in their approach, authors propose to consider the input size and the memory bandwidth to compute the data load time for UDFs:

\[
\text{LoadTime} = \frac{\text{ClockSpeed} \times \text{InputSize}}{\text{Bandwidth}}
\]  

(5.1)

Having computed the CPU time and the input load time, every UDF can be labeled as either compute or memory bound. The authors use this metadata in order to better divide operators into pipelines. For example, if a UDF can be vectorized but is discovered to be memory bound, it will be more beneficial to merge this UDF’s operator with the next stage to profit from data locality instead of executing the whole operator in a separate stage.

5.3 Lazy evaluation and staged programming

In this section we discuss existing frameworks that provide a dataflow functional interface and allow for operator pipelining through lazy evaluation. Apache Spark [34] provides users with a functional data flow programming interface where individual functions are chained together to define a directed acyclic graph (DAG) of operators. The computation only starts when the user specifies an action such as \textit{count}. Lazy evaluation allows the query processing engine to analyze the whole DAG and schedule the execution...
in the best way possible. Additionally, it avoids materializing intermediate operator results which are not required by the user. To allow query optimizations, Apache Spark implemented new DataFrame API, which allows users to specify their workflows in a similar to SQL manner but still be able to use host language constructs such as functions and objects. Since the UDFs remain black boxes to the engine, DataFrame APIs only allow a subset of the host language to be used inside Operator UDFs such as `data.filter(data["my.column.name"] > 0)`. This is similar to other Python DataFrame APIs such as pandas [8]. In an effort to improve the CPU performance in Apache Spark, project Tungsten was introduced. It aims at improving the Java Virtual Machine memory management. Additionally, it generates Java bytecode for queries. The Python users, however, can only profit from the new performance improvements if they use the DataFrame API, which does not allow arbitrarily user defined functions inside the operators.

Dask is another recent project, that aims at distributed query execution on individual workstations and clusters[2]. It comes in form of a lightweight library for Python. Internally dask relies on NumPy and pandas libraries for computations and takes care of data distribution and query execution. Currently, it is one of the best solutions for data scientists and analysts who work in Python programming language. Dask, however, does not support lazy evaluation and operator pipelining and also treats UDFs as black boxes.

In contrast to Apache Spark, Weld [29] aims to add lazy evaluation across different libraries used for data analytics and machine learning. The library operators are wrapped in Weld expressions which delay the execution and allow to generate specialized code. The authors show that Weld outperforms HyPer database in SQL TPC-H benchmarks and was close to hand written optimized C++ code performance. The reasons they outlined are vectorization of the generated code in the backend and a lightweight hash table implementation. They also measured against the Apache Spark framework on a 21 node cluster and achieved a 6.1x speedup on Q1 and Q6 from TPC-H benchmark. The reason for the speedup is the machine code generation instead of Java code. In the presence of a user-defined function code in a map operator, Weld measured a 21x speed up with a map UDF followed by a reduce. Integrating with Weld would require from library developers to add glue code to their frameworks and individual operators, which, according to the authors of the paper, should not require a lot of effort and could be achieved in several graduate student workdays for a single library.

Multi-staged programming techniques aim at generating type-safe code at runtime. Lightweight Modular Staging [30] represents the staged expressions as a tree and keeps a global symbolic reference to each expression. The users can then specify which code to generate based on the runtime
argument values. Flare [18] makes use of the LMS framework to generate code for UDFs defined in Scala. The users, however, have to wrap the UDF code to be able to use the framework. The compilation in this setting takes up to 1.5s for a single query, which can be explained by the overhead imposed by Scala type checking. Flare could achieve up to 89x faster run times than Apache Spark in a distributed setting. One of the major reasons for the better performance is the fact that Flare generates code for the whole query instead of single stages, as Spark does. Compared to HyPer, Flare executed some queries up to 4.5x faster. Queries that could profit from indexing were executed faster by HyPer, since Flare does not currently support indexing.
In this thesis, we have investigated the possibility of implementing an efficient dataflow query engine in a dynamic programming language. We have analyzed the existing data analytics tools for Python. Apache Spark provides simple to use functional dataflow programming interface and is one of the most used frameworks for big data. Pandas is a data analytics library which relies on efficient Python extension modules written in C++. Big data frameworks scale poorly to smaller clusters with better hardware and do not take full advantage of modern hardware. The problem with this framework is performance gap compared to efficient, hand-written C++ code. We also show that the existing data analytics tools for Python perform poorly in the presence of user-defined-functions written in Python. The UDF code is executed in the Python interpreter and the difference in performance reaches several orders of magnitude in execution time.

We have identified just-in-time-compilation of dataflow tasks as the key to high performance. We use Numba to translate Python user code to LLVM IR. The execution engine then generates code for every individual query which is then compiled and linked together with the UDFs’ code. The compiler is able to inline the UDFs and perform intermodular link-time optimizations. This results in tight vectorized native machine code. We have shown that our high-level abstract dataflow operators with UDFs reach the same level of performance as handwritten C++ code.

With static analysis, we are able to infer the semantics of user supplied code. This allows the query engine to perform various well-studied query optimizations from relational databases research. As an example, we have implemented the predicate push-down optimization. We are also able to keep track of useful column properties such as being sorted or grouped. This can be used to choose the physical plan for the query.
6. Conclusion

**Outlook**  One limitation of our framework is the support for string data types by Numba. In the future we should investigate how to address this.

Another obvious direction for improvement is to support multicore and multiple node computation. The input data can be divided into chunks and distributed between nodes, each of which execute individual computation stages in parallel. From the statical analysis perspective it could be beneficial to dissect individual UDFs into independent parts. For example, a filter operator which contains a conjunctive predicate could be separated in several filters which could be moved around independently to allow further query optimization. Furthermore, by profiling at runtime and through static analysis, we can keep track of different statistics per column such as min, max, even, odd, and others. This could further open new possibilities for query optimizations.
Bibliography


[26] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost?


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PYSPARK AT BARE-METAL SPEED

Author(s) (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

AKhadov

First name(s):

Sabit

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 29.09.2017

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.