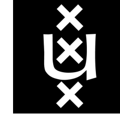
Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master Thesis

Loop-Adaptive Execution in Weld

Author: Richard Gankema (2594274)

1st supervisor: Peter Boncz
daily supervisor: Peter Boncz (CWI)
2nd reader: Hannes Mühleisen

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

August 26, 2018

Acknowledgements

I would like to thank my supervisors Peter Boncz and Hannes Mühleisen, as well as Stefan Manegold, for giving me the opportunity to take almost a full year for this thesis (albeit part-time at first), allowing me to investigate and understand virtually every little detail of Weld. I further thank Peter for his guidance, as well as his unrelenting belief that there should be some opportunity for loop-adaptivity in Weld while the results seemed more and more depressing in the early stage of this research. Most importantly, because of his seemingly limitless knowledge compared to that of a humble MSc student, I feel that I have learned more during this process than I have in any other academic year. I would also like to thank Mihai, whose work and help enabled me to perform TPC-H benchmarks in Weld, and Shoumik Palkar, who clarified the underlying reasons of many of the problems I faced while working with Weld.

On a more personal level I wish to thank my friends and flat mates (which fortunately are not mutually exclusive concepts), and Selas, who have given me both focus and distraction when I needed it, as well as their full understanding for my absence while I was finishing this thesis.

Abstract

We investigate the implementation and benefits of *loop-adaptivity* in Weld, a data-centric JIT-compiling data processing framework. Loop-adaptivity is inspired by *micro-adaptivity*, where a DBMS is able to pick the best operator (in terms of performance) for a given combination of hardware, runtime environment and data, even when this changes over time. Micro-adaptivity was proposed in Vectorwise, a vectorized-interpreted DBMS, but does not fit the data-centric approach where the granularity of execution is not an operator, but an entire `for`-loop that implements a full pipeline of operators. Loop-adaptivity therefore adapts entire loops, rather than operators. Integrating this kind of adaptivity in a JIT-compiling system is challenging because the amount of code that needs to be generated increases exponentially in the number of adaptive decisions. This is (i) at odds with the already high pressure to keep compilation times down, and (ii) can make it challenging to find the best implementation in time. Moreover, to keep overheads down, getting runtime statistics cannot be done on a tuple-at-a-time basis, which is again at odds with the data-centric approach. In this thesis we show how to tackle most of these problems, but leave the challenge of finding the best variation as future work. We investigate the opportunities for loop-adaptivity in Weld, which appears to be more limited than micro-adaptivity in Vectorwise. We evaluate how loop-adaptivity performs using both micro-benchmarks and the TPC-H benchmark suite, and find that we can gain modest improvements, limited to higher scale factors.

Contents

1 Introduction **5**
1.1 Micro-adaptive execution 5
1.2 Loop-adaptive execution 7
1.3 Research questions 7

2 Related Work **9**
2.1 Adaptive query processing 9
2.1.1 Static query optimization 10
2.1.2 Adaptive selection 11
2.1.3 Adaptive joins 11
2.1.4 Adaptive aggregation 13
2.2 Micro-adaptive execution 13
2.2.1 Multi-armed bandit algorithms 13
2.2.2 Large search spaces 14
2.3 Query compilation 14
2.3.1 Reducing interpretation overhead 14
2.3.2 Recent query compilers 15
2.3.3 Abstraction without regret 16
2.3.4 High-level query compilers 16
2.3.5 Weld 17
2.4 Contributions 18

3 Weld **19**
3.1 Overview 19
3.2 Internal representation 19
3.2.1 Data types 20
3.2.2 Basic expressions 20
3.2.3 Builder expressions 21
3.2.4 Examples 21
3.2.5 Annotations 23
3.3 Optimizer 23
3.3.1 Optimization passes 23
3.3.2 Code generation 24
3.3.3 LLVM 25

3.4 Execution engine 26
3.4.1 Parallelism 26

4 Opportunities for Loop-Adaptivity 28
4.1 Experimental setup 29
4.2 Selective versus full computation 29
4.2.1 Flavors 29
4.2.2 Experiment 31
4.2.3 Discussion 31
4.3 Branching versus predication 33
4.3.1 Branch prediction and selectivity 34
4.3.2 Flavors 34
4.3.3 Experiment 35
4.4 Hash joins with Bloom filters 36
4.4.1 Hash tables and Bloom filters in Weld 37
4.4.2 Flavors in Weld 37
4.4.3 Experiment 39
4.5 Conclusion 39

