Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Improving Multi-Library Data Analysis Pipelines Through Abstract Representation and Co-compilation

Author: Radu Jica 2534945

1st supervisor:	Dr. Hannes Mühleisen	Centrum Wiskunde & Informatica
2nd reader:	Prof. Dr. Peter Boncz	Centrum Wiskunde & Informatica

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 2, 2018

Abstract

Typical data analysis pipelines involve the use of multiple libraries, many with their own internal data representation, hence requiring intermediate result materialization and internal conversions upon execution. Weld and Julia are two approaches using intermediate representations (IR) that aim to improve these pipelines along with using LLVM for backend optimizations. In this work, the benefits brought by full-pipeline cross-library optimizations is evaluated and compared through implementations of a representative data analysis pipeline. Both Weld and Julia show speedups of up to 50% compared to standard implementations however still fall short to the ideal performance.

Contents

1 Introduction **4**

2 Exemplary Data Analysis **9**
2.1 Input Data 9
2.2 Pipeline 12

3 Related Work **16**

4 Weld Extensions **19**
4.1 Weld 19
4.2 Caching Intermediate Results 23
4.3 Lazy File Parsing 24
4.4 Results & Evaluation 27

5 Experimental Setup **34**
5.1 Implementation Specifics 34
5.2 Results 37
5.3 Discussion 44

6 Conclusion **46**

Appendix **48**

Chapter 1

Introduction

Typical (big) data processing and analysis pipelines involve the use of multiple libraries, each with their own internal data representation, and hence require several internal conversions upon execution. Such an example can be seen in Python when data scientists read and process data with Pandas but train a machine learning model in Tensorflow, thus requiring a conversion between Pandas dataframes[1] and Tensorflow tensors[2]. The reasons behind using specific libraries may be ease of use, different features, or even familiarity, hence the well known trade-off between efficiency and productivity.

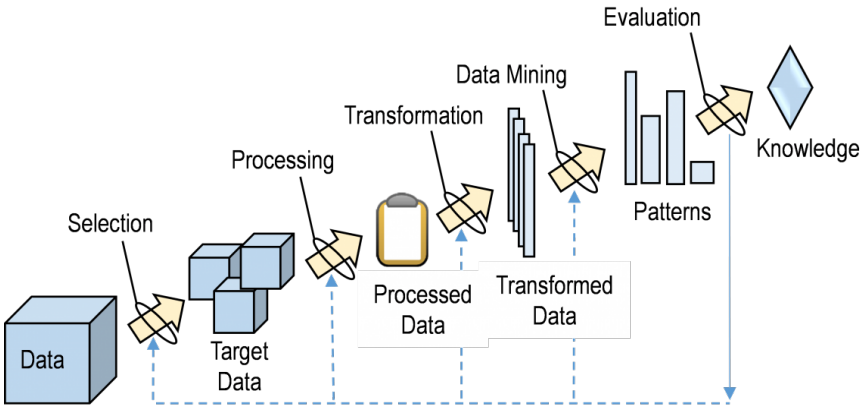


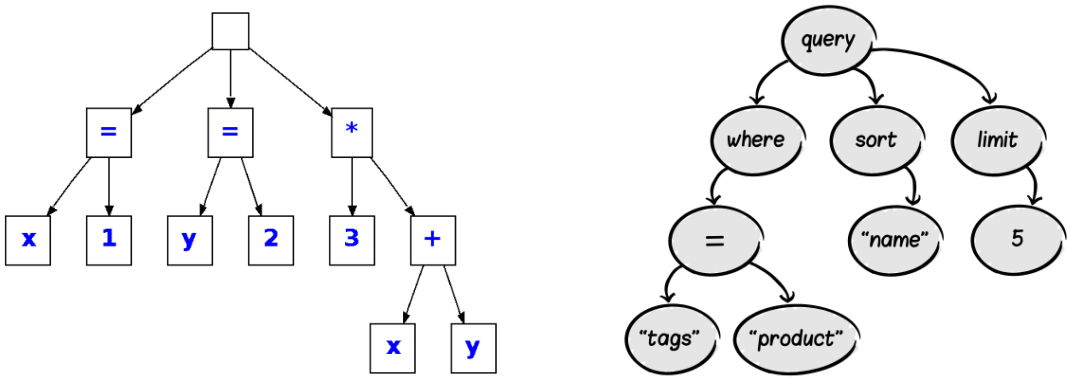
Figure 1.1: Example data analysis pipeline in the context of Data Mining [1].

A Data Analysis Pipeline involves doing a specific set of operations on different input data with the same structure. For example, input data might be temperature and precipitation measurements which are continuously generated over time and stored in a given format. One may wish to find correlations between these variables. Since the processing steps are usually the same but only the data changes, one would write a pipeline for analysis. The process of extracting knowledge from data has been named Data Mining [2] or Data Wrangling [3]. As exemplified in Figure 1.1, extracting knowledge typically requires multiple processing steps before relevant patterns may be observed. When dealing with unknown data, the first step one

[1] `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html`
[2] `https://www.tensorflow.org/api_docs/python/tf/Tensor`

would take is to look at the *head* of it. This would reveal the type of the data, if values are missing, and metadata such as column names. The metadata is also important to parsers which could optimize file reading, for example by reading in 4-byte chunks when knowing that a file stores data only as 32-bit integers.

Often data is spread out in multiple files, therefore it needs to be *joined* into one piece. This operation is expensive and an important topic in databases [4] where certain features of the data may be exploited to choose the appropriate algorithm. For example, sorted data allows one to use a merge join, or disproportionate table sizes indicate that a hash join would be more efficient. Data might also be more meaningful when *grouped by* a specific value and *aggregated*, for example finding the average temperature in a month to use as a predictor for future months. To even reach this step, one might need to convert values from one form to another. This might be a simple operation, such as multiplying by X, or more complicated requiring the use of a *User Defined Function* (UDF).



(a) AST of a computation [5].

(b) DAG of a query [6].

Figure 1.2: Examples of intermediate representations.

Intermediate Representations (IR) can improve data analysis pipelines through better expressiveness and optimizations. Examples of IRs include the Abstract Syntax Tree (AST) used in programming languages compilers to parse source code, annotate with properties, and eliminate unnecessary parts when creating the machine code while potentially applying optimizations for fast execution, as seen in Figure 1.2a. Another example consists of the queries in databases which are represented through Directed Acyclic Graphs (DAG). These allow optimizers to re-order nodes for better execution, as shown in Figure 1.2b. Source code itself can be seen as an intermediate representation of the programmer's intended computation and machine code. By representing computations in an IR, one could optimize them and avoid intermediate result materialization.

Improving multi-library data analysis pipelines is possible through an intermediate representation which can be optimized with LLVM [7]. Zaharia et al. [8] propose a framework - Weld - which can be used to extend existing libraries such that during compilation, an intermediate representation can be optimized with both database- and compiler-inspired optimizations. This is achieved by using a lazily-evaluated API which allows the construction of a computation graph in Weld IR. Upon evaluation, this graph is optimized. Through the lazy API, Weld is able to limit intermediate results materialization and thus only output the required results. The

Weld IR is then converted to LLVM IR allowing LLVM to apply further compiler optimizations. This framework hence assumes that any library may represent its methods through the Weld IR. Pipelines are thus optimized across multiple libraries and efficient code runnable on multicores and GPU's is generated. One must be able to map the data structures in a library to the data structures available in Weld. This is easily supported in C/C++, however, for other languages such as Python, one must implement encoders and decoders that are able to translate from one format to the other in an efficient manner.

Lazy Evaluation is, as the name implies, evaluating a computation not immediately but only when required. It is a concept which is linked to functional programming languages due to their similar computational model. For example, multiplying every value in an array by a scalar number can be done by iterating over its elements in a for loop and multiplying. The same computation is described in a functional fashion using the *map* operator, as seen in Listing 1. Functional programming presents an elegant syntax for describing computations in a data-centric way. For the above example, one can notice the operation is essentially a 1-to-1 mapping over the data. However, there are also operations such as aggregations which combine all array elements into fewer results, such as for finding the max value. These are coined as *reduce* steps in functional programming.

```
1 val array = Array(1, 2, 3, 4)
2
3 val res = new Array[Int](4)
4 for (i <- 0 until array.length) {
5         res(i) = res(i) * 2
6 }
7
8 val res = array.map(x => x * 2)
```

Listing 1: Example of a for loop versus its corresponding map operator in Scala.

The functional representation of computations provides an effective area for parallelizing work over multiple computers and optimizing the computations. An array can be easily split into multiple chunks and distributed on multiple computers to effectuate a mapping operation in parallel. However, for reduce operations, there is extra work that needs to be done to combine the data while maintaining data integrity and also speed. Frameworks such as MapReduce [9] and Spark [10] take advantage of these concepts and provide an environment where one can code increasingly complex computations which are lazily recorded and then parallelized to any number of computers.

When desiring multiple computations on the same input data, there are also opportunities to optimize. For multiple map operations, one could merge them in a single loop instead of iterating multiple times over the data, as seen in Figure 2. However, these optimizations might only be possible in the background of the computations, i.e. users cannot directly code them. Thus, frameworks should be able to automatically apply them.

Another approach involving LLVM and that benefits entire data analysis pipelines

```
val res = array.map(x => x * 2).map(x => x + 1)

val res = array.map(x => x * 2 + 1)
```

Listing 2: Example in Scala of how two map operators could be combined into one.

is using the new programming language Julia [11]. Julia combines the ease of use of a dynamically typed programming language with the performance of a statically typed one. This is achieved through a rich type system supporting an expressive programming model and efficient type inference. At compilation, the type is immutable, hence, e.g., checking the type of a function parameter every time the function is called, can be avoided. Julia compilation, as shown in Figure 1.3, starts by converting the syntax to a type inference suitable IR which is then Just-In-Time (JIT) compiled through LLVM, hence benefiting from LLVM's optimizations. The language's primary abstraction mechanism is generic functions which is practical for both mathematical and object-oriented programming styles. Julia further allows directly calling C and Fortran functions. Since entire Julia programs are optimized through LLVM, data analysis pipelines written in Julia would also be improved.

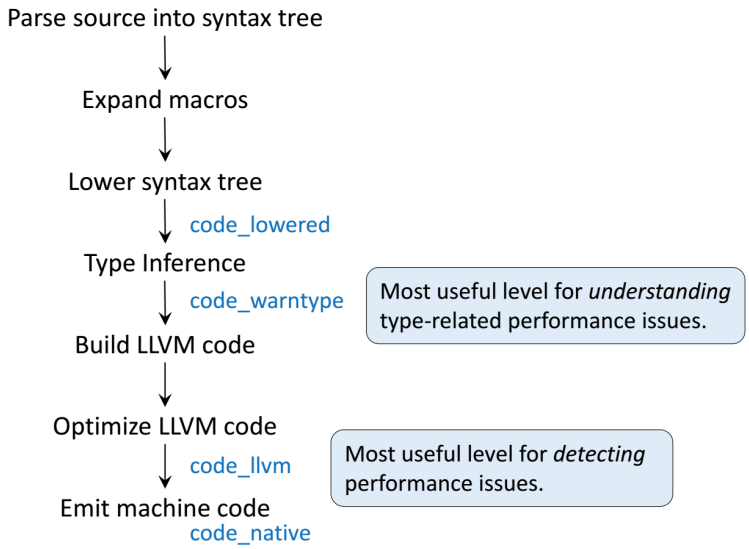


Figure 1.3: Julia JIT compilation process [12].

There is research focused on optimizing the compilation itself, though typically providing domain-specific solutions. Darkroom [13] is designed for hardware synthesis by providing an IR for line-buffered image processing pipelines which is more efficient than storing intermediate results in DRAM. Erbium [14] provides a concurrency model IR similar to OpenMP, enabling the compilation of data-flow tasks into streaming processes. Work is also done in better parallelizing programs through annotations for converting high-level code to efficient low-level parallel code through OpenMP/MPI [15, 16, 17].

Data formats and their design are often overlooked. Data may be stored in text or binary form, with the latter being typically more compact. An efficient storage model might allow compression and certain optimizations when parsing. This is

typically achieved by files starting with a header describing the contents, such as in the NetCDF [18] and Parquet [19] column-store formats. Among other benefits, such formats allow parsing specific subsets of the data. Column-stores provide further advantages, as seen in the context of databases [20]. It is easier to combine operations, such as maps and filters. Cache performance is also improved as the cache lines are not polluted with irrelevant nearby data.

Given so many different programming languages and frameworks, one is faced with the decision of choosing the most appropriate one. The performance along with memory and CPU usage then become deciding factors. The overheads, scalability, and bottlenecks are different depending on the framework. Libraries might also offer more functionality out-of-the-box than others, thus leading users to choose time over performance. A problem when comparing, though, is that new software is often presented with micro-benchmarks; a more general view of the performance over a wider array of operations is lacking. When choosing a library to implement an intricate data analysis pipeline, micro-benchmarks become less relevant because many operations need to be efficient, not just a few. It does not matter, for example, if map operations are incredibly fast when a join takes very long or worse, it is not supported. Therefore, testing such libraries on a full data analysis pipeline is expected to provide an important insight into performance, usability, and the benefits brought by their optimizations, IRs, co-compilation, and multi-library capabilities. The research questions in this work are thus:

1. How do approaches such as Weld and Julia compare against other data analysis pipeline implementations, such as Python and R?
2. What are the bottlenecks in current data analysis pipelines and how do the proposed approaches improve them?

Outline

The following chapter details the input data used in this work along with an exemplary data analysis pipeline to compare different implementations. Chapter 3 describes related work in this area. Two extensions to Weld are proposed and tested in Chapter 4 along with a description of Weld internals. Chapter 5 presents the experimental setup and the results of comparing the different pipeline implementations. Lastly, Chapter 6 discusses the research questions and points out possible future work areas.

Chapter 2

Exemplary Data Analysis

The input data used in this work is stored in the NetCDF file format and is described in the following section. Next, an exemplary data analysis pipeline is presented. This pipeline is designed to be comprehensive and to put stress on the implementations. Opportunities to optimize the pipeline are also identified. Lastly, the tools used to measure performance are briefly described.

2.1 Input Data

The input data is the E-OBS gridded weather dataset[1] of the Koninklijk Nederlands Meteorologisch Instituut (KNMI) containing data on a regular grid with a resolution of 0.25 degrees. It covers the area 25N-75N x 40W-75E with measurements recorded every day between 1950 and 2017. There are thus 3 dimensions: longitude, latitude, and time. There are 5 variables stored in the NetCDF classical 3.0 format (allowing up to 4GB per file), each in their separate file: daily mean temperature TG, daily minimum temperature TN, daily maximum temperature TX, daily precipitation sum RR, and daily averaged sea level pressure PP. Each variable has a pair variable recording the daily standard error, for a total of 10 variables. There are 46GB of columnar binary data with over 1 billion lines which is the ideal dataset to investigate performance, subsetting, and optimizations. Note that converting the data to CSV would result in a 6 times increase in size, therefore the full dataset would be approximately 280GB as CSV.

NetCDF

NetCDF[18] or Network Common Data Form is designed to hold array-oriented scientific data in a self-describing and machine independent binary format that allows efficient subsetting. Originally based on NASA's Common Data Format, it has diverged from version 4.0 basing itself on HDF5 [21], a format allowing hierarchical data. With version 4.0, files can go over 4GB in size due to 64-bit memory. Since it is a binary format, data is stored in a compact manner taking little storage space.

[1] `http://surfobs.climate.copernicus.eu/dataaccess/access_eobs.php`

Furthermore, parsing NetCDF is more efficient by exploiting type and dimensions information which would otherwise not be possible in other formats.

NetCDF is self-describing by including a header which details the names, types, sizes, dimensions, scale factors, missing value representations, and other metadata. Essentially a column-store, NetCDF stores arrays one after the other. N-dimensional data can furthermore be efficiently encoded by only storing the distinct values which, through a cartesian product, would form a unique key identifying each data point. Scale factors allow floating point data with few decimals to be stored as lower bit length integers. One could reach the original data by multiplying the stored data with the scale factor. Thus, one saves space. For example, the float value 12.43 can be encoded as 1243 16-bit integer with a scale factor of 0.01. With data points separated by a given character such as commas, missing values can be simply represented with nothing but replaced upon parsing with the *_FillValue* defined in the header.

Efficient subsetting is possible given the header data for types, sizes, and dimensions. Since after the header there is only raw data, a parser can skip entire arrays (or columns in a table) through offsets given the length and the size in bytes of given types. Similarly, a parser can select slices of the arrays (or rows in a table) by computing offsets from dimensions information. Take, for example, a dataset with 3 dimensions for coordinates plus time; knowing the lengths of these dimensions, one can directly filter and read data for specific coordinates or any combination of the dimensions.

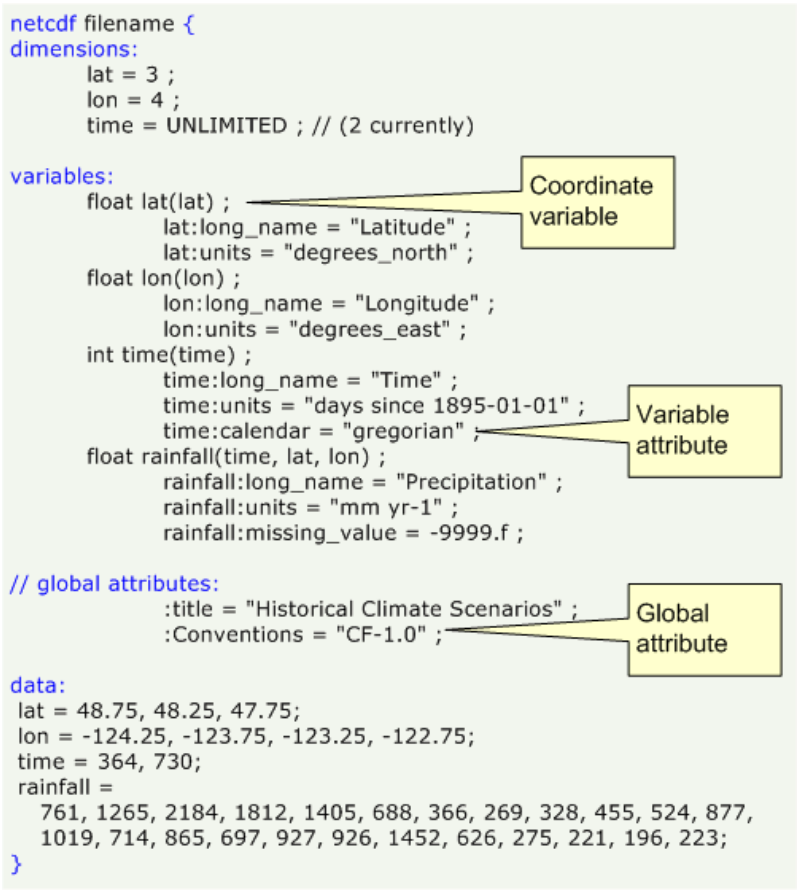


Figure 2.1: Example text CDL file for a NetCDF file [22].

NetCDF files can be freely converted to Common Data Form Language (CDL) files and back allowing easy interaction with this data format. Figure 2.1 shows a human-readable text representation of a netCDF file as CDL. One can define the dimensions which through a cartesian product uniquely identify the variables data points. Note how dates are encoded as numbers of type integer but through the 'days since' metadata can be converted to meaningful dates. Global attributes for the file can also be stored along with convention versions to aid the parsing of various slightly different formatted files.

Pre-processing

Data available in a table-like format is more useful therefore we would like to bring all the data in a single data structure. Furthermore, we are interested in the performance over different input sizes to identify the breaking and/or bottlenecks of each implementation. Therefore, there are 2 pre-processing steps done on the above data. These allow testing various input sizes while simplifying the data analysis pipeline.

File Combining Bringing all variables in a single table requires joining them, however measuring the performance of parsing 10 files and doing 9 joins is excessive. Therefore, there is a single join with variables grouped into 2 files, as seen in Table 2.1. Since the size of these files would exceed the previous limit of 4GB, they are converted to the 4.0 classical NetCDF format which allows larger sizes and is based on the HDF5 format.