5 Loop-Adaptivity Implementation 41
5.1 Overview 41
5.2 Internal representation 42
5.2.1 Instrumentation and deferred let-expressions 44
5.3 Flavor generation 46
5.3.1 Selection-projection loop reordering 46
5.3.2 Branching versus predication 47
5.3.3 Adaptive Bloom filters 48
5.4 Code generation 49
5.4.1 Switchfor 49
5.4.2 Deferred let 50
5.4.3 Instrumentation and global variables 50
5.4.4 Lazy function compilation 50
5.5 Runtime 51
5.5.1 Switchfor 51
5.5.2 Deferred let-expressions 52
5.6 VW-greedy 52

6 Results and Discussion 55
6.1 Experimental setup 55
6.2 Adaptive predication 55
6.3 Adaptive selection projection order 57
6.4 Adaptive Bloom filters 57
6.4.1 Simple join 57
6.4.2 Three-way dictmerger join 59
6.4.3 Three-way groupmerger join 60

6.5 Bloom filter creation 61
6.6 Instrumentation overhead 62
6.7 TPC-H benchmarks 63
6.7.1 Varying scale factors 63
6.7.2 Varying thread counts 64
6.7.3 Overall speedups 65

7 Conclusion **66**
7.1 Future Work 68

A Literature Study **74**

Chapter 1

Introduction

Databases typically store data in a Relational Database Management System (RDBMS), which allows users to efficiently structure and query large sets of data. More recently, data processing frameworks such as Spark [51] are also being used for data science tasks. In essence, the goal of these systems are the same: abstracting away the low-level operations required to efficiently process data, so that users can efficiently express and execute their queries without being systems experts. For example, programs written in Spark are automatically parallelized and distributed over clusters through a relatively simple API. Abstractions like this become only more relevant as hardware becomes more complex.

Data processing frameworks such as Spark are increasingly turning to Just-In-Time (JIT) compilation in order to improve performance by eliminating interpretation overhead. However, translating a high level abstraction such as SQL into efficient machine code is far from trivial, potentially resulting in less than optimal performance, or hard-to-find bugs. Koch argues in [23] that, to ease the development of these systems, multiple levels of abstraction are needed, so that the high level user facing interface can be lowered to machine code one step at a time.

A recent interesting example of such an abstraction is Weld [32]. Weld is a low level JIT-compiled domain specific language (DSL) that libraries can use to efficiently implement their computationally expensive operators. Code written in Weld can target both CPUs and GPUs, is automatically parallelized and SIMDized, and can even optimize across functions and libraries.

1.1 Micro-adaptive execution

Not only did hardware become more complex over the years, it also became more diverse. Modern hardware platforms typically offer a wide range of features that make them more efficient, such as branch prediction, out-of-order-execution and SIMD instructions [10]. The design process of a hardware architecture is one of making compromises between conflicting optimizations, meaning that two different platforms can behave very differently, even when running the same instructions. This makes generating efficient code for an algorithm a challenging task, because some algorithmic variant that was optimized for one machine may very well be suboptimal for another.

To make matters more complicated, there are more factors that influence the performance of an algorithmic variation. These include:

- Compiler (settings): each compiler makes its own optimization decisions that can lead to radically different machine code. These are also influenced by the actual compiler settings. In JIT-compiling systems there is the additional trade-off between faster code through optimizations at the cost of longer compilation times, meaning that long running programs will be better off with more optimizations than short running programs.
- Runtime constraints: the runtime environment in which a machine is operating can influence the performance of the machine. As an example, modern CPUs will throttle their clock speeds as a response to heat.
- Concurrent load: running an algorithm single-threaded will give different results from running it on all cores in terms of per work-unit performance. Even if the computation requires no synchronization that would slow it down. Both the computation infrastructure (throttling down as more cores are busy) and memory infrastructure (cache footprint, memory bus usage) will degrade due to concurrent usage.
- Data: the way data is distributed can have a significant impact on the performance of a given algorithm. For instance, a filter can be implemented using either branching instructions or predicated instructions. When the filter predicate is mostly true or mostly false for the input data, the CPU will be able to effectively perform branch predictions, so an algorithm using if-then-else (branching) instructions will perform well. In other cases, an algorithmic variant that transforms the if-then-else into a computation will typically be faster, as it doesn't suffer from branch mispredictions.

All these factors mean that there is typically no one-size-fits-all implementation for an algorithm when it comes to optimal performance. This means that to gain maximum performance, the right algorithmic variation should be chosen for the given job.