Filename	Variables
data1	tg, tg_stderr, tn, tn_stderr, tx, tx_stderr
data2	pp, pp_stderr, rr, rr_stderr

Table 2.1: Input files and variables for the data analysis pipeline.

Subsets They are chosen on a scale factor of 2 from the total size with the same starting date. The actual filter used when creating the subset files is done by selecting between 2 dates, as seen in Table 2.2. Converting from number of days to date intervals is done through an online tool[2]. The sizes in the table are in GB.

Name	Start date	End date	Size (%)	No. days	Size NetCDF	Size CSV
data_100	1950-01-01	2017-12-31	100	24837	46.328	277.829
data_50	1950-01-01	1983-12-31	50	12418	23.163	138.908
data_25	1950-01-01	1966-12-31	25	6209	11.582	69.457
data_12	1950-01-01	1958-07-01	12.5	3104	5.790	34.722
data_6	1950-01-01	1954-04-01	6.25	1552	2.895	17.361
data_3	1950-01-01	1952-02-15	3.125	776	1.447	8.677
data_1	1950-01-01	1951-01-23	1.5625	388	0.724	4.342
data_0	1950-01-01	1950-07-13	0.78125	194	0.362	2.171

Table 2.2: Subsets of the input data used in benchmarking.

[2] `https://www.timeanddate.com/date/duration.html`

2.2 Pipeline

The pipeline is centered on creating the features necessary for a weather forecast machine learning model. There are many models in literature [23] that show promising results in weather predictions, however the model itself and its training are excluded from the pipeline here. The data analysis pipeline in this research focuses more on the data mining aspect, i.e. parsing the data, first views of it, and data wrangling steps taken towards training a model. The steps are shown in Figure 2.2 and the individual steps are detailed next.

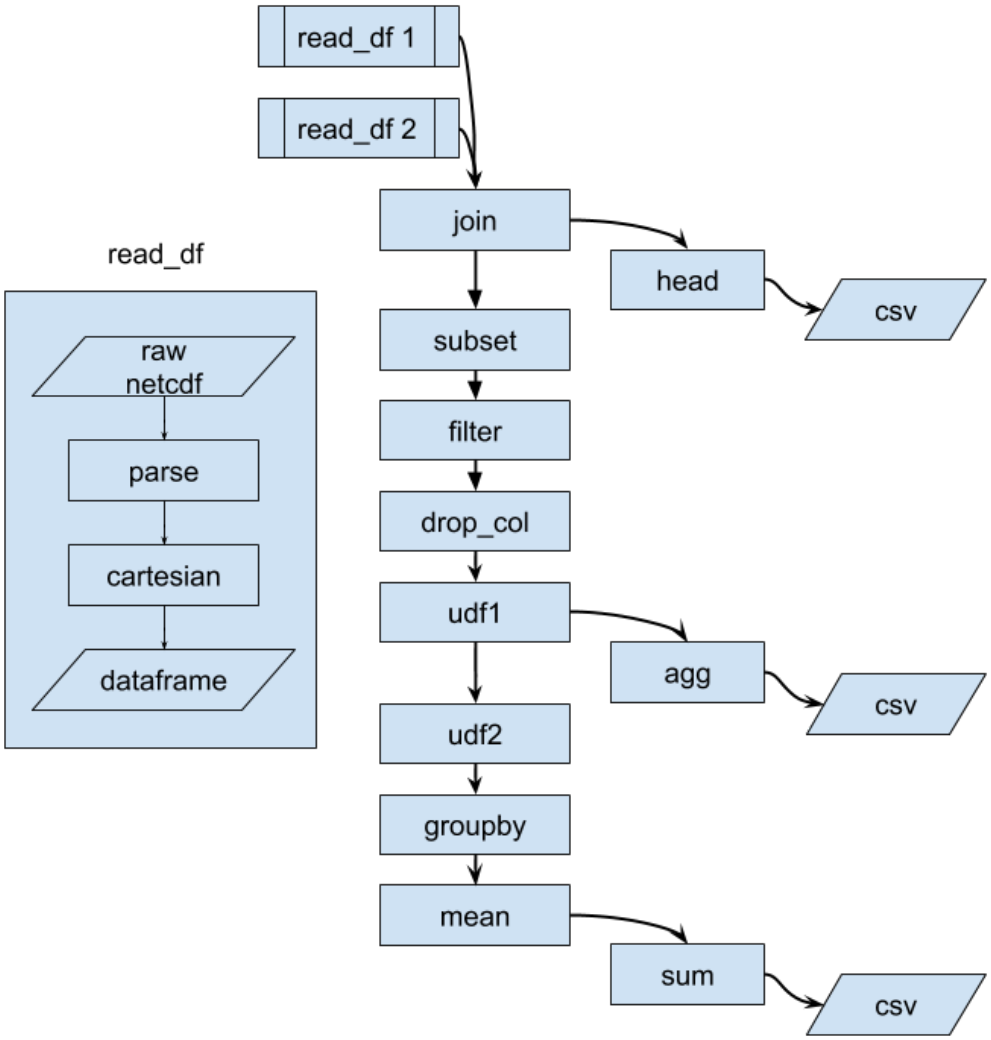


Figure 2.2: The data analysis pipeline.

read_df The first step involves **parsing** the data and converting it into a table-like data structure. The low-level parser used within all the implementations in this research is maintained by UCAR[3], however each high-level implementation would have different settings en-/disabled by default. To ensure the same results the following steps are taken: 1) missing values are replaced by the values described in the *_FillValue* attribute, 2) raw data is scaled appropriately given the *scale_factor*

[3] `https://www.unidata.ucar.edu/downloads/netcdf/index.jsp`

attribute, 3) data is read/converted to 1-dimensional arrays for use later in the pipeline, and 4) type is preserved, unless multiplied by a scale_factor upon which the type will be *float*, as converted by default by the parsers.

The dimensions of the dataset are stored separately, as 3 arrays. Upon a **cartesian** product, the resulting pairings uniquely identify each data point. This operation is necessary for all the following steps, however could be done in multiple ways. In NumPy terms, broadcasted views over these dimensions would be enough. However, whether the materialization in memory of this cartesian product is necessary and/or useful is debatable.

join The datasets have been pre-combined into only 2 files such that a single join could be measured (as opposed to 9 joins for the 10 columns). Upon manual inspection and metadata checking, both files have the same *sorted* index (after the cartesian product), and could hence be simply appended next to each-other for best performance, as the rows match 1-to-1. However, this fact is assumed to be unknown when the pipeline is written and hence is up to the join implementation to figure it out, if possible. It is required to keep only the rows in both datasets, therefore an inner join is specified.

head The first evaluation step. Viewing the first 10 rows of the data is a very common operation, however can be more difficult with a lazy API. A good implementation does not read everything and then select the head of the data but start by just reading the first 10 rows. This is particularly problematic when the index of these rows actually relies on the result of the cartesian product. Note that this step is aided by placing it at the start of the pipeline, before any other filter/map operations are done.

subset It is common to decide, perhaps after viewing the head of the data, that only a subset of it is required. Often this would be a filter step, however this input dataset has certain features: the order and the length of the dimensions are known from the header. This means that when requiring only the data for some given dimensions, in this case the longitude which is the first dimension, one can compute the number of the required rows as seen in Listing 3. The slice selects the data with longitude between -10.125 and 17.125. This roughly translates to western Europe and amounts to 23% of the total rows. If only the required data could be read, it would bring performance benefits. This is also possible given the NetCDF format accepts subsetting. This is done at this point and not explicitly when parsing to emphasize any possible optimizations done by each implementation.

```
1 start = longitude.index_of(-10.125) * length(latitude) * number_days
2 stop = (longitude.index_of(17.125) + 1) * length(latitude) * number_days
```

Listing 3: Formula to compute the slice on the dataset given dimensions.

filter The data contains missing values which will not be used, therefore filter out rows with missing values. Note that to keep this step fairly simple, it was observed that filtering out the missing values from only three columns would remove all rows with missing values, therefore this is the exact chosen filter.

drop_col It is assumed that some columns from the original dataset are not necessary after-all, therefore filter them out. Again, the pipeline could be optimized to not read the corresponding data at all except the subset necessary for the head at the previous evaluation step. The pipeline drops 2 out of a total of 10 columns (13 including the dimensions).

udf1 Mapping is the perfect 1-to-1 operation that can be parallelized, therefore it is also a step in this pipeline. To add an extra layer of difficulty, it is written as a UDF which, in a well optimizing pipeline, would not be treated as a black box but 'understood' along with the other operations for optimizations such as loop fusion. This udf computes the absolute difference between two of the columns and stores the result in a new column.

agg As the second evaluation step, compute the minimum, maximum, mean, and standard deviation of each column. These statistics are again very common, informative, and put a distinctive load on the pipeline. These aggregations could also take a varying amount of passes over the data, depending on the implementation. For example, minimum and maximum could easily be computed in the same loop. Furthermore, the implementations should also handle very large numbers, particularly in computing the mean.

udf2 As a prerequisite to the following groupby step, this UDF will go in a finer detail by creating a new column representing a year + month format. This is the kind of UDF that is difficult to *not* treat as a black box given the operation on strings, despite being a 1-to-1 map in reality. The UDF essentially converts the 'yyyy-mm-dd' date format to 'yyyy-mm'. The result is then stored in a new column.

groupby With the purpose of computing a statistic per month, the groupBy operation will use the previously created year-month feature along with the coordinates to create groups. For each group, the mean of each column will be computed. At this point, this would be the final evaluate in the pipeline. However, to check the correctness of the implementations and to create a small result, the mean aggregations are summed per column to create a small dataframe. This last sum operation is included in the pipeline.

The 3 evaluate steps described above are forced by printing to csv files. The costs of formatting to csv are low since there is little data to print, i.e. up to 10 rows, therefore they are negligible to the timings. The pipeline described here is hence different to other benchmarks in that 1) there are multiple evaluation points, 2) we are interested in the overall performance, not micro-benchmarking, and 3) as the pipeline progresses, the evaluation steps rely on the previous steps. Running separate queries for each evaluation starting from raw data, as one could do naively with SQL, would prove to be inefficient due to the need to recompute several of the previous and expensive steps on each query.

Possible Optimizations

The above pipeline presents several locations where optimizations could be applied. First, the pipeline features both a subset and column selection/drop, therefore the data filtered out could just not be parsed in the first place. This might not be

possible for all data formats, however it is possible with NetCDF. Second, the join operator could deduce that the columns on which the join is done form, in fact, a unique primary key which is also already sorted in both input dataframes. Therefore, the columns could just be appended next to each other. This is a feature often implemented within databases [24] which indeed require to gather this sort of statistics while parsing the data. However, with a lazy API, this shall be possible outside of databases as well. The dimensions of the data, which are the join columns, could also not be explicitly materialized through the cartesian product.

A third possible optimization is by merging filters, maps, and UDFs which is possible only through a lazy API. All are 1-to-1 operations and therefore could also be easily parallelized. With eager evaluation, each operation would be executed immediately making it impossible to optimize. Particularly with UDFs, this can be observed only if they are not treated as black box operations. A fourth optimization is computing the aggregation in as few data passes as possible. For example, the min and max can be computed in the same loop while also computing the sum required for the mean. Optimizations are also possible through vectorization while doing these passes over the data.

The groupby operator could also be optimized on a single thread when knowing that the columns on which the operation is done are already sorted; one could iterate and add members to a group until the value in the last column changes without actually comparing all the values for every row. Coupled with the mean aggregation next, this can be done in one less pass over the data by computing the sum and count while grouping the entries. Other compiler optimizations, such as loop unrolling and dead code elimination, could also improve the pipeline overall. Effective compilers choose code representations for optimum performance.

Experimental Data Gathering

To test the performance of pipeline implementations, both total and profiling measurements are required. Each implementation is executed 5 times to obtain an average execution time with the caches being cleaned in-between each run. Total measurements are done using Linux's `/usr/bin/time` command, as opposed to Bash's time command. This allows one to track not only memory and CPU usage but also disk I/O. The pipelines are profiled using *collectl*[4] which is similar to the Linux `ps aux` or `top` command however can provide a wider arrange of details.

Collectl is set to profile the specific pid of the pipeline. Since the pipeline pid is obtained after the pipeline is started, profiling measurements might miss the first second of execution, which is deemed acceptable. Memory, CPU, and I/O usage are recorded every second. Lastly, the main steps in the pipeline are delimited by timestamped markers which are saved in a separate file. These markers along with the timestamped profiling data shall provide insight into which steps take longer to execute in each implementation.

[4] `http://collectl.sourceforge.net/`

Chapter 3

Related Work

The optimizations that can be done on data analysis pipelines are typically inspired from either the database or the compiler world. Common to both is the use of an Intermediate Representation (IR) which poses certain challenges in terms of flexibility and completeness during its design. K. Palacz et al. describe in [25] the software design patterns used in creating the OvmIR, detailing the trade-offs between an easy-to-use IR and its extensibility. IR's typically form a Directed Acyclic Graph (DAG) [10, 26] which is the skeleton used by database optimizations such as predicate pushdown. The Abstract Syntax Tree (AST) used by compilers is also an IR which is used to perform rule-based optimizations. LLVM [7] is a compiler framework which generalizes the three-phase compiler design by proposing a first-class language (IR) with well-defined semantics as the middle-phase. This means that, if a language can be reduced to this IR representation, then it can be used to generate machine code for various hardware architectures without developing its own compiler. Furthermore, LLVM is known to aggressively optimize across all compilation stages.

Database-inspired optimizations became increasingly relevant over time as developers find it more productive to work with relational interfaces than pure lower-level programming. This can be seen, for example, with the development of SparkSQL over Spark, and Pig or Hive over MapReduce. There are well studied optimization techniques for relational databases which can therefore be now applied to data analysis pipelines. Optimizations, such as predicate pushdown and join reordering [4], are implemented in such systems, for example by SparkSQL through Catalyst [10]. However, Catalyst optimizations apply only within SparkSQL, therefore possible optimizations across a pipeline involving multiple such frameworks are lost. A solution is through Algebricks [27] which proposes the use of a common IR that, if implemented by each relevant framework, would be able to optimize across them and generate data-parallel programs for clusters without losing user-facing API. This framework is a data model-agnostic query compiler built on HyRacks, a push-based dataflow-parallel runtime similar to Hadoop. The implementor of a high-level query language then has access to every step of the compilation to add custom rules, types, or optimization passes.

Query optimizations, however, have been ported to non-relational contexts. Tupleware [28] is a new framework which uses such optimizations across UDF-centric

workflows. The writers observed that UDFs are often treated as blackboxes and that libraries do not optimize for complex analytics or different hardware. Through Tupleware, one can encode computations into a workflow graph with the TSet abstraction similar to Spark's RDD and the UDFs in LLVM IR. From there, the system applies both high-level query optimizations, such as predicate pushdown, and low-level optimizations. It analyzes the UDFs for characteristics, such as CPU cycles and vectorization, and inlines the functions in the parallel code generated from the workflow graph. Musketeer [29] is designed to decouple backends from data analysis pipelines such that queries are optimized through its IR and, depending on the result, Musketeer can (automatically) choose the most appropriate backend. For example, instead of running Hive on Hadoop, one is able to run it on Spark if the system observes that it provides better performance. Lara [30] is an embedded language in Scala which enables authoring scalable pipelines while combining the needs of linear algebra and relational algebra from typical machine learning tasks and ETL, respectively, through its own optimizing IR. Database optimizations have also been brought to R [26], where they are used within the Renjin compiler on an IR to improve the performance of any R script.

Compiler optimizations, such as loop fusion and loop fission, are typically seen in code generation contexts where rules are applied on an IR. There is much research focused on parallelizing code at the low level, i.e. the compiler itself, but also higher up in the pipeline. Tapir [31] aims to optimize compilers for parallel tasks by embedding fork-join parallelism in the IR of the compiler by representing logical fork-join parallelism asymmetrically in the program's CFG. The authors show that by converting the compiler IR to Tapir IR, one is able to apply optimizations such as tail-recursion elimination and common-subexpression elimination to LLVM through the use of only 3 new operators: sync, detach, reattach. These make the LLVM IR seem serial such that LLVM can optimize. Darkroom [13] optimizes line-buffered image processing pipelines by converting C code to an IR and optimizing it with the goal of minimizing intermediate data storage. The output of Darkroom can then be used to synthesize hardware descriptions for multiple architectures, such as ASIC or FPGA. Pegasus [32] goes a level lower by translating C programs into hardware circuits through an IR for the CASH compiler which is optimized. Erbium [14] presents a concurrency model to convert dataflow tasks into streaming processes through an IR on multicore systems. This model seems to improve over OpenMP, however OpenMP itself has potential for improvement, as shown by INSPIRE [33]. Here C code, optionally using frameworks such as OpenMP/MPI, is parsed with LLVM's frontend (Clang) and then converted to an IR which is then optimized through compile-time analyses.

As shown in [34], in order to reap the benefits of parallelization one might need hundreds of cores with existing systems to be any faster than a single threaded implementation. There is research in improving parallelism in high-level programming languages through user-friendly annotations. ParallelAccelerator, proposed in [16], aims to intercept Julia's AST just before LLVM compilation and enhance it with new nodes aimed for parallelization which can then be optimized and used to generate efficient MPI/C++ code. This is achieved merely by adding an annotation to relevant functions. However, extra care must be given when writing functions for them to be eligible for parallelization. This approach is extended and automated

in [15] through a dataflow algorithm exploiting domain knowledge and high-level mathematical semantics without using approximations such as cost models. This approach proves to be the middle ground between Spark and MPI for performance and productivity providing a speedup of up to 2000 on a supercomputer when compared to Spark. Pydron [17] provides annotations to parallelize sequential Python code to be run on multicore, cluster, or cloud infrastructures. This approach similarly requires special care when writing functions, however provides a more complete backend that can distribute the work.

Spark SQL could benefit from implementing certain operations in C and interfacing them with Spark's Java runtime, as shown in Flare [35]. Taking this one step further, Flare presents a new runtime which drops the Spark assumption of Google-level scaled-out cluster and the requirement for fault tolerance. This provides performance similar to HyPer [36], a hybrid OLTP & OLAP main memory database. Flare can then be taken another step further to combine multiple workflows, including relational and iterative algorithms expressed as UDFs. This is done by using the Delite compiler framework embedded into Scala, possible through Scala's reflection and metaprogramming features. The optimized logical query plan returned by Spark's Catalyst is thus mapped to OptiQL which can then be used with the Delite framework for parallel, distributed, and/or GPU work.

Chapter 4

Weld Extensions

Weld presents a framework that covers all themes in this work: multi-library, IR, co-compilation, and optimizations. Thus, it is investigated in detail to understand its benefits and drawbacks. This chapter starts by an in-depth description of Weld's design, components, and its integration with Python. Next, a Welded version of Pandas is presented that is later used to compare Weld with the other implementations. This new library is built on top of two extensions to Weld which are presented next: 1) intermediate results caching, and 2) lazy file parsing. These extensions are expected to benefit Weld execution and are tested and evaluated in the last section.

4.1 Weld

Core to Weld is its intermediate representation of the computation graph. Weld's IR is a statically typed, referentially transparent language with built-in parallel constructs. Essentially a large string, the computation graph is built by sequentially adding chunks for each desired computation. This graph as a whole is called a Weld *function* and consists of a list of named arguments and an expression for the result. Each expression in Weld is a pure function of its inputs and there are no side-effects. This means Weld features a stateless design. Figure 4.1 shows how the computation graph is built by encoding a map and then a filter operation in Weld IR, followed by an aggregation. The input is defined in a header between vertical bars (|). Note that the input is named and represents a vector of 32-bit integer values.