The classical way of dealing with this problem is to build logical cost models that estimate the number of tuples, their data distributions, as well as physical cost models that try to predict the behavior of the underlying hardware. Although logical cost models have their use in the optimization of query execution plans, they are error-prone and rarely accurate [12]. With hardware becoming increasingly diverse, making an accurate physical model is extremely challenging. Moreover, such a physical cost model cannot account for future architectures and advances in technology. Finally, physical cost models simply cannot account for any unexpected change in the system that might happen during runtime. All in all, a static analysis will generally fail to predict query behavior for a specific combination of hardware and data, and so picking the right algorithmic variation is unlikely.

A way out of this "performance jungle" is to introduce *micro-adaptive* query execution [41], where a task is decomposed of many mini-tasks. If the query engine has different ways of executing the same task, it can measure their effectiveness during runtime, and choose the implementation that proves to be the most effective. This approach means

that the system can be agnostic about all the factors that may influence performance, as it can simply base its decisions on the raw performance it observes.

1.2 Loop-adaptive execution

Micro-adaptivity was proposed in Vectorwise which uses a query processor, but has so far not been implemented in a JIT-compiling infrastructure such as Weld. Integrating any kind of adaptivity in a JIT-compiling system such as Weld (or Spark, or Hyper [21]) is challenging because:

- Adaptive decisions cannot be made for each tuple, but must be done for chunks of tuples at a time to amortize overhead. This is at odds with the data-centric JIT approach [29] that fuses all operators in a pipeline into a single tuple-at-a-time-loop. We need to break these loops into chunks in order to make adaptive decisions.
- Every adaptive factor between two alternatives increases the number of code paths in the enclosed loop by two, so the amount of generated code increases exponentially with the number of decisions. This is at odds with the need to keep compilation times down. Also, with an exponential number of options, it becomes increasingly harder to home in on the optimal choices. Ideally, independent choices should not increase the number of code paths, nor the amount of options to test, but this requires breaking up generated code in separate parts. However, introducing function calls for every tuple introduces significant overhead.

Micro-adaptive execution differs from regular adaptive query processing in that it does not adapt entire query plans, but single operators (hence the term *micro*). One should observe that micro-adaptivity in Hyper style (data-centric) JIT systems is impossible, as the granularity of execution is a `for`-loop that corresponds to one query pipeline, which may correspond to multiple operators and many primitives. Given that the unit that can be varied is larger (than micro), we propose *loop-adaptive execution*, where we implement adaptivity in the same spirit as micro-adaptive execution, but at the larger scale of a data-centric `for`-loop. We demonstrate the concept in this thesis, by implementing it in Weld.

1.3 Research questions

This research topic raises many questions, of which the most important ones are listed below.

1. Where are the opportunities for loop-adaptive execution in Weld, if any?
2. Should Weld be adapted, and how do we design loop-adaptive execution in Weld?
    - 2.1. How do we deal with code explosion?
        - 2.1.1. How does lazy compilation (during query execution) fit into Weld?

    2.1.2. How can we quickly home in on the best variations?

  2.2. How do we implement instrumentation in Weld?

    2.2.1. How do (global) variables fit into Weld?

    2.2.2. What is the overhead of the instrumentation?

3. What are the potential speedups gained by loop-adaptivity in Weld?

Micro-adaptivity has shown to have positive effects on performance in a interpreted database system, but has not been tried in a JIT-compiled framework yet. Therefore, the first question is about identifying those opportunities that are specific to Weld. We are interested in loops that can be implemented in multiple variants, each outperforming the others in some contexts, but no variant dominating all others. An example of this may be a loop that implements the probe phase of a hash join. The hash table lookup could be preceded with a Bloom filter lookup, or not. Neither version should be faster in all cases, but instead the preferable choice depends on the hit ratio of the hash table.

If we find those opportunities, there is a reason to implement it. This raises the second part of question two, the challenging part of our research. The main problem is about code explosion (question 2.1), the other problem is about *instrumentation* (question 2.2). Code explosion is a problem for two reasons: first of all, because Weld is JIT-compiled, code explosion will result in longer compilation times, which may nullify any performance gains. A potential solution would be to compile only those variants when we need them. Right now the entire Weld IR is compiled before it is executed, so how to implement compilation-on-demand would be an interesting question (2.1.1). Question 2.2.2 is about the other problem of code explosion. The adaptive policy from Vectorwise has shown to quickly choose the fastest variation when there are a limited number of variations, but this may not be the case if the number of variations grows exponentially.