There are two basic data types implemented: values and builders. The values can be scalars, vectors, dictionaries, or structs of various types. Weld uses builders to construct results in parallel from values that are merged into them. From the previous Figure 4.1 example, the map and filter operations *merge* computed values into *appender* builders to construct new result vectors. The aggregation uses a different builder called a *merger* which features the commutative binary operator + to compute the sum of all the values. The result would hence be a scalar. Weld also futures higher-level syntactic "sugar" for common functional operators. For example, the previous map operation could be reduced to just 'let res_m = map(x, |e| e * 2)'. These functional operators are then replaced with their actual for loop representation upon compilation.

```
x = [1, 2, 3, 4, 5]

x = map(x → x * 2)


x = filter(x > 4)


sum(x)
```

```
|x: vec[i32]|

let res_m = result(for(x, appender[i32],
                       |b, i, e| merge(b, e * 2)))

let res_f = result(for(res_m, appender[i32],
                       |b, i, e| if(e > 4, merge(b, e), b)))

result(for(res_f, merger[i32, +], |b, i, e| merge(b, e)))
```

Figure 4.1: Example of sequentially encoding operations from functional pseudo-code (left) into Weld IR (right).

```
|_inp0:vec[i32]|
 result(
  for(
   _inp0:vec[i32],
   merger[i32,+],
   |b__5:merger[i32,+],i__5:i64,n:i32|
    (let tmp:i32=((n:i32*(i32(2))));
    (let e:{i32,bool}=({tmp:i32,(tmp:i32>(i32(4)))});
    if(
     e.$1,
     merge(b__5:merger[i32,+],e.$0),
     b__5:merger[i32,+]
    )))
 ))
```

Figure 4.2: The Weld-optimized IR for the computation in Figure 4.1.

When desiring to evaluate to a result, one needs to first compile a Weld *module* upon which Weld takes the IR and applies several optimizations, including loop fusion, vectorization, and inlining. When creating the module, one is able to select which optimizations to apply through configuration keys. The result of these optimization passes is then an optimized IR. Figure 4.2 shows the IR of the previous operations after the optimization passes. Note that all for loops are now fused into one and that all data types now have an inferred type. Weld then takes the optimized IR and maps it to LLVM IR, allowing LLVM to apply further compiler optimizations, such as aggressive dead code elimination and loop unrolling. In order to get a result, one passes to Weld references to the input data and runs the module. Weld allocates memory for each evaluated result and the implementer has to free this memory when no longer required.

Weld's design allows the use of any programming environment as long as the required computations can be reduced to its IR. However, there is an extra restriction: the data structures must also be convertible to/from the source language and Weld's C++ implementation. The conversion feasibility then relies on how easy C++ can be integrated with the chosen programming language. Weld provides a generalized

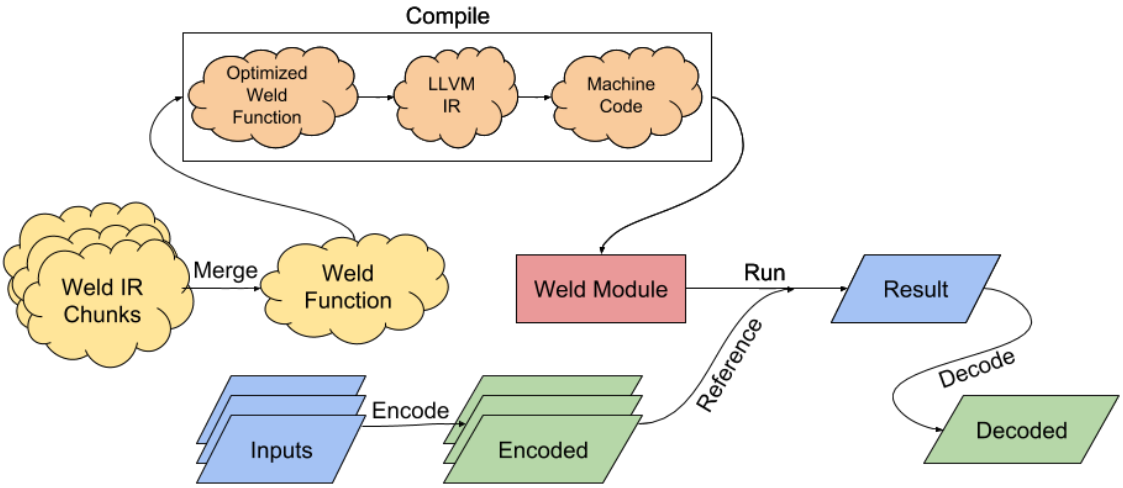


Figure 4.3: The Weld execution pipeline from IR and input data to result.

abstraction to this conversion process through encoders and decoders. The input data from a library must first be encoded to a format Weld can understand before execution. After execution, the result must be decoded in a format the library can understand and use further. Figure 4.3 shows an overview of all the steps taken during Weld execution. Note that this decoupling of the inputs from the module creation allows one to compile once and execute multiple times, as long as the data is of the same type.

Weld with Python

Python is a general purpose programming language emphasizing code readability. Due to this user-friendly focus, it has become highly popular[1] amongst users. This lead to the development of many libraries, including ones for data analysis. Pandas [37] is such a library which arose from the need of having a table-like structure in finance and statistics. It is built on top of NumPy [38] which is a Python/C package centered around a C implementation of an N-dimensional array, the *ndarray*. Weld could be combined with Pandas to provide both usability and higher performance. To do so, one needs to provide a mapping to/from the NumPy ndarray and the Weld vector. Their underlying structures are shown as C/C++ structs in Listing 4.

Note that both store the length of the array and a pointer to the start of the data. This means converting from one to the other should be just a matter of copying pointers. However, issues arise when one desires the use of an array with more than 1 dimension. In Weld, this would be a vector of vectors or internally a struct of structs. NumPy represents this array with the same single struct and going from one dimension to the other is encoded within the dimensions and strides fields. All the data is stored in a long contiguous block which can be traversed with different offsets for each dimension. This is not the case in Weld. Encoding a multi-dimensional NumPy into Weld is still possible by copying/creating pointers, however when decoding a Weld struct of structs, data needs to be copied into a

[1] As seen in annual surveys on StackOverflow: https://insights.stackoverflow.com/survey/2018/

```
template<typename T>
struct vec {
  T *ptr;
  i64 size;
};
```

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;
```

Listing 4: C/C++ structs of the Weld vector (left) and the NumPy ndarray (right).

single contiguous block of data for NumPy to use. This highlights that despite a generalized interface, integrating Weld into existing libraries and data structures might not be so effective.

Welded Pandas

Weld authors provide an example implementation of a small subset of Pandas in a library titled *Grizzly*. En-/decoders for ndarrays of most types are available through this library. Furthermore, a separate package *Pyweld* provides an interface from Python to Weld through a *WeldObject* class. A WeldObject records an IR chunk and provides an evaluate method to en-/decode data, construct a Weld module, and execute. This is done by WeldObjects storing their data inputs in a context dictionary and their dependencies on other WeldObjects in a dependencies dictionary. This provides a clean way of building the whole computation graph at the evaluation step by traversing the context to create the header and the dependencies to combine all Weld IR code pieces into one. Furthermore, this ensures that the inputs and the contexts themselves are re-used when creating more WeldObjects. One can therefore use WeldObjects as the abstraction for lazy computation objects throughout Python libraries.

Grizzly cannot be used due to two conflicting design choices for this work. First, Grizzly's data structures start from existing Pandas data structures. This means that Grizzly is the starting point to Weld; no other computations can be Welded before Grizzly. Furthermore, this means that all data must be already in memory when using Grizzly. Second, while providing a decent subset of Pandas' operators, they are not sufficient for the implementation of the intended pipeline. Extending Grizzly is thus out of the question. Therefore, to compare Weld against other implementations, this work brings a new Welded Pandas library, *Pandas_weld*. It re-uses the en-/decoders provided by Grizzly, though with a few additions of our own for missing types, including the 16-bit integer and boolean arrays of 2 dimensions. WeldObjects are similarly used to track computations in the background.

Pandas_weld is designed to behave like an exact copy of Pandas with a larger variety of supported operators than Grizzly. The DataFrame, i.e. the abstraction for a

table, can be created from raw data or WeldObjects, hence it can be connected to other Weld computations graphs originating outside the library. The Series, i.e. the abstraction for a column/array, along with both single and multi-dimensional index classes are similarly implemented. Slicing, mapping, filtering, aggregations, joins, and groupby's are all supported and used to implement the Weld pipeline in this work. Continuing the previous example, the Weld IR computation graph is encoded through Pandas_weld as shown in Listing 5. Note that the only difference with Pandas is the import statement. Internally, the sum method evaluates the Weld IR chunks recorded so far and returns the result.

```
1 import pandas_weld as pdw
2 import numpy as np
3
4 x = pdw.Series(np.array([1, 2, 3, 4, 5], dtype=np.int32))
5 x = x * 2
6 x = x[x > 4]
7 x.sum()
```

Listing 5: Creating the Weld IR for the previous example in Figure 4.1 through Pandas_weld.

There are, though, still slight differences to Pandas. Pandas internally uses highly abstracted data structures, such as the NDFrame for n-dimensional tables. These have been omitted from the Pandas_weld implementation due to the increased complexity of mimicking the behavior. Another difference can be seen through UDFs. Pandas_weld provides two Weld-tailored methods for UDFs: first method accepts Weld IR directly and second integrates with Weld's *cudf* operator which dynamically links a compiled C++ UDF to the Weld runtime. These methods are of course not available in Pandas however have been used in the implementation of the pipeline for udf1 and udf2, respectively. Lastly, to showcase Weld's multi-library design, the cartesian product has been implemented in a separate self-sufficient package.

4.2 Caching Intermediate Results

Weld provides a great way to encode computations in its IR, thus eliminating intermediate result materialization. Within a single evaluation, Weld can apply effective optimizations to speed up computation efficiency. However, Weld cannot optimize across multiple evaluation points as part of its stateless design. Adding state could prove beneficial for the implementation of the pipeline from this work. An example is the inner join. Two tables could be joined by obtaining two arrays, one for each table, and encoding the row indices that should be kept from the input tables in the resulting joined table. These arrays would be required at any operation encoded after the join for each evaluation step. Since there is currently no direct mapping between a Weld dictionary and a Python dict, one is left to encoding each column as a separate WeldObject which means that upon evaluation, each column relies on the join result. Therefore, the Weld IR representing the join has to be added to the Weld IR of each column, as illustrated in Figure 4.4. Since each column has to be

evaluated separately, this would trigger the join operation to be executed once for each column.

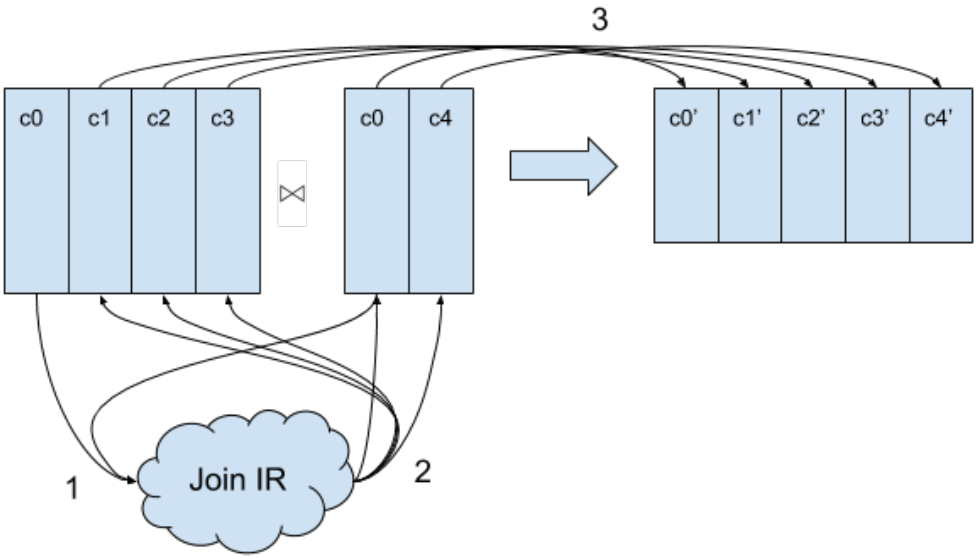


Figure 4.4: Encoding the join into a table where each column is a separate WeldObject.

It would be good to be able to store intermediate results when deemed useful, such as in the example above. This can be achieved by implementing a *cache* operator in Weld. Continuing the join example from Figure 4.4 with intermediate result caching, step 2 would instead be directed to a *single* separate result. This result would then be evaluated a single time and used as dependency to compute step 3. As proof of concept and due to easier implementation, intermediate results are implemented within the LazyResult Python framework described next. This only brings an extra checking point in the evaluation steps that would fetch already computed results from a cache instead of re-computing the whole chain of operations. Since intermediate results are implemented at a level below Pandas_weld, they can be used with any libraries.

4.3 Lazy File Parsing

Due to its lazy API and multi-library support, Weld presents the perfect testing ground for the concept of lazy file parsing. The idea is centered around tabular data where only fragments of it are actually needed during data analysis, however this fact is unknown to the user in the early stages. In other words, the responsibility shall be handed over to the library itself which shall deduce that only specific chunks of data are required from files during execution. This is, of course, only possible with a lazy API which tracks all processing steps up to evaluation points. An example subset can be seen in Figure 4.5. It is unnecessary to parse all the data when only the subset surrounded in red is required.

Disk I/O is known to be orders of magnitude slower than memory and cache, as shown in Figure 4.6, therefore reducing it shall provide significant speedups and lower

Id (Name)	Region	Length (m)	Area (km^2)	Slope	Aspect	Mean Z (m)	Median Z (m)	Min. Z (m)	Max. Z (m)	Range Z (m)
1	C	513	0.109	22.7	E	2472	2471	2360	2590	230
2	P	452	0.135	19.5	W	2587	2593	2469	2682	213
3	P	641	0.155	27.5	NE	2607	2619	2413	2785	372
4	P	779	0.210	18.0	N	2485	2480	2379	2651	272
5	P	922	0.264	21.1	N	2523	2528	2324	2718	394
6	Q	731	0.276	22.2	SE	2532	2545	2256	2692	436
7	Q	952	0.422	21.3	NW	2386	2383	2228	2621	393
8	Q	1366	0.563	18.3	N	2283	2279	2093	2494	401
9	P	1386	0.648	25.7	NW	2420	2415	2117	2968	851
10	P	1552	0.650	26.6	NE	2406	2400	1990	2751	761
11	C	1173	0.653	21.2	SE	2452	2445	2246	2733	487
12	P	1671	0.830	21.6	N	2399	2407	2169	2646	477
13	C	1017	0.860	20.7	NE	2363	2343	2135	2720	585
14	C	1419	0.913	16.2	NE	2322	2314	2175	2580	405
15	P	1327	0.983	23.8	NE	2665	2662	2248	3157	909
16	P	2027	0.995	24.5	NW	2699	2716	2171	3093	922
17	P	1584	1.145	26.7	S	2728	2724	2313	3188	875
18	Q	1209	1.349	16.7	E	2362	2364	2172	2515	343
19	Q	2199	1.609	10.8	SE	2469	2478	2187	2633	446
20	C	1870	1.816	18.5	E	2387	2398	2044	2749	705
21	Q	1796	2.188	15.7	S	2421	2449	2052	2669	617
22	Q	2965	2.464	12.3	N	2438	2436	2198	2803	605
23	Q	3002	2.655	18.5	N	2482	2490	2054	3019	965
24	Q	3562	3.049	17.3	NW	2352	2359	1991	2789	798
25	Q	2300	4.280	14.9	N	2463	2488	2061	2752	691
26	Q	3294	4.625	11.9	NW	2417	2425	2137	2708	571
27	Q	3062	7.037	12.8	NE	2416	2408	2056	2917	861
28 (Castle Creek)	C	5875	9.645	11.1	NE	2387	2397	1881	2827	946
29 (Tête)	P	5971	12.674	16.9	N	2520	2565	1649	3357	1708
30 (David)	P	5228	13.533	19.3	SW	2609	2616	1352	3339	1987
31	Q	4010	14.782	13.2	NE	2319	2334	1868	2745	877
32 (Kiwa)	P	9597	17.154	17.2	N	2617	2705	1491	3509	2018
33	C	7302	19.094	12.8	N	2273	2288	1803	2910	1107

Figure 4.5: Example of subset that may be required out of an entire table.

memory usage in data processing. Furthermore, there shall be minimal hindrances to usability as a lazy file parser shall behave the same as an eager implementation.

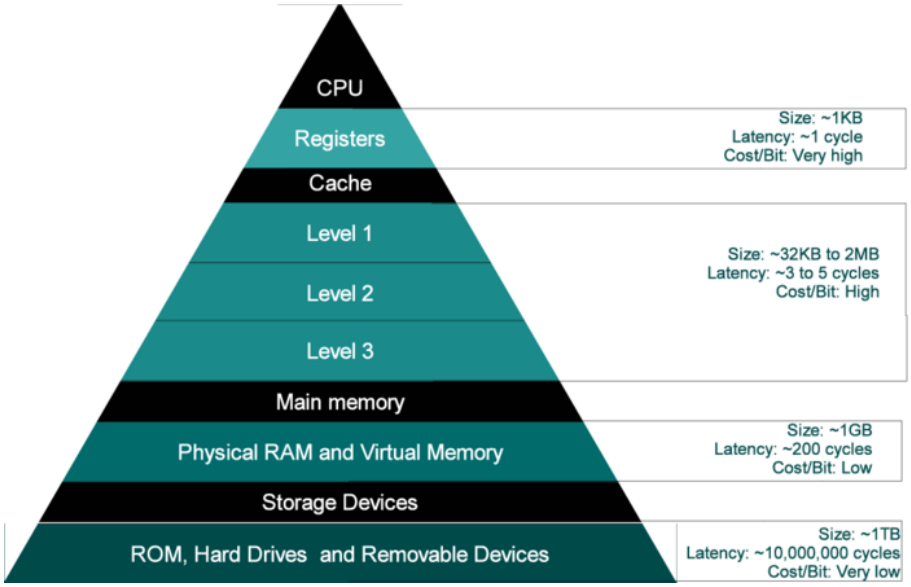


Figure 4.6: Storage hierarchy showing the difference in access speeds and sizes [39].

Framework

The core concept we introduce is the use of placeholders in place of actual data when initially reading a file. Thus, we develop a framework that provides and keeps track of a mapping between placeholders and methods to obtain the actual data. The framework must be the single bridge between multiple libraries and multiple parsers, as illustrated in Figure 4.7. Parsers must be able to return subsets of the data and this must be communicated from a library, through the framework, and to the relevant parser when an operation requests such a subset. Parsers would initially read the header and possibly some rows of data from a file to deduce as much as possible about it without reading the entire thing. Particularly, details such

as variable names, types, and other metadata. These are necessary to the parser but also highly relevant to the user.

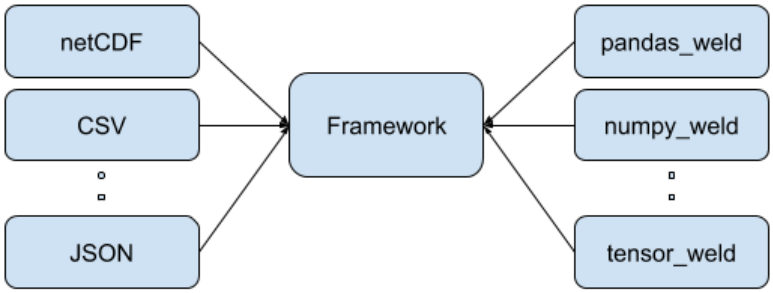


Figure 4.7: The framework must accommodate both multiple data input formats and multiple libraries.

Lazy file parsers, however, cannot work unless they are integrated somehow into the Weld evaluation steps and are also accessible to libraries built using Weld. To this end, the framework proposed here is built on top of the WeldObject, tracks lazy files/data, and substitutes the placeholders with actual raw data on computation evaluation. Furthermore, it exposes certain methods which, when used by libraries, shall indicate to the framework and hence to the parsers themselves that only fragments of the data are required. Lastly, a 'shortcut' method is available which bypasses the lazy parsing for cases such as eagerly requiring the head of the data. This framework, titled LazyResult, is implemented as a Python class and shown as pseudocode in Listing 6.

```
class LazyResult:
    def generate_data_id():
        "generate placeholder for raw data"
    def register_lazy_data():
        "record the existence of some lazily parsed data"
    def retrieve_data():
        "fetch from cache or eagerly read"
    def evaluate():
        "uses retrieve_data to replace placeholders"
    def update_columns(columns):
        "signal the parser that these columns shall not be read"
    def update_rows(slice):
        "signal the parser that only this slice of data shall be read"
    def head(n):
        "bypass cache and eagerly read n rows"
```

Listing 6: Essential methods from the LazyResult class which intercepts evaluations, as pseudo-Python code.

A single interface is however not enough when considering the inability of parsing only file chunks for certain data formats. With tabular CSV data for example, it is highly difficult to implement an efficient single-column parsing method. It would

instead perhaps be more efficient to read everything and then select a column while caching the file in memory for subsequent column requests. Therefore, we propose two interfaces: 1) *LazyFile*, which behaves as a file opening handler which shall provide access to the file itself, and 2) *LazyData*, which encodes a chunk of lazily parsed data, typically a column. In conjunction, these interfaces are general enough to handle most lazy parsing cases effectively and are shown in Listing 7. The design of LazyFile further allows one to not use it when not relevant. Note that improving the performance of existing file parsers is beyond the scope of this research; the work is focused on extending existing parsers to lazily read data.

```
class LazyFile(object):
    def read_metadata():
        "parse the header/metadata of a file"
    def read_file():
        "read the file; usually a handler to retrieve columns from"