For some variations we will likely need instrumentation to decide whether we want to test it. An example of this is a hash join that can be implemented with or without a Bloom filter. A Bloom filter can improve the performance of a hash join, but only if the hit rate of the hash table is low enough. Therefore, to decide whether it is worth it to build and test a Bloom filter, we will want to keep track of the hit rate of a hash table. A potential lightweight approach of doing this would require variables, but Weld is referentially transparent and so only offers constants. Moreover, the runtime needs to be made aware of these variables so it can make adaptive decisions. This is what question 2.2.1 is about. Question 2.2.2 is about the additional overhead that instrumentation will introduce. Keeping this low is imperative for loop-adaptivity to be of any use.

Finally, we want to evaluate the system by looking at potential speedups gained by loop-adaptivity in Weld (question 3).

Chapter 2

Related Work

This research focuses on the intersection of the field of (micro-)adaptive execution and JIT-compiling DBMSs. As such the related work can be found in these two topics. In section 2.1, we will take a look at adaptive query execution, and micro-adaptive execution in section 2.2. Section 2.3 explores related work in JIT-compiling queries, while we explore the gap between the two fields and our contributions in section 2.4.

2.1 Adaptive query processing

Because of the declarative nature of query languages for relational databases (most commonly SQL), a DBMS is not tied to a single strategy (or query plan) to execute it. Instead, it is free to pick one of many equivalent query plans, as long as they all return the correct answer to the query. Which one is faster depends on a number of factors, such as the distribution of the data that is processed by the query. This was already recognized by the designers of System R [1], one of the first relational database management systems (RDBMS). Before executing a query, System R would analyze a given query and pick the best query plan among a set of plans, using a cost model and basic statistics about each table in the database. This manner of statically optimizing a query plan before execution has been improved over the years, but in its essence it is still used by most query processors [11].

Static optimization of query plans can work quite well, as long as there are sufficient statistics available, there are few correlations in the data, and the runtime execution is predictable. However, as queries, datasets and hardware becomes more complex, these assumptions hold less and less often. Even when statistics on tables as a whole are perfect, correlations between predicates can cause cardinality estimates for a query to be off by several orders of magnitude [18]. To account for this, several extensions to static query optimizations [3, 8, 15] have been proposed. For example, robust query optimization [3] tries to find the plan that has the least expected cost over a range of values that the parameters are expected to take, rather than merely using a point estimation of these parameters. Although clearly an improvement over classical query optimization, it still requires accurate statistics, and (like any kind of static planning) cannot account for changes in data or execution environment during runtime.

Adaptive query processing (AQP) is a technique that tries to counter this problem by monitoring the query during runtime to gather more accurate statistics, and adapting the query plan if it is found that the current one is performing suboptimal. In this section, we will give an overview of different operators and techniques that are used to this end. We will roughly follow the structure of [12], a comprehensive 2007 survey of AQP techniques by Deshpande et al., but also review some more modern adaptive execution techniques that were proposed since then. First we will take a brief look at static optimizations, specifically for selection ordering and joins, as they form the basis for many adaptive algorithms. We will then continue with several techniques for adaptive selection ordering, join ordering, aggregation, as well as various other adaptive execution schemes. Many adaptive query execution algorithms are developed specifically for streaming algorithms, but they are mostly left out of this discussion, as we focus on more mainstream data processing.

2.1.1 Static query optimization

According to [12], most static query optimization algorithms used can be classified as either *selection ordering* or *join enumeration* strategies. We will briefly discuss these here as they form the basis of many adaptive algorithms.

Selection ordering

Selection ordering refers to ordering the predicates of a selection query as efficiently as possible. To minimize overall work, the predicate that filters out most of the tuples should be processed first, while the least selective predicate should be last, also taking into account the cost of evaluating the predicate itself. Let a query have n predicates, $S_1, \ldots, S_n$, each to be applied on tuples from some table. Furthermore, let c_i be the cost of evaluating predicate S_i, $p(S_i)$ the probability that a tuple satisfies predicate S_i, and $p(S_i|S_{j1}, \ldots, S_{jk})$ the conditional probability that a tuple satisfies predicate S_i given that it already satisfied the previous set of predicates $\{S_{j1}, \ldots, S_{jk}\}$. If we assume the predicates are independent, then $p(S_i|S_{j1}, \ldots, S_{jk}) = p(S_i)$, and selection ordering boils down to simply sorting the predicates in increasing order of $c_i/(1 - p_i)$ [17].

Unfortunately, predicates of real life queries are often correlated, in which case the above strategy will not yield an optimal selection order. If the predicates are correlated, finding the optimal order is NP-Hard [4], so an approximation will have to do. One method is the *Greedy* algorithm [4], which works by first selecting the predicate with the lowest $c_i/(1 - p(S_i))$, and then repeatedly selecting the predicate with the lowest $c_j/(1 - p(S_j|S_i, \ldots))$ until no predicates remain. Although this algorithm can find reasonably performing orderings, it does require us to know the conditional probabilities of all the predicates, which may be unrealistic.

Join enumeration

Compared to the matter of picking a selection order, more considerations have to be made when optimizing a query that joins two or more tables. Similar to selections, the order in which tables are joined can heavily impact the performance of a join. The optimal

order of a join can be determined in a fashion similar to that of selection ordering. Other choices include the used access method for each table as well as the join algorithms used. The access method refers to the way tuples are accessed for a table in a query, such as a direct table scan or a using an index. The choice for an access method can influence the choice of a join algorithm. The join algorithm refers to the implementation of the join itself, such as (indexed) nested loops join, merge join or hash join. They each perform differently depending on factors such as the size of the table.

2.1.2 Adaptive selection

A relatively simple algorithm for adaptively ordering selection predicates is *A-Greedy* [4], which is based on the Greedy algorithm that we discussed in section 2.1.1. A-Greedy was developed with stream processing in mind, but in principle should also work for traditional data processing. Rather than estimating the selectivities of the predicates up-front, it monitors them continiously during query execution. A-Greedy consists of roughly three components: the query executor, profiler and reoptimizer. The query executor simply executes the query according the execution plan. The profiler monitors the selectivities of the predicates using a sliding window over the data, thereby only taking recent data into account. Using a sample of the tuples in this window, the profiler estimates the expected evaluation costs of each predicate. Finally, the reoptimizer is responsible for ensuring that the plan is optimal, using the profile generated by the profiler and the Greedy algorithm.

Other algorithms for adaptive selection ordering include the usage of the *Eddy* operator [2], which can adapt the order in which operators are applied on a tuple-at-a-time basis. Because of the high associated overhead, their use is mostly limited to streaming environments.

2.1.3 Adaptive joins

An optimizer has different adaptive opportunities and decisions to make depending on whether the (sub-)query is fully pipelined or contains materialization points. In distributed settings we have yet another set of challenges. We will look at methods for adaptive joins for each of them.

Pipelined execution

When the *driver* table is fixed, the remaining tables in a join can be ordered similarly to selection ordering. A-Greedy can be extended to this end, as discussed in [4]. This assumes a left-deep join tree and so does not consider bushy plans. Moreover, the choice for a driver table is one of the most important decisions an optimizer can take with regards to join-order [12], and so fixing the driver table is rather limiting. Symmetric hash joins [50] or MJoins [35] can be used to change the order of a join on-the-fly, but these require significantly more memory and are typically only used in streaming solutions.

A more flexible approach is *corrective query processing* [19]. In this scheme, execution cost and tuple cardinalities are continuously monitored during query execution, to determine if the current execution plan is performing as expected. If not, the engine will suspend execution and reoptimize the plan based on the previous statistics. It will then

proceed to execute the new plan on the remaining data. This reoptimization may happen an arbitrary number of times until all the data has been processed. The intermediate results of each phase are combined in a final *stitch-up* phase.

In contrast, the approach in [34] only generates a single execution plan that embeds a small number of switchable access methods in it, and can adapt the join order in mid-execution. This avoids the overhead of generating and deciding among a large number of alternative execution plans. It only works for indexed nested-loop joins, but in principle can be extended to pipelined hash-joins as well.

Non-pipelined execution

Materialization points in a query plan offer natural opportunities for adaptivity. For instance, if the result of a join is to be sorted, and then used as input for the remaining plan, we could measure the cardinality of the result, and (re-)optimize the remainder of the plan accordingly. Many application programs exploit this by explicitly splitting up a query into sub-queries, writing intermediate results to temporary tables. This allows the optimizer to gather accurate statistics on the intermediate results and therefore create a more accurate plan for the remainder of the operation. This is referred to as *plan staging* in [12].

A more general approach, which puts the execution engine in charge, is mid-query reoptimization [20]. The main idea is that a query plan has additional *checkpoint* operators, that measure the cardinality of tuples flowing through it. If the cardinality is very different from what was estimated, the remaining execution plan may be reoptimized. This is trivial when checkpoints are placed after a materialization point, which more or less boils down to plan staging. However, when placed in the middle of a pipeline, query reoptimization means that intermediate results may have to be discarded, or duplicates have to be removed. One way of dealing with this is by forcing materialization at specific points in the execution plan [27]. Although materialization can be expensive, in some cases it may be worth it if it allows the optimizer to avoid a bad plan that might be even more expensive.