class LazyData(object):
    def lazy_skip_columns(columns):
        "lazily record that these columns shall not be read"
    def lazy_slice_rows(slice):
        "lazily record that only this slice of data shall be read"
    def eager_read():
        "parse and return the raw data as lazily recorded so far"
    def eager_head():
        "return the head of the raw data ignoring slice and skip"
```

Listing 7: Interfaces to implement when writing lazy file parsers.

Each evaluation step triggers the replacement of the placeholders with raw data from file, even if it is the same input data. To avoid reading it multiple times, a simple cache is implemented to keep the data in memory for subsequent reads. Given the speed of reading from file versus memory, the execution speed shall be improved, however at the cost of using more memory. In the worst case scenario, all the file data will be brought into memory which poses challenges when there is not enough memory available in the machine. To aid in this scenario, a flag could be used to disable caching all-together or perhaps only specific inputs. A lazy NetCDF parser is implemented with this framework to test its performance and demonstrate the framework usage.

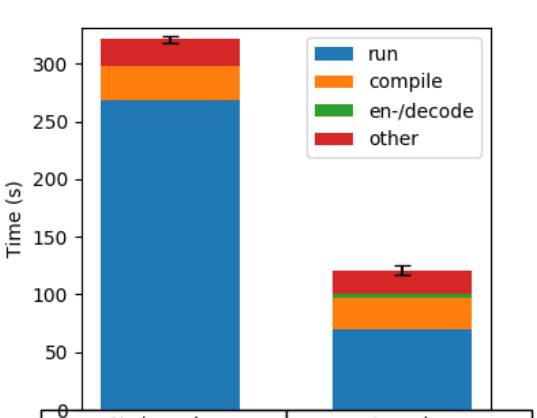
4.4 Results & Evaluation

To identify whether the proposed optimizations are worthwhile, we prepare small separate experiments comparing the benefits brought by intermediate results caching and lazy parsing. These experiments entail running the Pandas_weld implementation of the data analysis pipeline with different configurations. Due to memory limits, the intermediate results caching experiment is done on a smaller dataset, namely data_3, while lazy parsing is done on the larger data_12.

The framework works with both lazy and eager file parsers, therefore testing non-lazy file parsing is done by using parsers which do not use the framework, i.e. placeholders, but return the raw data eagerly. Caching can be disabled by setting an environment variable which acts as a flag within LazyResult which handles all evaluations. Caching of intermediate results can similarly be done with a flag for both the framework and the libraries implementing the cartesian product and the join. The libraries hence contain two implementations for each of these operators depending on whether intermediate result caching is possible or not.

Intermediate Results Caching

Figure 4.8 shows the mean total execution time split into the total time spent in each phase of execution as output-ed by WeldObject. The time spent in Weld compilation, en-/decoding data, and during Weld module run is subtracted from the total real time with the remainder being labeled as other. This remaining time is mostly time spent in Python and calling the parser libraries. One can notice that the total time is significantly reduced when caching the intermediate results, i.e. the cartesian product and the join indexes, by approximately 63%. The time spent in Weld computation is reduced because Weld is not required to recompute the intermediate results in the subsequent evaluation steps. The time spent en-/decoding is slightly higher, though, because decoding ndarray's of 2 dimensions, i.e. the intermediate results, requires data copying.



	No-ir-cache	Ir-cache
Weld	268.78	69.74
compile	29.42	27.72
en-/decode	0.02	3.61
python	22.91	20.13
total	321.13	121.21

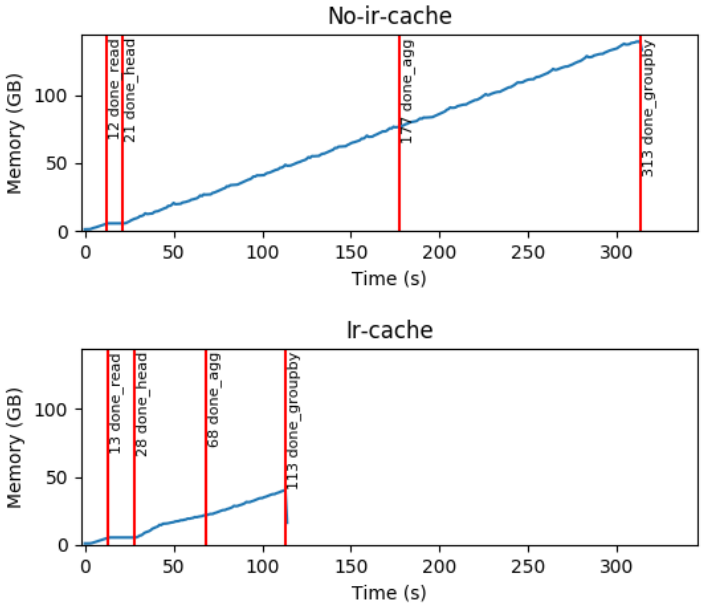


Figure 4.8: Mean execution time of Weld pipeline (left) along with memory profiling over a single execution (right) with and without intermediate results caching over data_3.

The memory usage over time of a given run, as shown in Figure 4.8 on the right, points out the drastic reduction in memory usage (and execution time) along with

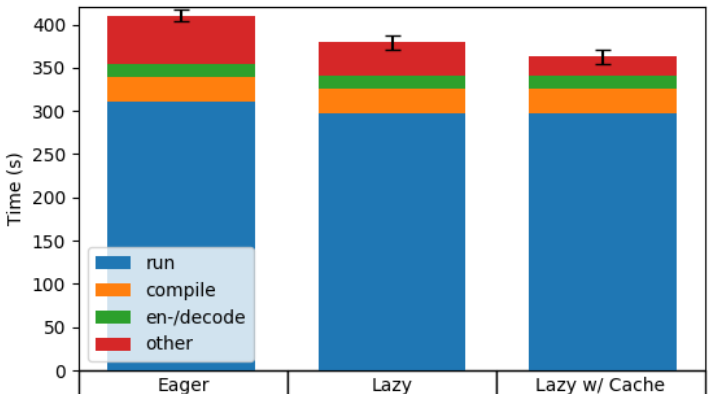
markers indicating when each step of execution has finished. The reduction is over 60% and one can notice the continuous rise in memory usage as intermediate results are re-computed. The higher memory usage of decoding the intermediate results can be seen as the slightly steeper line between done_head and done_agg in the ir-cache graph.

As expected, reading the data takes the same time regardless of intermediate results caching and the markers occur earlier than the no-ir-cache version. However, head takes longer when caching which could be explained by the Weld computation graph being sliced into two and compile twice instead of a single time to obtain the final result. Furthermore, the intermediate result, even if evaluated, is not cached during head execution which is something that could be improved in future work.

Since caching intermediate results presents great benefits in both time and memory usage, it is enabled by default in the experiments next. It would also not be possible to evaluate higher datasets than data_3 without it due to memory limits.

Lazy File Parsing

Lazy file parsing done comes in two flavors: first, parsing data from file whenever necessary, and second, caching the data in memory once parsed. These are denoted as *Lazy* and *Lazy w/ Cache* in the following graphs. Figure 4.9 shows the mean total execution time over data_12. The graph indicates an approximate 7% speedup of lazy parsing over eager parsing, and a slightly higher speedup of approximately 10% when also caching the parsed data. Recall that the pipeline requests for 2 columns out of 10 (16 including dimensions) to be skipped, along with selecting approximately 23% of the rows. This amounts to selecting only 20% of the total data. Given so much less data, one would expect a larger speedup.



	Eager	Lazy	Lazy w/ Cache
Weld	310.59	297.86	297.94
compile	28.13	27.96	28.16
en-/decode	14.98	14.50	14.56
python	56.60	39.06	22.13
total	410.29	379.38	362.79

Figure 4.9: Mean execution time of Weld pipeline over data_12.

The biggest reduction in time is seen in the 'other' section of the graph which includes time spent in Python and I/O. The reduction is from 56.6s to 22.13s which indicates a drop of 60.9%. This is indeed close to the expected reduction in time spent in I/O, i.e. 80%. Time spent during Weld module run is of course also reduced due to less data to process.

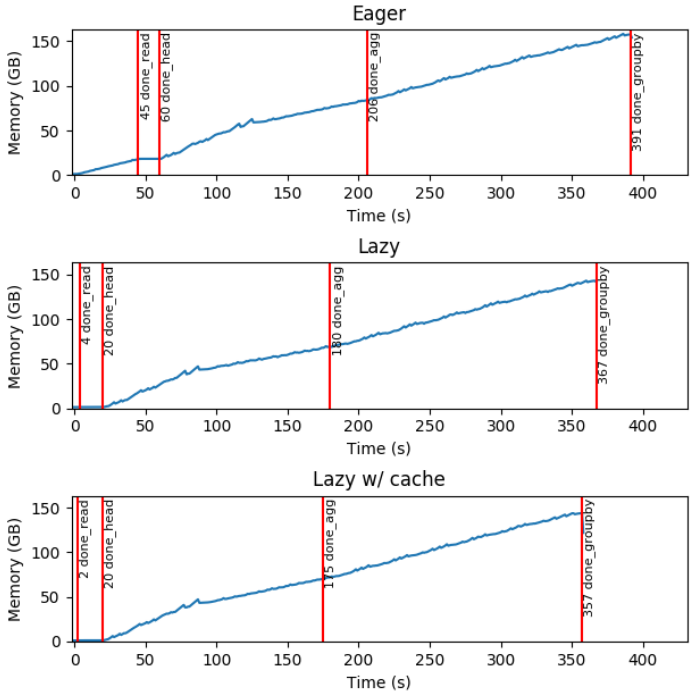


Figure 4.10: Memory usage over time of an arbitrary execution data_12.

Memory usage over time of a single execution is shown in Figure 4.10. The markers show that getting past the data reading point is faster since no data is initially read from file. Without caching the read data, one can notice that both the aggregation and the groupby take longer time because the same data is parsed multiple times from file. Unfortunately, not caching the data still presents a similar memory usage. This could be explained by a skew in the graph, i.e. the input data is only a small fraction of this memory usage. Given data_12 is approximately 34.7GB as CSV, the estimated memory usage of the input data is less than 7GB.

The results of eager and lazy parsing are summarized in Figure 4.11, where Lazy includes caching. There are benefits in all dimensions of time, maximum memory, and I/O, however only I/O seems significant. It drops to almost the expected 20%, however not exactly. The NetCDF parser sometimes reads slightly more data than the requested subset. This happens, for example, when the slice boundaries are not exact multiples of the dimensions of the data. For data_0, the parser reads 0.8% more data for each data column. The reduction in I/O does not seem to cause a big reduction in time execution. This may be because the hardware used in the experiment provides good read speed. However, when using older hardware, the differences are expected to be more significant.

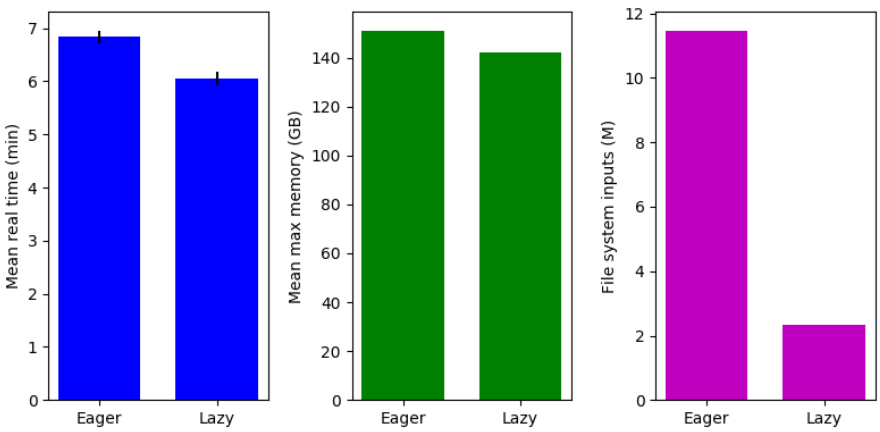


Figure 4.11: Mean time, memory, and I/O for execution over data_12.

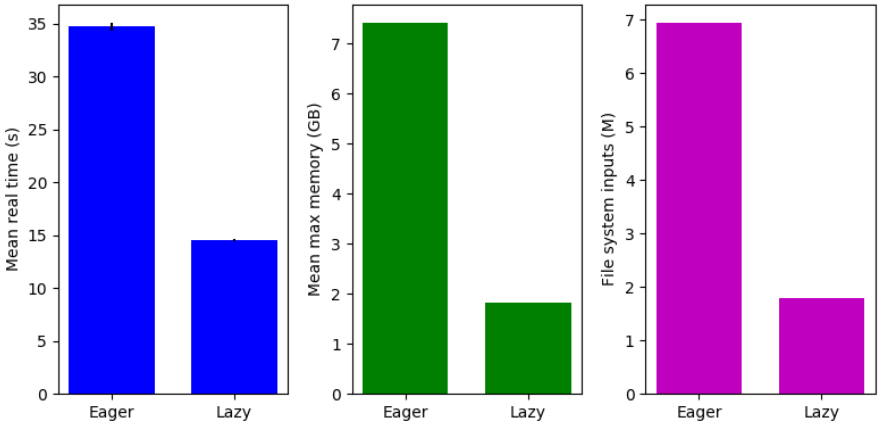


Figure 4.12: Mean time, memory, and I/O for small experiment.

The reduction in memory usage and time does not seem significant, though this might be due to the entire pipeline measurements overshadowing the actual benefits brought by lazy parsing. Therefore, a final small experiment is devised as follows: data1.nc of data_12 is parsed (3.47GB as NetCDF), 2 out of its 6 data columns (9 including dimensions) are dropped, and 24% of the rows are selected. The requested data would thus be approximately 16% of the total file size. To trigger evaluation, all data columns are summed. The results can be seen in Figure 4.12. The differences are now more noticeable with total execution time improving by over 50%. Max memory usage and I/O both improve by over 75%. The percentages of 16 and 75 do not add, indeed, to 100. The difference is attributed to the other overheads of execution, such as object wrappers around data.

Discussion

The caching introduced here to Weld is, indeed, quite similar to the Spark *cache* operator [40] or the recycler introduced into MonetDB [41]. Caching poses the threat of higher memory usage, possibly over the maximum allowed. Spark handles this by making the cache operator behave only as an indication that a result might be

used again. If there's enough memory, the result is maintained otherwise the cached object would be recomputed. This is possible because the computation graph is maintained. The recycler in MonetDB functions in a similar way by maintaining a recycle pool which is periodically cleaned based on the least expected utility estimation. The Weld caching introduced here does not go as much in depth as it does not use any policy to evict cache entries when necessary.

Caching intermediate results poses the drawback of breaking/cutting the Weld computation graph whenever the cache operator is used. This means that Weld will no longer be able to perform optimizations across the entire graph but only within the smaller chunks. As was shown in this chapter through the cartesian product and the join, caching intermediate results brings benefits by reducing the amount of re-computation upon evaluation.

There are a few drawbacks to the proposed lazy parsing framework and its current implementation. It is unclear if this approach scales well with multiple libraries. Using non-NumPy data also requires extending the methods in the framework. This could be improved by generalizing a step further and providing extension points which could be implemented by the libraries using these different data representations. Lastly, the current implementation tracks the file inputs which means that when one subsets an input, the subset applies everywhere this input is used. Hence, different subsets of the same input are currently not possible. Adding this functionality requires tracking which objects use which subset. This is difficult given the current chaining of WeldObjects through dependencies. For example, given object A as a dependency to objects B and C. B requires a subset of A while C requires the full data. If B stores that only a subset is to be read from file, C will not receive the entire data. Possible solutions include duplicating A, as shown in Figure 4.13, or storing the slice and columns-to-skip in each object.

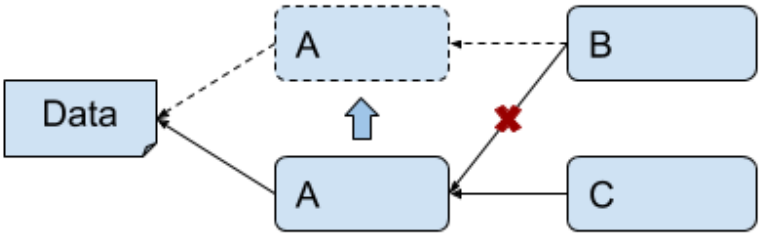


Figure 4.13: When two objects have the same dependency but require different subsets of the data, the dependency could be duplicated.

Related approaches to efficient file usage are found in literature. Noga et al. [42] formalize the domain of tree-based XML processing and introduce a lazy XML parser. The parser is required to first analyze the XML file to infer the structure and to build an internal representation suitable for partial parsing. The user is then able to select only subsets of the data which are cached into memory. The need to first analyze the XML file to determine its structure is similar to the requirement shown in the LazyFile interface above through the method read_metadata. By using the framework, parsers essentially build a similar internal representation of the files from file metadata.

The NoDB paradigm in database-systems [43] makes raw data files as first class citizens. Queries are then possible without first loading all the data; the database-system retrieves from files just the subset of data required to execute queries on-the-fly. Once read, data is also temporarily kept in a cache. The authors implement a NoDB version of PostgreSQL and observe competitive performance against other DBMS. Parsing is optimized to the data required by a specific query. This is indeed similar to the LazyData interface above which tracks which subset of the data is required. The framework introduced in this work, though, is tailored to exploratory analysis, as shown through the LazyFile.read_metadata and LazyData.eager_head. It also abstracts a level further by allowing multiple frameworks/libraries to use it simultaneously. However, the current implementation presents the drawbacks described previously which make it less flexible than expected by the NoDB model.

Chapter 5

Experimental Setup

The main comparison shall be done by implementing the previously introduced data analysis pipeline in 6 different languages / frameworks: Python, Weld, Julia, R, Java, and Spark. Since the pipeline targets all operations as a whole, it will provide a better overall insight into the performance brought by these frameworks, at least in their current developmental stage.

The experiment has as independent variables the subsets mentioned before and the pipelines as factors. For each pair, the dependent variables are the measurements of execution time, memory, and CPU, both total and during execution through profiling. Several other markers are adopted, where relevant and possible, to delimit pipeline execution progression along with details such as disk I/O. Furthermore, note that the implementer could also be considered a factor for a typical data analyst in these experiments; such a user cannot be expected to know all the inner workings but only perhaps basic optimizations across the respective implementations. Therefore, unless the public-facing API allows configurations for, e.g. which join algorithm to use, no further optimizations have been done.

This chapter consists of the comparison experiments done in this work and is divided as follows: Section 5.1 details specific implementation details for each language. Each framework has its own data representation and features available by default and thus might require different levels of configuration. The results of the pipeline comparison are shown and evaluated in Section 5.2 with further profiling and analysis in Section 5.3.

5.1 Implementation Specifics

The data analysis pipeline cannot be implemented everywhere exactly as described mostly due to API differences. Particularly, the parsers provide different behavior depending on the high-level wrapper and some operations are not supported by default by the respective API, therefore need to be implemented. The implementations are meant to work on a single core of the CPU and use as much memory as required, with the exception of Weld and Spark which feature 2 variants each, non-parallel and parallel. The parallel versions are configured to use all the machine cores, i.e.

32. All implementations, apart from Spark, feature a column-store. Technical details such as software versions are available in the Appendix. Following are details regarding the chosen languages and the specific implementations used in this work.

Python It features a dynamic type system and automatic memory management and can easily interface and call native C libraries. This means libraries can also be heavily optimized and hence be highly efficient. Furthermore, one may use Cython [44] which is a superset of the language designed to give C-like performance while writing mostly Python. Cython is often to speed up Pandas even more.

The Python implementation uses Pandas' DataFrame which is built on-top of NumPy. This allows Pandas to take advantage of certain NumPy features that lessen the load on the memory bus. Examples include: views to subset, filter, or transpose underlying data without copying, vectorization to speed up array computations, and broadcasting to efficiently expand data without creating intermediate arrays. The core functionality of NumPy is written in C and also relies on the BLAS [45] specification for efficient linear algebra computations.

Creating Pandas DataFrames from NetCDF files poses certain challenges. The go-to method would be to use the Xarray[1] library which can parse and convert the files to Pandas DataFrames. However, Xarray performs several extra steps which cannot be avoided: checking conventions, reading into its own memory format, and most importantly, changing the order of the dimensions. This is internally an efficient NumPy transpose which is however an extra unnecessary step which is hard to replicate in the other implementations. Therefore, a method using the 'raw' Python wrapper over the NetCDF parser has been implemented to convert straight into a Pandas DataFrame while maintaining dimensions order.

Weld The implementation of the pipeline in Weld has been detailed in the previous chapter through the use of Pandas_weld. However, it is worth mentioning that a valid Weld module could also be instead written from scratch by hard-coding a very large (and thus optimized) Weld IR for the pipeline. This, though, provides a much higher difficulty to implement and to test. Furthermore, it is not true to Weld's design for multi-library support.

Julia Julia's DataFrame implementation does not offer an API similar to Pandas that can compute the aggregations, therefore it needs to be implemented. The Julia NetCDF parser package is a low-level wrapper over the C API which also requires some extra work, e.g. the cartesian product, to reach the format required by the pipeline. Lastly, it was observed that some methods from Julia require more care when using. First, the Julia *repeat* method which was initially used to compute the cartesian product turned out to be very slow. By implementing it manually with for loops, the execution was improved by over a factor of 2. Second, by default, slicing Julia arrays returns copies of the sliced area instead of a mask over the initial data. This is unlike NumPy or other libraries. To overcome this, one must use the *view* method.

R The first stable version of R was released in 2000 as an implementation of the S language for statistical computing and graphics [46]. One is able to use its REPL with semantics closer to functional programming languages such as Lisp. R features

[1] `http://xarray.pydata.org/en/stable/`

the `data.frame` abstraction out of the box which together with the *dplyr*[2] provides all the tools necessary to implement the data analysis pipeline.

Java The best case scenario optimized pipeline has been implemented in Java which was chosen due to the friendlier yet still efficient programming environment, as opposed to C. Java is strongly typed and verbose, therefore significantly more code is required to implement the pipeline and also justifies the lack of an existing viable DataFrame implementation. Therefore, the Java implementation involved writing a small DataFrame structure with only the operations required to run the pipeline. There are no shortcuts in the operators themselves, however support for more operators or types than required has been foregone. The aggregations, though, have been coded as efficiently as possible while the join is restricted to a merge join.

Java features a functional Stream[3] API which could be used to parallelize work, however does not provide an API for floats but only for doubles. Hence, using Streams would requires casting most of the data to double. Given that the implementation performed well anyway, Streams were not used, but are a potential avenue for future work.

Spark Apache Spark [40] is a unified analytics engine for large-scale data processing and is arguably the step forward from MapReduce with hundred-fold speedups. It is arguably the odd one out of the implementations due to its different design goals, particularly parallelization. The core abstraction is the Resilient Distributed Dataset (RDD) which are partitioned across an arbitrary amount of cluster nodes. RDDs then keep track of all computations such that when a fault occurs, the results can be recomputed at minimum cost. This fault tolerance, however, adds a fairly large overhead to computations and inter-node communications. This makes Spark a questionable choice when running computations on less than big-data or on newer machines which can contain hundreds of GB in memory.

Spark's operations are of two kinds: 1) *transformations*, which are lazy operations executed in parallel, and 2) *actions* which trigger result computations. This distinction allows a clear separation between cheap and expensive operations on data, with actions typically requiring data communication between nodes, i.e. *shuffling*, which furthermore involves a degree of synchronization overhead.

The RDD has more recently been wrapped in a table-like data structure, the *DataFrame*, along with an extensible database-like optimizer titled Catalyst [10]. The DataFrame extends the functionality of RDDs with relational processing operations which are then optimized through Catalyst. The API is also more expressive as one is able to easily select columns by name and do operations. Catalyst applies relational optimizations at both the logical and the physical plan by carefully modifying the tree IR of queries while maintaining the fault tolerance and lazy API of RDDs. Furthermore, through project Tungsten, Spark benefits from better memory management and cache-aware computation.

Spark features here a less than ideal pipeline implementation in Scala as its row-store design makes it difficult to read columns from files and to create a single DataFrame. The columns have to be read separately and parallelized to RDDs while merging into

[2] `https://dplyr.tidyverse.org/`

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`

final large RDDs which can finally be converted to DataFrames. It is desirable to operate on DataFrames due to Catalyst which shall optimize the evaluation steps as much as possible. An alternative would be to load all the data in the driver's memory as Scala data structures and then convert to RDDs and DataFrames. However, this is an inherently flawed approach as the data size used in a Spark application is expected to not fit in the memory of a single machine. It was, though, attempted, but proved to be slower. The pipeline implementation shall be general enough that it would work as-is when using an actual cluster, hence showcasing the big data handling capabilities that the other implementations cannot achieve.

It is expected to run Spark jobs through the *spark-submit* script which handles script distribution and other preparation steps. One can set the master to a cluster scheduler, such as Apache YARN[4], or even to a pseudo-cluster master which is the local machine. This second option is arguably only for testing purposes however is expected to provide an effective way to run Spark scripts without an actual cluster setup.

Parallelization brings a lot of overhead particularly when Spark is forced to run in less-than-ideal conditions, namely as a standalone local cluster on a single CPU, as required in this work. Submitting the job requires careful tuning work to make sure the overhead is limited while extracting the highest performance possible. Some steps taken towards this goal include making sure both the RDDs and the DataFrames are partitioned in the same way which aids with the actions which require shuffling, including the join and groupBy. Careful use of caching also ensures that important intermediate results are not dropped and can be immediately used in the following steps. The configuration parameters are available in the Appendix.

5.2 Results

The pipeline comparison results are presented in this section, first by analyzing the `time` outputs for total time taken to execute the pipelines, and then by looking at memory and CPU usage over time profiled through collectl. Lastly, an investigation into scalability is made by comparing the performance over the different input sizes.

Execution Time

Figures 5.1 and 5.2 show the total execution time of the pipelines over input datasets data_0 to data_25, averaged over 5 runs. One can immediately notice a pattern: Java is the fastest and Spark is the slowest. Java, as the ideal implementation, is expected to be the fastest while Spark is expected to be slow due to the overheads described in the previous section. However, the difference between Spark and the other implementations exponentially increases with higher datasets such that Spark has been omitted from data_3 upwards. This behavior shall be evaluated in Section 5.3.

[4] `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

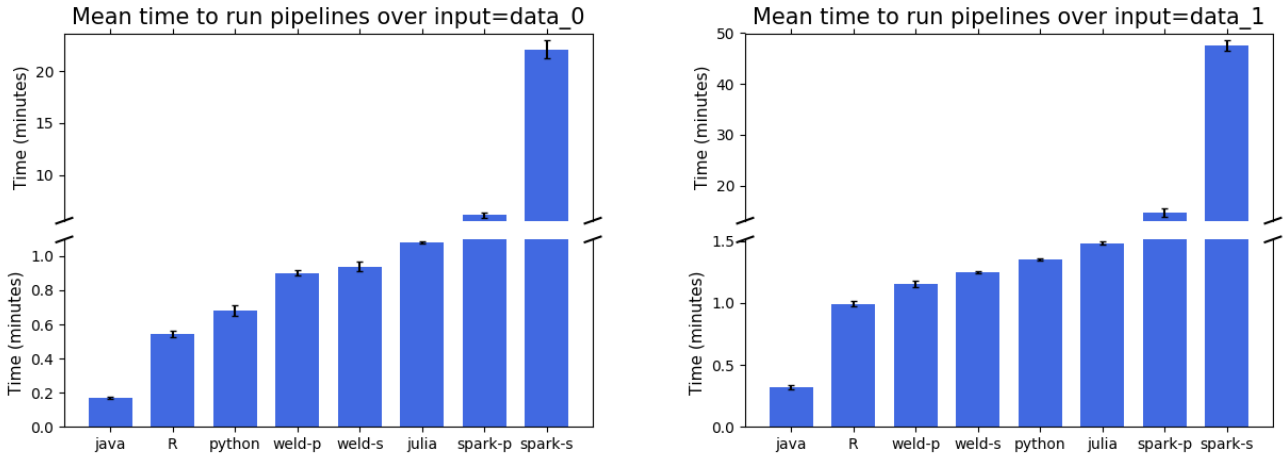


Figure 5.1: Barplots for mean total execution time of the pipelines over the different inputs.

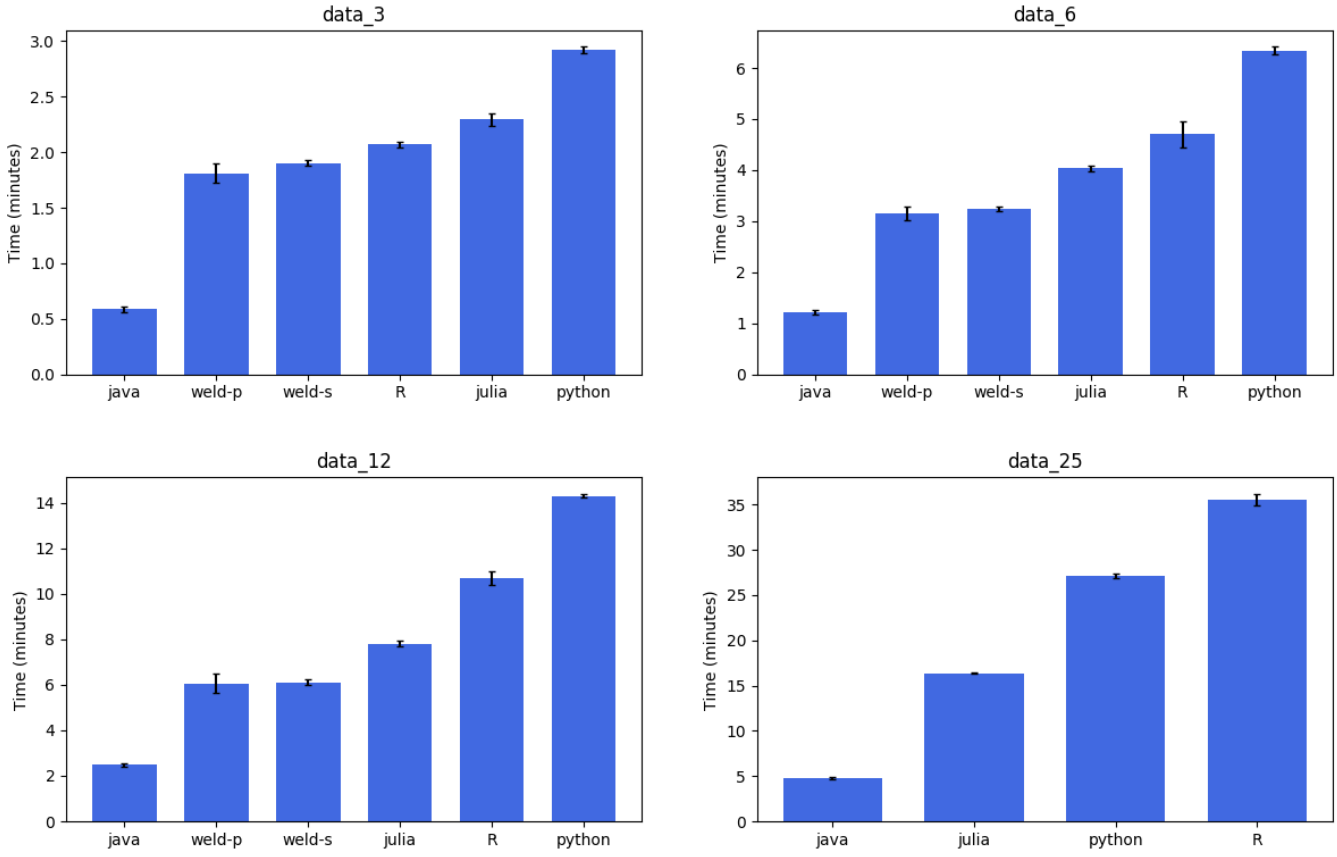


Figure 5.2: Barplots for mean total execution time of the pipelines over the different inputs.

Note that from data_3 (1.45/8.67 GB raw/csv), Weld takes the second spot indicating that it does identify and apply optimizations to the pipeline. Some examples are available in the Appendix. The parallel version is also always faster but by a small

margin; with small datasets, the cost of spawning and synchronizing threads seems to outweigh the benefits brought by parallelization. Julia starts by being the slowest but overtaking Python from data_3 and also R from data_6, after which it stagnates. This means that the optimizations brought by LLVM outweigh the compilation time only with higher datasets.

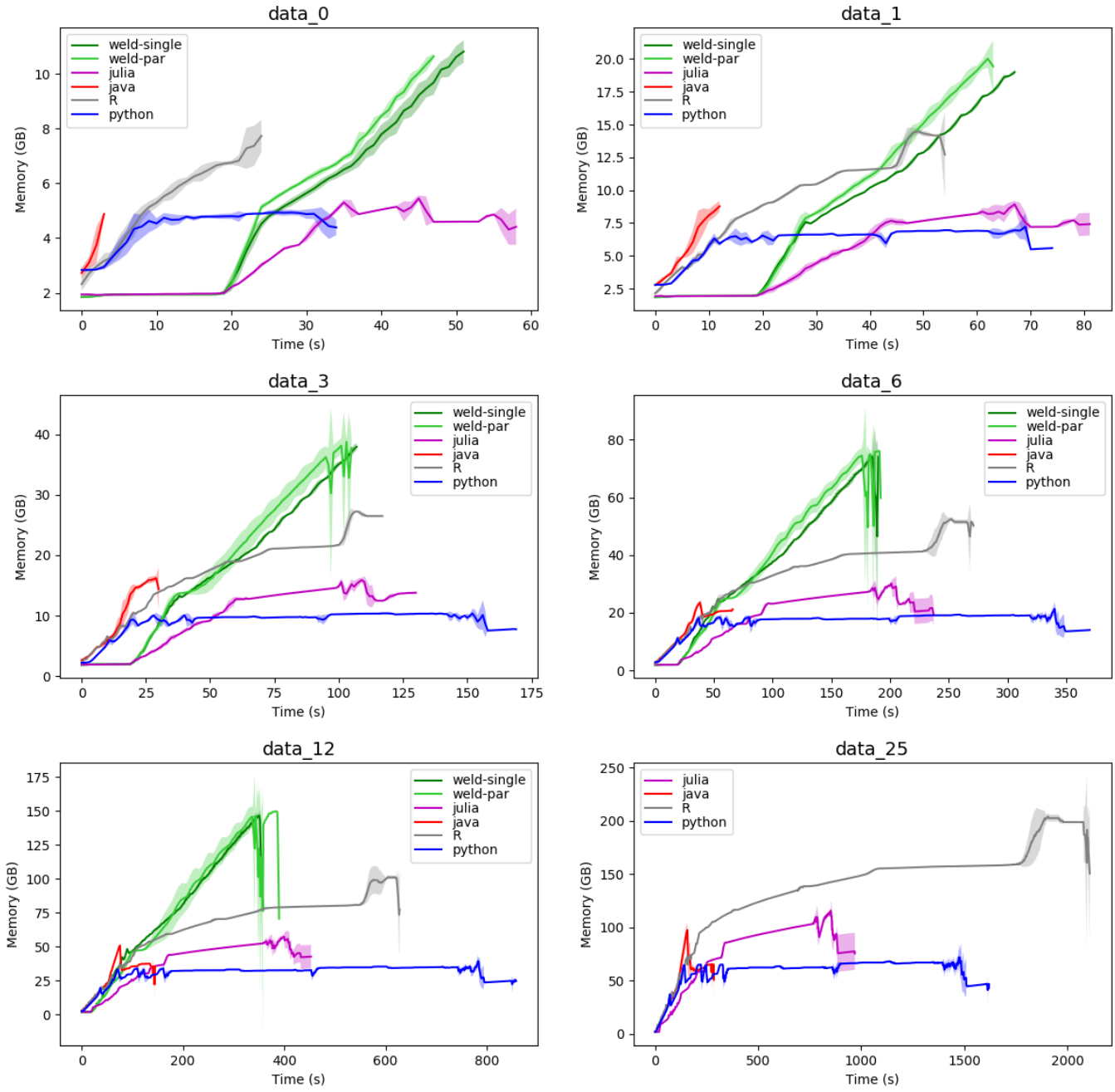


Figure 5.3: Mean memory usage scatterplots of the pipelines over the different inputs.

One might notice that the Weld pipeline is missing from the data_25 plot which is due to it requiring more memory to compute the groupby result than available. Only the Python and Java pipelines succeed on data_50 and none on data_100 which

is perhaps expected given their sizes as CSV (approx. 139 and 278 GB, respectively) which are a fair estimate of the amount of memory required to just load the data. The other implementations require more memory than available.

Memory Usage

Figure 5.3 shows the mean memory usage over time for inputs data_0 to data_25, averaged over 5 runs and with shaded area to represent the standard deviation between the runs. The negative gradient lines after the groupby are leftovers as collectl continues to profile until the pipeline process has exited; this might take longer than 1 second due to, e.g. garbage collection. Weld consistently ends up using up to 30% more memory compared to the other implementations. One reason is that all the parsed data is kept in memory along with intermediate results while the other implementations drop the unnecessary results, typically through garbage collection.

R consistently requires approximately 15% more than Python/Julia/Java, hence being in the middle between these and the Weld implementations. The lazy parsing feature of Weld can be observed in the plots as the memory usage is horizontal up to a certain point where evaluation is triggered and data is parsed. Intermediate results are also being materialized from that point onward. Interestingly, Julia also has the same period of constant memory usage which could be explained by it using the same NetCDF parser internally.

The efficiency of NumPy and Pandas implementations can be observed as the python-libraries pipeline consistently uses the least maximum memory. This can be explained by the NumPy features detailed in a previous section which are seamlessly integrated with Pandas. After parsing, the Python pipeline seems to not increase in memory usage. Spark has been omitted from the graph as it could not be run on datasets higher than data_3, as previously mentioned. Its memory usage was twice of Weld's.

CPU Usage

CPU usage is collected during profiling as a percentage of all the 32 cores, therefore multiplying the recorded number with 32 returns a percentage measure of a 'single' core. Single threaded programs are expected to execute at close to 100%, therefore all except the parallel pipeline implementations. Figure 5.4 shows these plots over each input size averaged over 5 runs for data_0 to data_6. Spark, however, is omitted due to the significantly larger execution times and memory usage which would greatly skew the graphs. The shaded area represents the standard error of the mean which better emphasizes the differences between the CPU usage. All pipelines seem to have a peak over 100% at the beginning of the execution which can be explained by the interfacing with the low-level NetCDF parser. This, however, is a systematic error as all the parsers in these implementations are built on top of it.

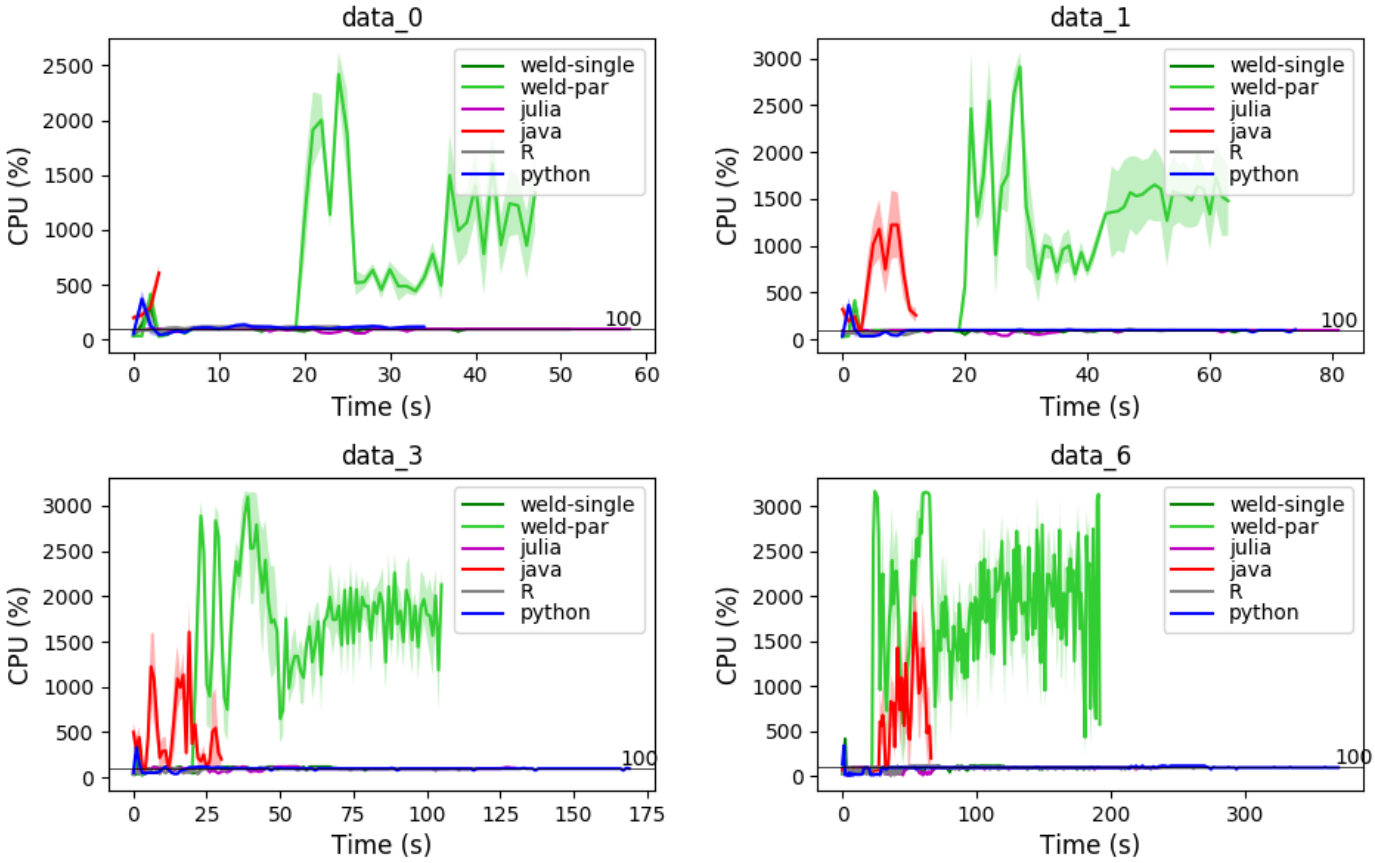


Figure 5.4: Mean CPU usage scatterplots of the pipelines over different inputs.

Note that Java seems to always use more than 100% of the CPU. This can be explained by Java spawning a JVM process 'in parallel' (through the fork+exec pattern) but which is still tracked together with the actual pipeline execution by collectl. JVM handles most importantly the garbage collection and translation of Java bytecode to machine code, processes that can happen in parallel. Weld-par can clearly be seen parallelizing the work up to all the 32 cores, hence also having a large variation in CPU usage over time.

Scalability

To understand how well the implementations scale over the input sizes, the mean execution time over 5 runs for each pipeline from data_0 to the maximum input size that succeeded is plotted in Figure 5.5. The input sizes follow a power curve and thus, to have a measure over scalability, the curve has been plotted along the barplots. The curve assumes that each consecutive input takes twice as long as the previous one, a phenomenon which can be seen in Java's graph. This means that for Java, the execution time is linear to the input size; however, this is not the case in the other implementations. Both Python and R start to take up to twice as much time than expected from data_12 upwards. On the other hand, Julia and Weld both scale very well as the datasets take significantly less time to execute than hypothesized. What they have in common is LLVM which seems to speed up the

computations through features such as vectorization. Weld furthermore contains its own optimizations over its IR which speed up the overall total execution time.

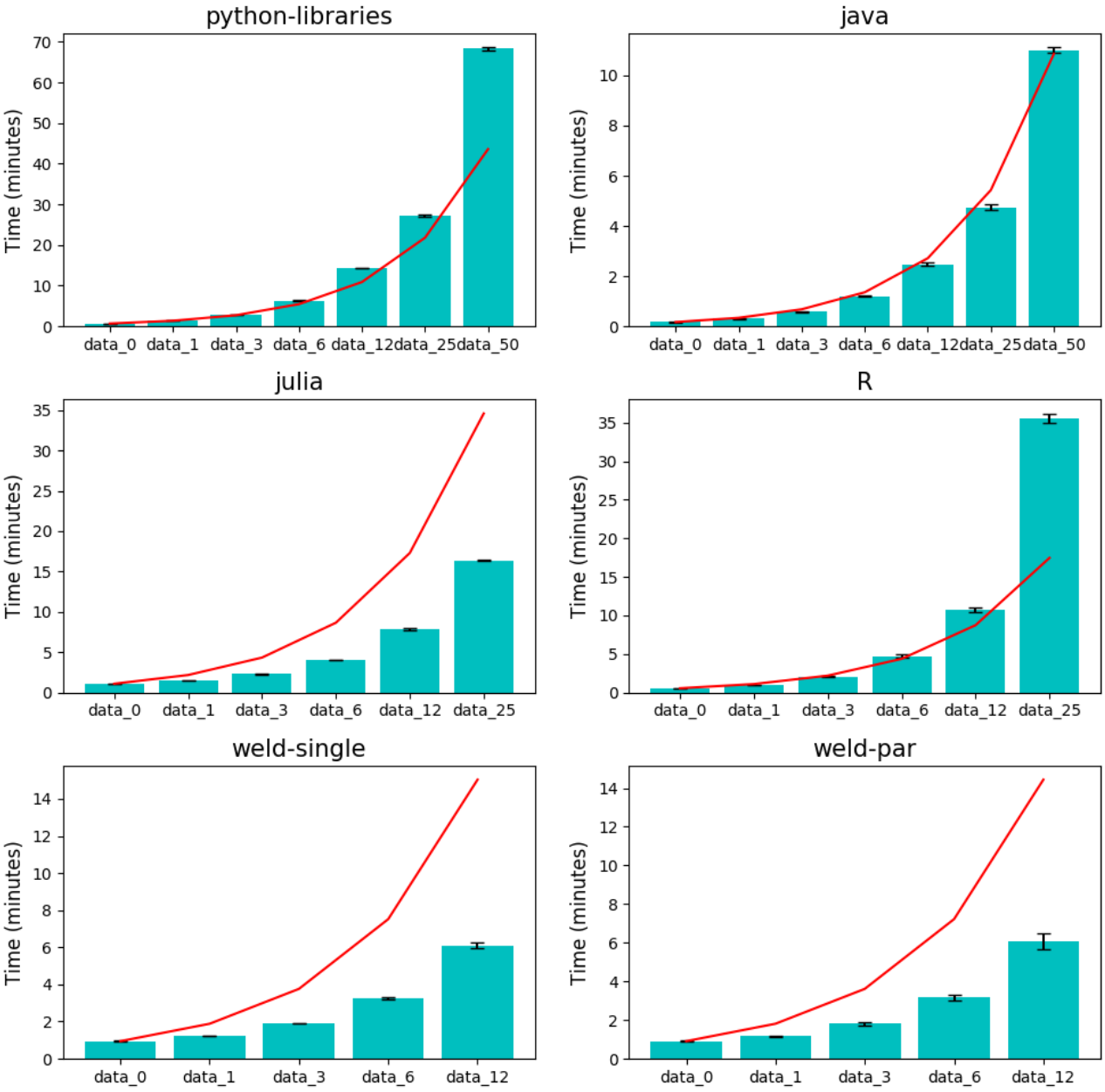


Figure 5.5: Barplots for mean total execution time over different inputs for each pipeline.

The same plot has been made for the maximum memory used (as reported by `time`) with a curve over the CSV input sizes. These sizes are a forgiving estimate of the amount of memory that would be required to store the data. Similar to the previous plot, the curve assumes that each consecutive input uses twice as much memory than the previous one. The barplots shown in Figure 5.6 reveal in a decreasing order the scalability of each pipeline implementation. Python showcases the ideal memory usage as it uses almost exactly as much memory as expected. This is

due to the implementation over NumPy which limits memory usage, as described before, however comes at the cost of less scalability in terms of execution time. The Java measurement might be misleading due to a peak in memory usage, as shown in the memory profiling graphs before, which in fact drops after the aggregations are completed. Julia and Weld do not scale well in terms of memory usage which contrasts with the high scalability in terms of execution time. R fails to scape well in terms of memory usage; together with less scalability in terms of execution time but better execution times on small datasets indicates R is the best option for small datasets.

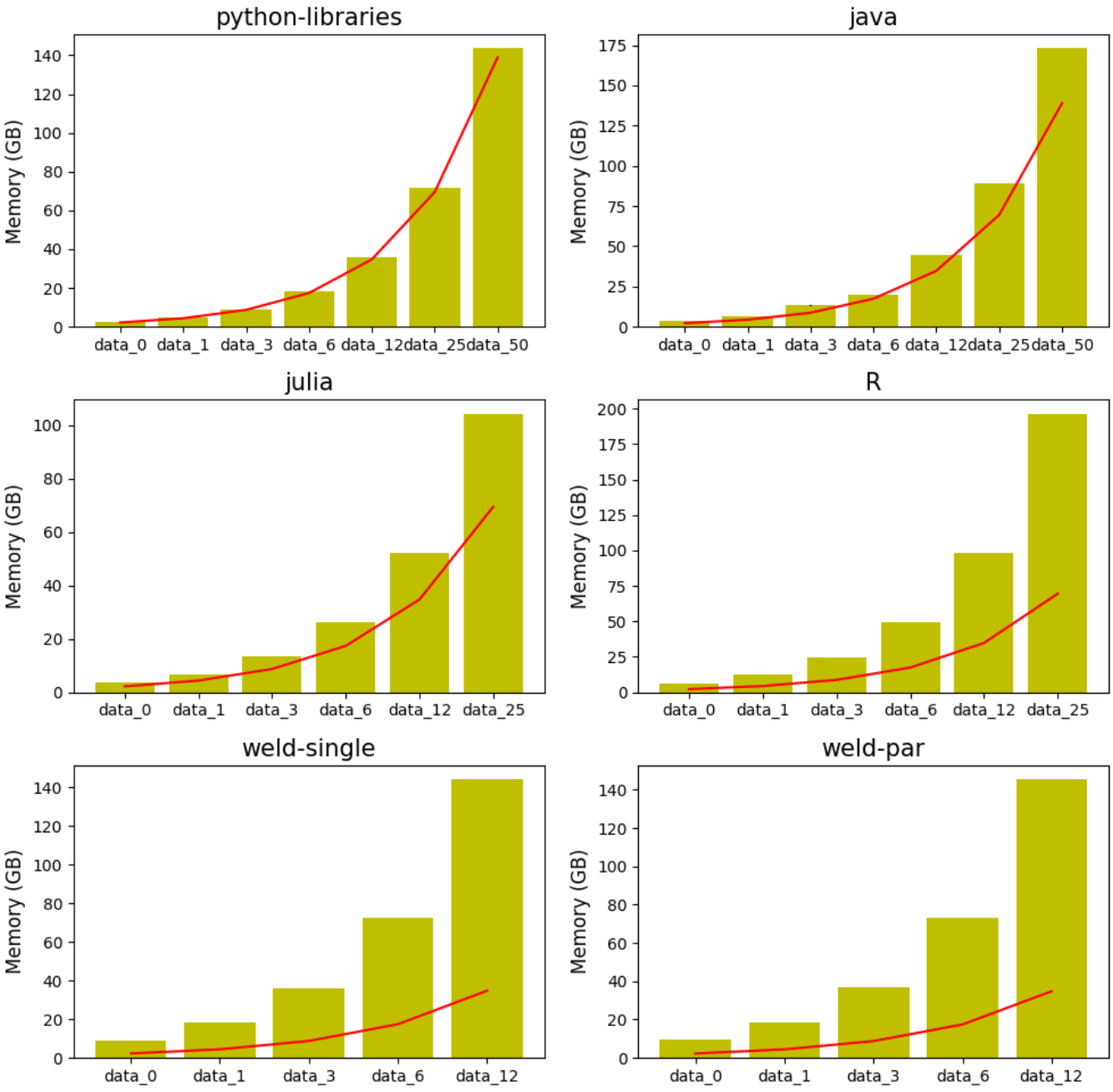


Figure 5.6: Barplots for maximum memory usage over different inputs for each pipeline.

5.3 Discussion

Implementing the pipeline was easiest and fastest with Python and R, and also Weld given it mimicking the Python version. The performance was good, however lacking with larger datasets where they fail to scale well. The optimal implementation in Java succeeds in outperforming all other implementations significantly. This came at the cost of manually handling types and implementing all operations. The trade-off between productivity and performance is again evident when comparing these implementations. Weld and Julia, in contrast, shine when handling larger input data by scaling well, however at the cost of more memory usage. They are both not mature enough to handle everything well and particularly Weld could be improved, as hinted throughout this work. Despite these drawbacks, Weld is the most promising given the best performance after Java and the possibility of user-friendly syntax. It is usable through multiple libraries and programming languages, and also by supports parallelization.

All implementations seem to suffer by being CPU-bound because the CPU usage is always close to maximum throughout pipeline executions. This indicates that the biggest benefit next could be brought by efficient parallelization, something which Weld and Julia were built to support. None, however, can scale out as much as Spark while maintaining a user friendly API; the poor behavior when ran on local is investigated in the next section. Best memory usage could be observed in Python with Pandas/NumPy due to the now mature and optimized library. R proves to be the best option on small datasets with good execution time; with larger datasets, a different language/library should be used.

Problems of Spark on local

Spark is unexpectedly extremely slow when run on local, i.e. a single machine, due to what seems to be inter-thread communication. As seen in the stacktraces flamegraph in Figure 5.7, approximately 60% of the time is spent in *epollWait* which is required when piping data between threads or processes. This happens because despite Spark running on a single machine and in the same JVM, it still behaves almost as if on a cluster of machines: there are still separate threads (processes in a cluster) for the driver, the executors, the master, and the scheduler which must all communicate with each other. Coupled with the overhead of partitions and RDD lineage, this proves to be very slow. While this performance could be part of a design choice to allow testing a full Spark application locally, one would still expect relatively good performance. Nevertheless, Spark is clearly not the best option when all data can fit on one machine.

Weld

Besides the two extensions presented in the previous chapter, there are certain other details that could improve Weld. The en-/decoding framework is fairly restricted in its current state and could use more Python-Weld mappings provided by default. Currently, for decoding a Weld result, it is only possible to en-/decode a vector

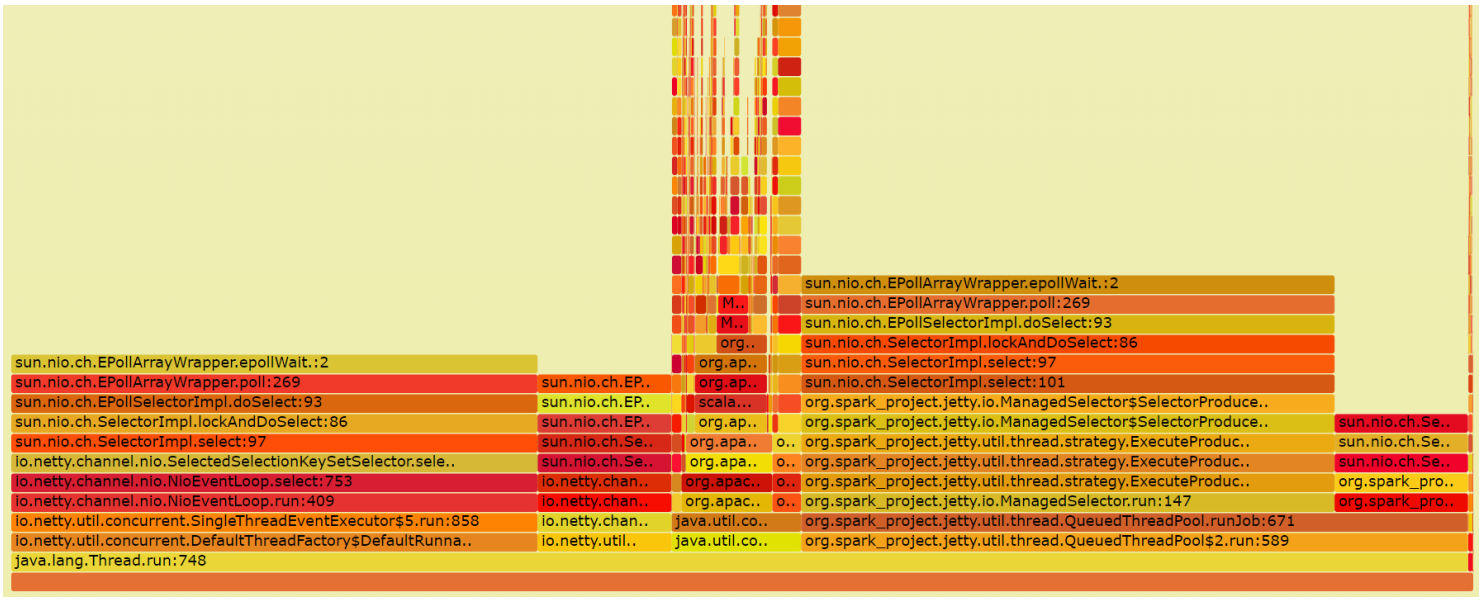


Figure 5.7: Spark flamegraph over single executor pipeline execution on data_0.

to/from NumPy arrays; it is not possible to obtain a Python dictionary. Having a direct mapping between a Python dict and the Weld dictionary could prove quite useful. The dictionary implementation in Weld also does not allow for common operations in its IR, such as extracting the keys or the values from it. A dictionary cannot be evaluated into something more usable unless the *tovec* operator is used which converts it into a vector of structs. Integration with NumPy could also prove beneficial as it is a mature framework with its own optimizations. Weld could hence natively support NumPy features such as broadcasting without the need to encode them in Weld IR.

Chapter 6

Conclusion

Implementations in lower-level languages, such as Java in this work, are very difficult to overcome, however come at the cost of more implementation hours. Furthermore, the entire process from programming to machine code and execution typically relies on optimizations specific to the language-family, such as the GCC compiler for C. Attempts to generalize and create common frameworks featuring optimizations are very attractive to the programmer however add costs in interpretation and conversion. This is seen in both Weld, where Weld IR needs to be compiled and converted to LLVM IR, and Julia, where source code is JIT compiled for LLVM to find possible optimizations. The benefits are not visible when there is little data; as shown in this work, there need to be over a few GB of data for them to become faster than other implementations. The optimizations brought by them and LLVM end up scaling very well, despite up to twice the costs in memory, by being up to 2.5 times faster than Python. Nevertheless, both Weld and Julia and still in their infancy with less predictable behavior and many changes in the API.

For the time being, established libraries such as Pandas/NumPy (Python) and dplyr (R) prove to be more reliable with good performance especially for relatively smaller datasets, such as under 10GB. These datasets can easily fit into the memory of a single machine with no need for parallelization to obtain reasonable speed. However, it all depends on the scale of the data. When Python or R libraries start to slow down, it is time to switch to more optimized, scalable, and perhaps parallelizable libraries. When data cannot fit on a single machine, one has to switch to a distributed framework such as Spark. While MPI would prove to be more efficient, it comes at the cost of many more programming hours; Spark provides a clear and user-friendly syntax, though it was designed and relies on a large-scale cluster. When running on a single machine, despite having plenty of computing power available, it cannot be as efficient as expected due to the assumption of large-scale hardware and the requirement of fault tolerance that went into its design.

Weld was shown to have plenty of potential by closing in to the ideal implementation. However, there are still many possible improvements that can be done, such as intermediate result caching and more flexibility in its IR. Lazy parsing can bring great benefits in workflows involving subsetting of the input data while also speeding up the initial exploration of the data. The proposed framework generalizes such that multiple libraries and parsers can be used while still maintaining good performance.

There are several ways to extend this work. First, while the data analysis pipeline introduced here contains a good variety of operations, more iterative algorithms shall be added for a more comprehensive overview of library performance. Normalization of the input data or a machine learning model, such as a neural network, are great candidates for addition. Second, parallelization was overlooked with only Weld and Spark having a parallel version. Java could also be parallelized through its Stream API and Julia provides a parallelization framework through annotations. Furthermore, there are frameworks proposed in literature, such as HPAT, Pydron, and Dask, which aim to parallelize Python and/or Pandas. Lastly, there is work in running Spark more efficiently, such as through the Flare runtime. These, along with other frameworks, could also be compared on the data analysis pipeline.

Appendix

Repository

https://github.com/radujica/data-analysis-pipelines

Hardware Used

Machine with 32-cored Intel Xeon E5-2650 CPU, 256GB RAM, and HDD for storage.

Software Used

Pipeline	Software	Version
python-libraries	Python	3.6.5
python-libraries	pandas	0.22.0
python-libraries	netCDF4	1.3.1
weld	LLVM	6.0.0
weld	Python	2.7.14
weld	pyweld	*
weld	grizzly	*
java	java	1.8.0_171
R	R	3.4.4
R	readr	1.1.1
R	ncdf4	1.16
R	dplyr	0.7.6
R	tidyr	0.8.1
julia	Julia	0.6.3
julia	NetCDF	0.6.0
julia	DataFrames	0.11.6
julia	CSV	0.2.5
spark	Scala	2.11.8
spark	Spark	2.3.0
spark	netcdf	4.3.22

* Using old-working-state branch with latest Weld developers input from April 12 and minor adjustments by myself: https://github.com/radujica/weld/commits/old-working-state

Spark Configuration

Single: master=local, spark.sql.shuffle.partitions=4, spark.executor.heartbeatInterval=115, driver-memory=200g, RDD partitions=4

Parallel: master=local[32], spark.sql.shuffle.partitions=64, spark.executor.heartbeatInterval=115, driver-memory=200g, RDD partitions=64

Weld IR

Further examples of complete Weld functions (combined IR chunks) and their optimized IR from the Pandas_weld implementation of the pipeline.

Join

```
# Note that _inp17 and _inp9 are the cartesian products intermediate results
# for the indexes of each of the two dataframes
|_inp0: vec[f32], _inp1: vec[f32], _inp10: vec[f32], _inp11: vec[f32],
_inp12: vec[vec[i8]], _inp17: vec[vec[i64]], _inp2: vec[vec[i8]],
_inp9: vec[vec[i64]]| let obj100 = (_inp0);
let obj101 = (_inp1);
let obj102 = (_inp2);
let obj116 = (lookup(_inp9, 0L));
let obj117 = (lookup(_inp9, 1L));
let obj118 = (lookup(_inp9, 2L));
let obj119 = (_inp10);
let obj120 = (_inp11);
let obj121 = (_inp12);
let obj133 = (lookup(_inp17, 0L));
let obj134 = (lookup(_inp17, 1L));
let obj135 = (lookup(_inp17, 2L));
let obj136 = (
    result(
        for(
            obj116,
            appender,
            |b, i, n|
                merge(b, lookup(obj100, n))
        )
    ));
let obj137 = (
    result(
        for(
            obj117,
            appender,
            |b, i, n|
                merge(b, lookup(obj101, n))
        )
    ));
let obj138 = (
    result(
        for(
            obj118,
            appender,
            |b, i, n|
                merge(b, lookup(obj102, n))
        )
    ));
let obj139 = (
    result(
        for(
            obj133,
            appender,
            |b, i, n|
                merge(b, lookup(obj119, n))
        )
    ));
```