A related, more recent approach, is *incremental execution* [30]. Using this approach, the optimizer optimizes the entire query plan as usual, using classical cardinality estimations. However, if it finds that certain cardinality estimations are uncertain, it will first execute small parts of the plan and materialize intermediate results to find the actual cardinalities, and then optimize again. If required, a more promising query plan is picked for the remainder of the execution.

Distributed execution

In a distributed setting an optimizer should also consider how to distribute the rows of two tables. If the outer relation is small, it is usually best to broadcast it to all nodes, and distribute the other rows randomly (*broadcast-random*). If the relation is too large, it is best to distribute both tables by hash (*hash-hash*), to avoid huge transmission overheads. [5] solves this issue with an adaptive method called *hybrid-hash*. In this method, a join starts using the hash-hash method, but statistics are collected on a small number of rows

at the beginning of execution. If the cardinality of a table appears to be sufficiently low, it is broadcast to the remaining nodes. Otherwise it continues using hash-hash distribution.

2.1.4 Adaptive aggregation

There are multiple ways of aggregating data in a multi-threaded environment, and which one performs depends on the data. [9] proposes a method that combines a sampling phase and a cost model to pick the best strategy, which is a combination of decisions such as whether to use local or global hash tables, using locking or atomic operators, and whether to aggregate runs. Although results show that the method nearly always achieves optimal performance, it was tuned specifically for the Sun UltraSPARC T1, and so whether the algorithm is as effective on other machines is questionable.

For distributed settings, [5] proposes an adaptive strategy where group-by pushdown is performed adaptively. Group-by pushdown is a technique where data is already aggregated before distributing it over the nodes, and can heavily reduce networking overhead if there are relatively few keys. However, it also means there is an additional computation step, and may not pay off if the size of the data is not reduced enough. By monitoring by how much the data is reduced locally, the system can decide whether to continue aggregating locally, or just apply the aggregation after grouping globally.

2.2 Micro-adaptive execution

Another angle at AQP is *micro-adaptive execution* [41], which was first proposed and integrated in Vectorwise [52]. Rather than adapting entire query plans, adaptivity happens at the operator level. Vectorwise is a *vectorized* DBMS, which means that operators work on *vectors* of tuples at-a-time, rather than tuple-at-a-time. This ammortized interpretation overhead and provides better data locality, but also provides the opportunity to measure the performance of an overhead at little expense, as most of the work will go towards the actual computation. Operators are implemented by pre-compiled functions called *primitives*. In the micro-adaptivity framework, multiple implementations are provided for a primitive, which are referred to as *flavors*. The execution engine *explores* the different flavors randomly, and continuously tracks their performance, allowing it to find the best performing one, which is then *exploited*. The system only keeps recent performance data, and keeps periodically exploring random flavors even after it all flavors have been explored. This allows the system to deal with non-static cases where the best flavor may change over time.

2.2.1 Multi-armed bandit algorithms

In micro-adaptive execution, the optimization problem is viewed as a *multi-armed bandit* [37] problem. In multi-armed bandit problems, the goal is to maximize some reward through a series of actions. For each action, an agent can choose from a (static) number of options, each with an initially unknown reward. At each round, a unique reward is observed. By repeatedly trying different options, the agent can learn about the most

promising options and ultimately pick the most rewarding one. In the case of micro-adaptivity, the options are different flavors, and the reward is the inverse of the execution time.

Several algorithms for solving multi-armed bandit problems have been proposed [47], such as the *ϵ-greedy* [49] algorithm. The ϵ-greedy strategy works by choosing a random option with a probability of ϵ, and otherwise picking the option with the highest estimated mean, based on the results observed thus far. Vectorwise uses an algorithm based on ϵ-greedy, called *vw-greedy*, which only keeps recent observations to make it more robust for dynamic cases where the best flavor changes over time. VW-Greedy has three parameters. A random flavor is explored every `EXPLORE_PERIOD` primitive calls, for a duration of `EXPLORE_LENGTH` calls. The average cost of the primitive per tuple is then calculated for these calls. After exploring, the best flavor is exploited in separate phases of `EXPLOIT_PERIOD` calls each. After each exploitation phase the average cost is updated, and whatever flavor has the lowest average cost at that point is then exploited.

2.2.2 Large search spaces

In Vectorwise the number of flavors for a primitive is limited, and as such vw-greedy is not optimized for large search spaces. Rosenfeld et al. extend micro-adaptivity to deal with large search spaces in [39]. It tries to tackle the problem by generating only a limited number of alternatives, which are kept in the *working pool*. VW-Greedy only selects flavors from this pool. In between queries, the pool is updated using a search strategy that replaces poorly performing flavors with newly generated ones.