```
let obj140 = (
    result(
        for(
            obj134,
            appender,
            |b, i, n|
                merge(b, lookup(obj120, n))
        )
    ));
let obj141 = (
    result(
        for(
            obj135,
            appender,
            |b, i, n|
                merge(b, lookup(obj121, n))
        )
    ));

    let len1 = len(obj136);
    let len2 = len(obj139);
    # bool arrays shall be padded until maxLen so that result can be cached as np.ndarray of ndim=2
    let maxlen = if(len1 > len2, len1, len2);
    let indexes1 = {obj136, obj137, obj138};
    let indexes2 = {obj139, obj140, obj141};
    let res = if(len1 > 0L && len2 > 0L,
                iterate({0L, 0L, appender[bool], appender[bool]},
                |p|
                    let val1 = {lookup(indexes1.$0, p.$0), lookup(indexes1.$1, p.$0), lookup(indexes1.$2, p.$0)};
                    let val2 = {lookup(indexes2.$0, p.$1), lookup(indexes2.$1, p.$1), lookup(indexes2.$2, p.$1)};

                    let iter_output =
                        if(val1.$0 == val2.$0,
                            if(val1.$1 == val2.$1,
                                if(val1.$2 == val2.$2,
                                    {p.$0 + 1L, p.$1 + 1L, merge(p.$2, true), merge(p.$3, true)},
                                    if(val1.$2 < val2.$2,
                                        {p.$0 + 1L, p.$1, merge(p.$2, false), p.$3},
                                        {p.$0, p.$1 + 1L, p.$2, merge(p.$3, false)}
                                    )
                                ),
                                if(val1.$1 < val2.$1,
                                    {p.$0 + 1L, p.$1, merge(p.$2, false), p.$3},
                                    {p.$0, p.$1 + 1L, p.$2, merge(p.$3, false)}
                                )
                            ),
                            if(val1.$0 < val2.$0,
                                {p.$0 + 1L, p.$1, merge(p.$2, false), p.$3},
                                {p.$0, p.$1 + 1L, p.$2, merge(p.$3, false)}
                            )
                        );
                    {
                        iter_output,
                        iter_output.$0 < len1 &&
                        iter_output.$1 < len2
                    }
                ),
                {0L, 0L, appender[bool], appender[bool]}
    );
    # iterate over remaining un-checked elements in both arrays and append False until maxLen
    let res = if(res.$0 < maxlen, iterate(res,
            |p|
                {
                    {p.$0 + 1L, p.$1, merge(p.$2, false), p.$3},
                    p.$0 + 1L < maxlen
                }
    ), res);
    let res = if(res.$1 < maxlen, iterate(res,
            |p|
                {
                    {p.$0, p.$1 + 1L, p.$2, merge(p.$3, false)},
                    p.$1 + 1L < maxlen
                }
    ), res);
    let b = appender[vec[bool]];
    let c = merge(b, result(res.$2));
    result(merge(c, result(res.$3)))
```

```
|_inp0:vec[f32],_inp1:vec[f32],_inp10:vec[f32],_inp11:vec[f32],
_inp12:vec[vec[i8]],_inp17:vec[vec[i64]],_inp2:vec[vec[i8]],_inp9:vec[vec[i64]]|
  (let obj100:vec[f32]=(_inp0:vec[f32]);
  (let obj101:vec[f32]=(_inp1:vec[f32]);
  (let obj102:vec[vec[i8]]=(_inp2:vec[vec[i8]]);
  (let obj119:vec[f32]=(_inp10:vec[f32]);
  (let obj120:vec[f32]=(_inp11:vec[f32]);
  (let obj121:vec[vec[i8]]=(_inp12:vec[vec[i8]]);
  (let obj136:vec[f32]=(result(
    for(
      lookup(_inp9:vec[vec[i64]],0L),
      appender[f32],
      |b:appender[f32],i:i64,n:i64|
        merge(b:appender[f32],lookup(obj100:vec[f32],n:i64))
    )
  ));(let obj139:vec[f32]=(result(
    for(
      lookup(_inp17:vec[vec[i64]],0L),
      appender[f32],
```

```
      |b__3:appender[f32],i__3:i64,n__3:i64|
        merge(b__3:appender[f32],lookup(obj119:vec[f32],n__3:i64))
    )
  ));(let len1:i64=(len(obj136:vec[f32]));(let len2:i64=(len(obj139:vec[f32]));
  (let maxlen:i64=(if(
    (len1:i64>len2:i64),
    len1:i64,
    len2:i64
  ));(let indexes1:{vec[f32],vec[f32],vec[vec[i8]]}=({obj136:vec[f32],result(
    for(
      lookup(_inp9:vec[vec[i64]],1L),
      appender[f32],
      |b__4:appender[f32],i__4:i64,n__4:i64|
        merge(b__4:appender[f32],lookup(obj101:vec[f32],n__4:i64))
    )
  ),result(
    for(
      lookup(_inp9:vec[vec[i64]],2L),
      appender[vec[i8]],
      |b__5:appender[vec[i8]],i__5:i64,n__5:i64|
        merge(b__5:appender[vec[i8]],lookup(obj102:vec[vec[i8]],n__5:i64))
    )
  )});(let indexes2:{vec[f32],vec[f32],vec[vec[i8]]}=({obj139:vec[f32],result(
    for(
      lookup(_inp17:vec[vec[i64]],1L),
      appender[f32],
      |b__6:appender[f32],i__6:i64,n__6:i64|
        merge(b__6:appender[f32],lookup(obj120:vec[f32],n__6:i64))
    )
  ),result(
    for(
      lookup(_inp17:vec[vec[i64]],2L),
      appender[vec[i8]],
      |b__7:appender[vec[i8]],i__7:i64,n__7:i64|
        merge(b__7:appender[vec[i8]],lookup(obj121:vec[vec[i8]],n__7:i64))
    )
  )});(let res:{i64,i64,appender[bool],appender[bool]}=(if(
    if(
      (len1:i64>0L),
      (len2:i64>0L),
      false
    ),
    iterate(
      {0L,0L,appender[bool],appender[bool]},
      |p:{i64,i64,appender[bool],appender[bool]}|
        (let val1:{f32,f32,vec[i8]}=({lookup(indexes1.$0,p.$0),lookup(indexes1.$1,p.$0),lookup(indexes1.$2,p.$0)});
        (let val2:{f32,f32,vec[i8]}=({lookup(indexes2.$0,p.$1),lookup(indexes2.$1,p.$1),lookup(indexes2.$2,p.$1)});
        (let iter_output:{i64,i64,appender[bool],appender[bool]}=(if(
          (val1.$0==val2.$0),
          if(
            (val1.$1==val2.$1),
            if(
              (val1.$2==val2.$2),
              {(p.$0+1L),(p.$1+1L),merge(p.$2,true),merge(p.$3,true)},
              if(
                (val1.$2<val2.$2),
                {(p.$0+1L),p.$1,merge(p.$2,false),p.$3},
                {p.$0,(p.$1+1L),p.$2,merge(p.$3,false)}
              )
            ),
            if(
              (val1.$1<val2.$1),
              {(p.$0+1L),p.$1,merge(p.$2,false),p.$3},
              {p.$0,(p.$1+1L),p.$2,merge(p.$3,false)}
            )
          ),
          if(
            (val1.$0<val2.$0),
            {(p.$0+1L),p.$1,merge(p.$2,false),p.$3},
            {p.$0,(p.$1+1L),p.$2,merge(p.$3,false)}
          )
        ));{iter_output:{i64,i64,appender[bool],appender[bool]},if(
          (iter_output.$0<len1:i64),
          (iter_output.$1<len2:i64),
          false
        )}))),
    {0L,0L,appender[bool],appender[bool]}
  ));(let res__1:{i64,i64,appender[bool],appender[bool]}=(if(
    (res.$0<maxlen:i64),
    iterate(
      res:{i64,i64,appender[bool],appender[bool]},
      |p__1:{i64,i64,appender[bool],appender[bool]}|
        {{(p__1.$0+1L),p__1.$1,merge(p__1.$2,false),p__1.$3},((p__1.$0+1L)<maxlen:i64)}
    ),
    res:{i64,i64,appender[bool],appender[bool]}
  ));(let res__2:{i64,i64,appender[bool],appender[bool]}=(if(
    (res__1.$1<maxlen:i64),
    iterate(
      res__1:{i64,i64,appender[bool],appender[bool]},
      |p__2:{i64,i64,appender[bool],appender[bool]}|
        {{p__2.$0,(p__2.$1+1L),p__2.$2,merge(p__2.$3,false)},((p__2.$1+1L)<maxlen:i64)}
    ),
    res__1:{i64,i64,appender[bool],appender[bool]}
  ));result(
    merge(merge(appender[vec[bool]],result(
      res__2.$2
    )),result(
      res__2.$3
```

```
    ))
  ))))))))))))))))
```

Aggregation

```
# _inp18 is the join intermediate result
|_inp0: vec[f32], _inp18: vec[vec[bool]], _inp3: vec[f32],
_inp5: vec[f32], _inp7: vec[f32], _inp9: vec[vec[i64]]|
let obj100 = (_inp0);
let obj103 = (_inp3);
let obj105 = (_inp5);
let obj107 = (_inp7);
let obj116 = (lookup(_inp9, 0L));
let obj136 = (
    result(
        for(
            obj116,
            appender,
            |b, i, n|
                merge(b, lookup(obj100, n))
        )
    ));
let obj143 = (slice(lookup(_inp18, 0L), 0L, len(obj136)));
let obj155 = (
    result(
        for(
            zip(obj107, obj143),
            appender,
            |b, i, e|
                if (e.$1,
                    merge(b, e.$0),
                    b)
        )
    ));
let obj159 = (
    result(
        for(
            zip(obj105, obj143),
            appender,
            |b, i, e|
                if (e.$1,
                    merge(b, e.$0),
                    b)
        )
    ));
let obj171 = (
    result(
        for(
            zip(obj103, obj143),
            appender,
            |b, i, e|
                if (e.$1,
                    merge(b, e.$0),
                    b)
        )
    ));
let obj294 = (
    map(
        obj171,
        |a: f32|
            a != f32(-99.99)
    ));
let obj295 = (
    map(
        obj159,
        |a: f32|
            a != f32(-999.9)
    ));
let obj296 = (
    result(
        for(zip(obj294, obj295),
            appender,
            |b, i, n|
                merge(b, n.$0 && n.$1)
        )
    ));
let obj297 = (
    map(
        obj155,
        |a: f32|
            a != f32(-999.9)
    ));
let obj298 = (
    result(
        for(zip(obj296, obj297),
            appender,
            |b, i, n|
                merge(b, n.$0 && n.$1)
        )
    ));
let obj307 = (
    result(
```

```
        for(
            zip(obj171, obj298),
            appender,
            |b, i, e|
                if (e.$1,
                    merge(b, e.$0),
                    b)
        )
    ));


    let agg_min = f64(
        result(
            for(
                obj307,
                merger[f32, min],
                |b, i, e|
                    merge(b, e)
            )
        )
    );
    let agg_max = f64(
        result(
            for(
                obj307,
                merger[f32, max],
                |b, i, e|
                    merge(b, e)
            )
        )
    );
    let agg_mean = f64(
        result(
            for(
                obj307,
                merger[f32, +],
                |b, i, n|
                    merge(b, n)
            )
        )
    ) / f64(len(obj307));
    let agg_std = sqrt(
        result(
            for(
                obj307,
                merger[f64, +],
                |b, i, n|
                    merge(b, pow(f64(n) - agg_mean, 2.0))
            )
        ) / f64(len(obj307) - 1L)
    );
    let agg_result = appender[f64];
    let agg_result = merge(agg_result, agg_min);
        let agg_result = merge(agg_result, agg_max);
        let agg_result = merge(agg_result, agg_mean);
        let agg_result = merge(agg_result, agg_std);

    result(agg_result)
```

```
|_inp0:vec[f32],_inp18:vec[vec[bool]],_inp3:vec[f32],
_inp5:vec[f32],_inp7:vec[f32],_inp9:vec[vec[i64]]|
  (let obj100:vec[f32]=(_inp0:vec[f32]);
  (let obj143:vec[bool]=(slice(lookup(_inp18:vec[vec[bool]],0L),0L,len(result(
    for(
      lookup(_inp9:vec[vec[i64]],0L),
      appender[f32],
      |b:appender[f32],i:i64,n:i64|
        merge(b:appender[f32],lookup(obj100:vec[f32],n:i64))
    )
  ))));(let obj307:vec[f32]=(result(
    for(
      zip(
        result(
          for(
            zip(
              _inp3:vec[f32],
              obj143:vec[bool]
            ),
            appender[f32],
            |b__3:appender[f32],i__3:i64,e__2:{f32,bool}|
              if(
                e__2.$1,
                merge(b__3:appender[f32],e__2.$0),
                b__3:appender[f32]
              )
          )
        ),
        result(
          for(
            zip(
              _inp5:vec[f32],
              obj143:vec[bool]
            ),
            appender[bool],
            |b__7:appender[bool],i__7:i64,e__3:{f32,bool}|
              if(
                e__3.$1,
```