2.3 Query compilation

Data processing systems have mostly been using the Volcano [16] iterator model for the last two decades. In this model, a query plan is represented as a graph of operators, where each operator continuously calls the `next` function of its child operator to produce tuples. This is a nice abstraction that makes operators easily composable, but it suffers from large overheads, which can be attributed to the fact that the `next` function is called once for every tuple produced, both as final result and between operators. This means values are not kept in registers, and many compiler optimizations such as the use of SIMD instructions are made impossible. Because database performance used to be mostly IO bound, this was not perceived as a problem, but with the advent of databases that operate largely or entirely in main memory, that is not the case anymore.

2.3.1 Reducing interpretation overhead

The MonetDB [7] way of dealing with this problem is *bulk processing*, where operators process entire input arrays before moving to the next operator, heavily reducing the number of function calls. Not only does this model reduce interpretation overhead, the tight loops with which operators are now implemented allow for automatic compiler optimizations that are not possible in Volcano style processing, such as the use of SIMD instructions. This idea was later refined as vectorized execution, where the `next` function operates on small

vectors of tuples at-a-time, increasing locality and thereby CPU efficiency. Vectorized execution was first proposed in MonetDB/X100 [6], which later evolved into Vectorwise [52].

A different (and mostly orthogonal) approach is to remove the interpretation overhead altogether by compiling queries directly into machine code. This is not a new idea; System R [1] experimented with query compilation, but abandoned it because this was very expensive to maintain. Other attempts have been made since then, such as [36], which compiles queries into JVM bytecode. However, only recently JIT-compilation is really taking off as a way of improving DBMS performance.

2.3.2 Recent query compilers

One of the most notable examples of a JIT-compiling DBMS is HyPer [29]. HyPer introduced the *data-centric* model as an alternative to the Volcano model. In the data-centric model, tuples are *pushed* towards the root operator, rather than *pulled*. Query plans are subdivided into pipelines, which are defined by operators between materialization points (such as a sort or aggregation operation). The system produces tight loops in which tuples are processed one at-a-time. The goal is to minimize data movement by keeping values in registers as long as possible, and HyPer manages to obtain excellent performance using this approach. HyPer uses the LLVM [26] compilation framework to produce efficient machine code.

Commercial examples include (among others) Hekaton [13] and Impala [48]. Hekaton is an in-memory database engine which is integrated in Microsoft SQL Server [28]. For the Hekaton project, Freedman et al. set as a design goal to increase throughput by 10-100X. To reach that goal they (i) optimized indexes for main-memory, (ii) removed all latches from data structures and (iii) implemented JIT-compilation for stored procedures. Several layers of abstractions are used: SQL is compiled into a *mixed abstract tree* (MAT) (similar to an abstract syntax tree), which is compiled to a *pure imperative tree* (PIT). The PIT in turn is compiled into C code, which is finally compiled into assembly. The decision to first compile from MAT to PIT, rather than from MAT to C, was made because of the large gap between SQL and C. For instance, C does not support null values or warnings for arithmetic errors such as integer overflow. Hekaton produces code that is contained by a single function, and operators are connected solely using labels and `goto`-statements, in an effort to keep values in registers as long as possible. Unlike HyPer, Hekaton does not go to lengths to keep compilation times down, but they report that most queries compile in less than a second. This is deemed sufficient, as compilation only happens for stored procedures (which, of course, makes the system somewhat limited).

Impala is an MPP database based on Hadoop [43]. It has an interpreted execution engine, but can JIT-compile certain functions to specialize them for the query at hand, by removing conditional statements and replacing load instructions with literals where possible, as well as replacing virtual function calls with the actual function calls. Although it significantly improves function performance, overall speedups are relatively limited because functions are still called on a tuple-at-a-time basis.

2.3.3 Abstraction without regret

There is a large semantic gap between a high-level declarative SQL query and low-level imperative machine code, which makes developing a query compiler far from trivial. Hekaton uses multiple layers of abstraction to bridge this gap. HyPer in turn introduces relatively high-level abstractions in C++ that are optimized away at compile-time to ease development [31]. Yet, in [23], Koch argues that these systems still work on a level of abstraction that is too low. Most databases (including HyPer, Hekaton and Impala) are implemented using low-level programming languages (PL) such as C/C++, which is deemed necessary because of the perceived low performance of high-level PLs. However, the downside of using a low-level PL is that it can hurt programmer performance, and introduce hard-to-find bugs.