```
                merge(b__7:appender[bool],(e__3.$0!=(f32((-999.9)))),
                b__7:appender[bool]
              )
            )
          ),
          result(
            for(
              zip(
                _inp7:vec[f32],
                obj143:vec[bool]
              ),
              appender[bool],
              |b__10:appender[bool],i__10:i64,e__4:{f32,bool}|
                if(
                  e__4.$1,
                  merge(b__10:appender[bool],(e__4.$0!=(f32((-999.9)))),
                  b__10:appender[bool]
                )
            )
          )
        ),
        appender[f32],
        |b__12:appender[f32],i__12:i64,data__2:{f32,bool,bool}|
          (let tmp__9:f32=(data__2.$0);
          (let data__3:{f32,bool,bool,bool}=({tmp__9:f32,(tmp__9:f32!=(f32((-99.99)))),data__2.$1,data__2.$2});
          (let tmp__16:{bool,bool}=({data__3.$1,data__3.$2});
          (let data__4:{f32,bool,bool}=({data__3.$0,if(
            tmp__16.$0,
            tmp__16.$1,
            false
          ),data__3.$3});(let tmp__20:{bool,bool}=({data__4.$1,data__4.$2});
          (let e__5:{f32,bool}=({data__4.$0,if(
            tmp__20.$0,
            tmp__20.$1,
            false
          )});if(
            e__5.$1,
            merge(b__12:appender[f32],e__5.$0),
            b__12:appender[f32]
          )))))))
      )
    ));(let agg_mean:f64=(((f64(result(
      (let a:vec[f32]=(obj307:vec[f32]);for(
        fringeiter(a:vec[f32]),
        for(
          simditer(a:vec[f32]),
          merger[f32,+],
          |b__13:merger[f32,+],i__13:i64,n__3:simd[f32]|
            merge(b__13:merger[f32,+],n__3:simd[f32])
        ),
        |b__14:merger[f32,+],i__14:i64,n__4:f32|
          merge(b__14:merger[f32,+],n__4:f32)
      ))
    )))/(f64(len(obj307:vec[f32])))));result(
      merge(merge(merge(merge(appender[f64],(f64(result(
        (let a__1:vec[f32]=(obj307:vec[f32]);for(
          fringeiter(a__1:vec[f32]),
          for(
            simditer(a__1:vec[f32]),
            merger[f32,min],
            |b__15:merger[f32,min],i__15:i64,e__6:simd[f32]|
              merge(b__15:merger[f32,min],e__6:simd[f32])
          ),
          |b__16:merger[f32,min],i__16:i64,e__7:f32|
            merge(b__16:merger[f32,min],e__7:f32)
        ))
      )))),(f64(result(
        (let a__2:vec[f32]=(obj307:vec[f32]);for(
          fringeiter(a__2:vec[f32]),
          for(
            simditer(a__2:vec[f32]),
            merger[f32,max],
            |b__17:merger[f32,max],i__17:i64,e__8:simd[f32]|
              merge(b__17:merger[f32,max],e__8:simd[f32])
          ),
          |b__18:merger[f32,max],i__18:i64,e__9:f32|
            merge(b__18:merger[f32,max],e__9:f32)
        ))
      )))),agg_mean:f64),(sqrt((result(
        for(
          obj307:vec[f32],
          merger[f64,+],
          |b__19:merger[f64,+],i__19:i64,n__5:f32|
            merge(b__19:merger[f64,+],pow(((f64(n__5:f32))-agg_mean:f64),2.0))
        )
      )/(f64((len(obj307:vec[f32])-1L)))))))
    )))))
```

Full averaged `time` measurements

* voluntary context switch

real	user	sys	mem_max(K)	maj_faults	min_page_faults	vol_switch*	input	output	pipeline	data
40.882	27.394	8.356	2323648.8	316.6	3348482.6	2312.2	833739.2	32.0	python-lib	data_0
56.39	42.066	10.192	9116796.8	442.0	4028160.6	2399.4	466907.2	48.0	weld-single	data_0
54.136	70.03	273.35	9404417.6	442.2	3210153.0	2922065.4	466635.2	48.0	weld-par	data_0
1324.604	4249.342	124.674	23536354.4	208.0	42557664.6	617803.6	939180.8	4114451.2	spark-single	data_0
369.348	3647.296	136.192	35308324.8	208.2	41344222.8	429977.4	939062.4	5248096.0	spark-par	data_0
32.756	22.232	4.38	6206599.2	177.0	2339818.6	956.8	782878.4	35.2	R	data_0
10.182	19.584	2.728	3707797.6	13.0	996605.0	11526.6	730297.6	124.8	java	data_0
66.002	55.868	5.29	3484957.6	380.2	2134168.8	1055.6	876296.0	32.0	julia	data_0
81.142	60.708	13.804	4565404.0	316.6	6786974.8	2585.8	1541156.8	32.0	python-lib	data_1
74.736	52.942	16.805	18110324.8	442.4	8040864.4	2491.0	749006.4	48.0	weld-single	data_1
69.362	107.464	552.944	18636389.6	442.2	6803260.4	5970668.8	748315.2	48.0	weld-par	data_1
59.682	43.566	8.414	12340069.6	181.0	4611043.0	1264.0	1492648.0	40.0	R	data_1
19.188	78.22	7.066	6680946.4	13.8	2535449.6	16023.8	1437305.6	96.0	java	data_1
88.72	73.438	8.126	6736064.8	379.2	3952093.2	1304.0	1581798.4	32.0	julia	data_1
175.072	139.396	24.618	9035976.0	317.0	14663282.0	3130.2	2954179.2	32.0	python-lib	data_3
114.223	76.532	30.336	36092417.6	442.2	16352925.8	2757.4	1313014.4	48.0	weld-single	data_3
108.602	190.516	1276.434	36697211.2	442.8	18483611.8	13626705.4	1312366.4	48.0	weld-par	data_3
124.22	95.514	16.416	24607161.6	178.8	9071817.0	1811.2	2905651.2	40.0	R	data_3
35.252	150.654	14.224	13257040.8	13.6	5332736.4	23259.0	2850220.8	124.8	java	data_3
138.098	112.534	14.352	13238147.2	379.0	7720058.6	1828.6	2994465.6	32.0	julia	data_3
380.192	310.894	48.3	18009652.8	316.4	31803549.6	4261.4	5782099.2	32.0	python-lib	data_6
194.528	124.482	59.334	72204584.0	442.0	34698520.6	3194.0	2437700.8	48.0	weld-single	data_6
189.274	357.278	2685.828	72956360.0	441.2	44710359.0	27711147.8	2437558.4	48.0	weld-par	data_6
282.268	223.536	36.506	49140826.4	178.2	19011492.8	2877.2	5726195.2	40.0	R	data_6
72.492	315.54	34.982	20053210.4	13.4	11257939.0	39156.6	5677744.0	200.0	java	data_6
240.794	192.638	28.29	26242741.6	379.2	18585139.4	2975.2	5823576.0	32.0	julia	data_6
858.758	714.616	102.358	35956838.4	316.8	76181537.6	6463.2	11436292.8	32.0	python-lib	data_12
366.474	228.044	120.246	144293156.0	441.2	72305538.8	4048.8	4689144.0	48.0	weld-single	data_12
363.6	731.914	5930.686	145427569.6	441.4	96142212.0	62188483.0	4689812.8	48.0	weld-par	data_12
641.26	527.64	71.966	98208552.8	177.8	36618535.0	5118.6	11384472.0	40.0	R	data_12
148.420	724.854	42.57	44406498.4	13.2	22423074.0	68125.4	11331003.2	280.0	java	data_12
468.538	368.868	62.032	52251774.4	378.6	47135254.6	5198.0	11477622.4	33.6	julia	data_12
1631.362	1329.778	222.402	71863173.6	317.4	173704747.4	10901.2	22748574.4	32.0	python-lib	data_25
2133.428	1771.862	282.49	196374418.4	179.0	117636545.2	9556.2	22697177.6	48.0	R	data_25
284.322	1194.948	89.318	89090696.8	109.6	46201228.6	118495.8	22672232.0	452.8	java	data_25
976.18	746.928	155.88	104286566.4	378.8	135576575.2	9585.8	22788422.4	32.0	julia	data_25
4098.794	3391.166	545.09	143663156.8	331.6	440954859.8	19778.2	45372936.0	32.0	python-lib	data_50
660.544	3823.974	232.912	173073301.6	114.8	105139119.0	284944.4	45294361.6	851.2	java	data_50

Literature Study

Res. Question	Improving multi-library data analysis pipelines through abstract representation and co-compilation	
Key.	improve, multi-library, data analysis/processing, pipeline, abstract/intermediate representation, co-compilation	
	Criterion	Rationale
Inclusion Criteria		
1	A study that directly proposes software architectures, architectural styles or strategies that are generally applicable to multiple libraries*.	Since typical data analysis pipelines use multiple libraries (e.g. pipeline using pandas, scikit-learn, and tensorflow), we need articles proposing software solutions that are generally applicable to multiple libraries.
2	A study that addresses computation space and/or time as a quality attribute.	We measure improvements through space and/or time, therefore a study must show that there are improvements in any of these areas.
3	A study that is developed by either academics or practitioners.	Both academic and industrial solutions are relevant to this study.
4	A study that proposes a solution using abstract representations and/or co-compilation.	We are interested in papers proposing a solution involving abstract representations and/or co-compilation.
5	A study that is written in English.	For feasibility reasons, papers written in other languages than English are excluded.
Exclusion Criteria		
1	A study that does not propose software solutions applicable to multiple libraries*.	We want to investigate typical data analysis pipelines which involve multiple libraries, therefore papers only capable of improving a single library are excluded.
2	A study that does not show what improvements are possible.	A study must show realistic tests and improvements against other existing solutions.
3	A study which does not show improvements through abstract representation and/or co-compilation.	We are interested in studies involving abstract representations and/or co-compilation, therefore papers proposing only other types of improvements are rejected.
	*library = used in a broader sense, including e.g. python packages like pandas and tensorflow, but also e.g. multiple programming languages like Java and Python or through the usage of UDF's; in other words, if the relationship is many-to-one.	

Read		
	Title	not read
	Abstract	includes skimming through paper
	Full-text	some paragraphs may be skipped
Selected		
	Relevant	Used for which papers are relevant to the thesis though might not always fulfil all criteria; more like relevant work.
	Very relevant	Used for great papers as comparison to the thesis' approach.

#	Sel.		Title	Author	Year	Inclusion					Exclusion			Read		
	Rel.	VRel.				1	2	3	4	5	1	2	3	T.	A.	F.
1	yes	yes	Weld: Rethinking the Interface Between Data-Intensive Libraries	Matei Zaharia et al.	2017	x	x	x	x	x						x
2	no	no	Julia: A Fresh Approach to Numerical Computing	Jeff Bezanson et al	2017	x	x	x		x		x	x			x
3	yes	yes	Julia: A Fast Dynamic Language for Technical Computing	Jeff Bezanson et al	2012	x	x	x	x	x						x
4	no	no	NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison	A B M Moniruzzaman and S A Hossain	2013			x		x	x	x	x			x
5	yes	no	Spark SQL: Relational Data Processing in Spark	Matei Zaharia et al.	2015		x	x	x	x	x					x
6	no	no	Spark: Cluster Computing with Working Sets	Matei Zaharia et al.	2010	x	x	x		x			x			x
7	no	no	Query Optimization in Database Systems	M Jarke and J Koch	1984			x		x	x	x	x		x	
8	no	no	Exploring the Performance of Spark for a Scientific Use Case	S Sehrish et al.	2016						x	x	x		x	
9	no	no	Dremel: Interactive Analysis of Web-Scale Datasets	S Melnik et al.	2010			x	x	x	x				x	
10	no	no	An Overview of the HDF5 Technology Suite and its applications	M Folk et al.	2011		x	x		x	x	x	x			x
11	yes	no	High-level GPU programming in Julia	Tim Besard et al	2016	x	x	x	x	x					x	
12	yes	no	Parallelizing Julia with a Non-invasive DSL	Todd A Anderson et al.	2017	x	x	x	x	x					x	
13	no	no	Engineering a Customizable Intermediate Representation	K Palacz et al.	2003	x		x	x	x					x	
14	no	no	A Common Compiler Framework for Big Data Languages: Motivation, Opportunities, and Benefits	Vinayak R. Borkar and Michael J. Carey	2013	x		x	x	x		x				x
15	yes	no	Algebricks: a data model-agnostic compiler backend for Big Data languages	V Borkar et al.	2015	x	x	x	x	x						x
16	yes	no	LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation	Chris Lattner and Vikram Adve	2004	x	x	x	x	x						x
17	yes	no	INSPIRE: The Insieme Parallel Intermediate Representation	H Jordan et al.	2013	x	x	x	x	x						x

#	Sel.		Title	Author	Year	Inclusion					Exclusion			Read		
	Rel.	VRel.				1	2	3	4	5	1	2	3	T.	A.	F.
18	yes	no	Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines	James Hegarty et al.	2014	x	x	x	x	x						x
19	no	no	Database Query Optimization Using Compilation Techniques	M D R Abadi	2013		x	x	x	x	x					x
20	no	no	DLVM: A modern compiler infrastructure for deep learning systems with adjoint code generation in a domain-specific IR	R Wei et al.	2017			x	x	x	x	x			x	
21	no	no	Data Structures for Statistical Computing in Python	Wes McKinney	2010					x	x	x	x			x
22	no	no	TensorFlow: A System for Large-Scale Machine Learning	Google Brain	2016					x	x	x	x		x	
23	no	no	Basic Linear Algebra Subprograms for Fortran Usage	Chris L Lawson et al.	1979					x	x	x	x		x	
24	no	no	Efficiently Compiling Efficient Query Plans for Modern Hardware	T Neumann	2011		x	x	x	x	x					x
25	no	no	Monad Comprehensions: A Versatile Representation for Queries	Torsten Grust	2004					x	x	x	x	x		
26	yes	yes	An Architecture for Compiling UDF-centric Workflows	A Crotty et al.	2015	x	x	x	x	x						x
27	no	no	LINQ: Reconciling Object, Relations and XML in the .NET Framework	E Meijer et al.	2006									x		
28	no	no	Dandelion: A Compiler and Runtime for Heterogeneous Systems	CJ Rossbach et al.	2013			x	x	x	x	x		x		
29	yes	no	Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud	SC Muller et al.	2014	x	x	x	x	x					x	
30	no	no	OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems	John E Stone et al	2010	x	x	x	x	x		x				x
31	no	no	An introduction to spir-v.	John Kessenick	2015	x			x	x		x		x		
32	no	no	FlumeJava: easy, efficient data-parallel pipelines	C Chambers et al.	2010		x	x	x	x	x					x
33	yes	no	Musketeer: All for One, One for All in Data Processing Systems	I Gog et al.	2015	x	x	x	x	x						x
34	yes	yes	Implicit Parallelism Through Deep Language Embedding	A Alexandrov et al.	2015	x	x	x	x	x					x	
35	no	no	HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots	A Kemper and T Neumann	2012		x	x		x	x		x		x	
36	yes	no	Flare: Native Compilation for Heterogeneous Workloads in Apache Spark	M Essertel et al.	2017	x	x	x	x	x						x
37	yes	no	Database-Inspired Optimizations for Statistical Analysis	H Mühleisen	2018	x	x	x	x	x						x
38	no	no	A Compiler Intermediate Representation for Reconfigurable Fabrics	Zhi Guo and Walid Najjar	2006		x	x	x	x	x				x	
39	no	no	META-pipe–Pipeline Annotation, Analysis and Visualization of MarineMetagenomic Sequence Data	EM Robertson	2016		x	x		x	x		x	x		
40	yes	no	Domain-Specific Language for Data Analytics Pipelines	C Misale	2017	x	x	x	x	x					x	
41	no	no	Tapir: Embedding Fork-Join Parallelism Into LLVM Intermediate Representation	TB Schardl	2017		x	x	x	x	x				x	
42	no	no	A Language for the Compact Representation of Multiple Program Versions	S Donadio	2005		x	x		x	x				x	
43	yes	no	Radish: Compiling Efficient Query Plans for Distributed Shared Memory	B Myers	2014		x	x	x	x					x	
44	yes	yes	HPAT: High Performance Analytics with Scripting Ease-of-Use	E Totoni	2017	x	x	x	x	x						x
45	yes	no	Erbium: A Deterministic, Concurrent IntermediateRepresentation to Map Data-Flow Tasks to Scalable,Persistent Streaming Processes	C Miranda	2010		x	x	x	x					x	
46	yes	no	Pegasus: An Efficient Intermediate Representation	M Budiu	2002			x	x	x		x			x	
47	yes	no	Bridging the Gap: Towards Optimization Across Linear and Relational Algebra	A Kunft	2016			x	x	x		x				x
48	no	no	Naiad: A Timely Dataflow System	G Murray et al.	2013		x	x		x	x		x	x		
49	no	no	AJIRA: a Lightweight Distributed Middleware for MapReduce and Stream Processing	J Urbani et al.	2014		x	x		x	x		x		x	
50	no	no	The Dataflow Model: A Practical Approach to BalancingCorrectness, Latency, and Cost in Massive-Scale,Unbounded, Out-of-Order Data Processing	T Akidau et al.	2015	x		x		x		x	x		x	
51	no	no	NoDB: Efficient Query Execution on Raw Data Files	I Alagiannis et al.	2008		x	x	x	x	x					
52	no	no	An architecture for recycling intermediates in a column-store	M Ivanova et al.	2010		x	x	x	x	x					
53	no	no	Scalability! But at what COST?	F McSherry et al.	2015		x	x		x	x	x	x			x

Bibliography

[1] Dragon1, "Data mining definition," 2018. [Online; accessed July 1, 2018].

[2] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[3] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono, "Research directions in data wrangling: Visualizations and transformations for usable and credible data," *Information Visualization*, vol. 10, no. 4, pp. 271–288, 2011.

[4] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)*, vol. 16, no. 2, pp. 111–152, 1984.

[5] G. Coder, "Ast," 2012. [Online; accessed July 1, 2018].

[6] R. Timmermans, "Creating immutable tree data structures in ruby," 2013. [Online; accessed July 1, 2018].

[7] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[8] S. Palkar, J. J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, S. Madden, and M. Zaharia, "Weld: Rethinking the interface between data-intensive applications," *CoRR*, vol. abs/1709.06416, 2017.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, 2015.

[11] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[12] H. Zhou, "Introduction to julia," 2017. [Online; accessed July 1, 2018].

[13] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level

image processing code into hardware pipelines.," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[14] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton, "Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 11–20, ACM, 2010.

[15] E. Totoni, T. A. Anderson, and T. Shpeisman, "Hpat: high performance analytics with scripting ease-of-use," *arXiv preprint arXiv:1611.04934*, 2016.

[16] T. A. Anderson, H. Liu, L. Kuper, E. Totoni, J. Vitek, and T. Shpeisman, "Parallelizing julia with a non-invasive dsl," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 74, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[17] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy, "Pydron: Semi-automatic parallelization for multi-core and the cloud.," in *OSDI*, pp. 645–659, 2014.

[18] R. Rew, G. Davis, S. Emmerson, H. Davies, and E. Hartnett, "Netcdf user's guide," 1993.

[19] J. Kestelyn, "Introducing parquet: Efficient columnar storage for apache hadoop," *Cloudera Blog*, vol. 3, 2013.

[20] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 967–980, ACM, 2008.

[21] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pp. 36–47, ACM, 2011.

[22] esri, "Fundamentals of netcdf data storage," 2016. [Online; accessed July 7, 2018].

[23] A. C. Lorenc, "Analysis methods for numerical weather prediction," *Quarterly Journal of the Royal Meteorological Society*, vol. 112, no. 474, pp. 1177–1194, 1986.

[24] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database," 2012.

[25] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek, "Engineering a customizable intermediate representation," in *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pp. 67–76, ACM, 2003.

[26] H. Mühleisen, A. Bertram, and M.-J. Kallen, "Database-inspired optimizations for statistical analysis," *Journal of Statistical Software*, 2017.

[27] V. Borkar, Y. Bu, E. P. Carman Jr, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras, "Algebricks: a data model-agnostic compiler backend

for big data languages," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 422–433, ACM, 2015.

[28] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1466–1477, 2015.

[29] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 2, ACM, 2015.

[30] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl, "Bridging the gap: towards optimization across linear and relational algebra," in *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, p. 1, ACM, 2016.

[31] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into llvm's intermediate representation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 249–265, ACM, 2017.

[32] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," 2002.

[33] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "Inspire: The insieme parallel intermediate representation," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 7–18, IEEE Press, 2013.

[34] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?," in *HotOS*, vol. 15, pp. 14–14, Citeseer, 2015.

[35] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Native compilation for heterogeneous workloads in apache spark," *arXiv preprint arXiv:1703.08219*, 2017.

[36] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 195–206, IEEE, 2011.

[37] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, pp. 51–56, Austin, TX, 2010.

[38] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[39] S. Andalam, R. Sinha, P. S. Roop, A. Girault, and J. Reineke, "Precise modelling of instruction cache behaviour," 2013.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[41] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An architecture for recycling intermediates in a column-store," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 4, p. 24, 2010.

[42] M. L. Noga, S. Schott, and W. Löwe, "Lazy xml processing," in *Proceedings of the 2002 ACM symposium on Document engineering*, pp. 88–94, ACM, 2002.

[43] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 241–252, ACM, 2012.

[44] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[45] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[46] R. C. Team, "R language definition," *Vienna, Austria: R foundation for statistical computing*, 2000.