Koch argues that another downside of using a low-level PL is in fact a *decrease* in performance, because (i) too much developer time is wasted on the intricate details of using a low-level language rather than the implementation of features and optimizations, and (ii) some automatic optimizations (such as loop fusion) are only realistic in higher levels of abstraction. He therefore proposes a principle called *abstraction without regret*. Rather than implementing the system in a low-level PL from the start, multiple levels of abstraction are needed, so that the system can be defined in a high-level abstraction and lowered to machine code one step at a time. *Specific* compiler optimizations should be defined at each level of abstraction, such as loop fusion at higher levels, and register allocation at the lowest level. By moving the burden of writing performant code to the compiler, the system as a whole can be more performant than when directly written in a low-level PL, while retaining the abstraction benefits of high-level PLs.

2.3.4 High-level query compilers

The first project to follow the abstraction without regret philosophy is LegoBase [22]. Klonatos et al. identified a number of problems with the state-of-the-art query compilers. Virtually all of them are based on the concept of *code template expansion*, in which code is generated in a single step from a query plan by replacing an operator node with a code template. Template expansion is hard to maintain, and can make cross-operator optimizations impossible. Moreover, all previous query compilers only compile queries, but not the system itself, missing inlining opportunities. Finally, although systems such as LLVM support runtime optimizations, many optimizations (such as AQP) are only possible using abstractions that operate on a much higher level than LLVM (as we have seen in the previous subsection).

To account for these problems, LegoBase was developed in Scala [45], a high-level (impure) functional PL. The Scala code is compiled to optimized, low-level C code which is specific to a given SQL query. *Generative programming* is applied, which allows automatic removal of abstraction overhead and supports optimizations on multiple abstraction levels. It is set apart from the competition by *continuous runtime optimization* of the whole query engine, meaning that the entire system is recompiled based on runtime information, rather than only queries. LMS [38] is used as the compilation framework, which can optimize Scala code on-the-fly, but was adapted by LegoBase to also compile to C code. The

authors claim to significantly outperform HyPer, but to gain these results they needed to resort to non-TPC-H-compliant indexing and precomputations [44]. Also, LegoBase does not support parallelism. Finally, it should be noted that the additional abstractions do seem to come at a price: a compilation time of about 2s is reported [22] for compiling TPC-H Q5, as opposed to 34ms in HyPer [29]. Similar differences can be seen for other TPC-H queries (unfortunately, [29] only reports the compilation times of a limited number of queries).

How to Architect a Query Compiler

How to Architect a Query Compiler [42] provides a principled methodology for building query compilers, based on the abstraction without regret principle. The methodology is demonstrated by the development of DBLAB, which is a re-implementation of LegoBase. Like LegoBase, the system is built in Scala, uses LMS, and only supports single-threaded execution. Shaikhha et al. again argue for a multi-pass compiler, progressively lowering high levels of abstraction into low-level machine code. The paper was recently revisited by Tahboub et al. in [44], which demonstrates that the added complexity of multiple compiler passes is in fact unnecessary, even when using a high-level PL. The result is L2B, a DBMS implemented in Scala, using single-pass compilation.

The main idea of L2B comes from Futamura projections [14]. Futamura found that "compilation can be profitably understood as *specialization* of an interpreter, without explicit reference to a given hardware platform or code generation strategy" [44]. Specializing an interpreter to a query boils down to removing all abstractions until all that is left are the minimum number of operations that are required to execute the query. L2B implements this concept by first implementing a traditional query interpreter in Scala. Using LMS, the interpreter can be specialized (partially evaluated) given a query, which produces optimized C code. The C code is then compiled into machine code and executed. Specialization goes as far as simple data structures, e.g. hash maps and records are compiled into native C types and arrays. In this model, the task of the programmer is mostly limited to writing an interpreter[1], while the compiler is responsible for writing efficient code. The authors claim that L2B is the first system developed in a high-level language to match or even beat the performance of HyPer.

2.3.5 Weld

Another data processing framework that uses JIT-compilation is Weld [32]. Weld is a runtime and programming language aimed at improving the performance of data-intensive applications, by optimizing across libraries. This is done using the combination of a lazily evaluated intermediate representation (IR), a runtime API, and an optimizer. Libraries represent their computations in the IR, and submit them to Weld through the API. When a user forces evaluation (e.g. printing a result), the IR is optimized, compiled to machine code using LLVM, and executed. The optimizer performs optimizations such as loop fusion and SIMDization. Because Weld code is lazily evaluated, it can perform optimizations across functions that otherwise would not be possible, such as fusing a filter and map

[1] Some care needs to be taken to make sure efficient machine code can be derived from the interpreter.

function into a single loop. This means that less data will be materialized, improving overall performance. In a sense it follows the abstraction without regret principle: Weld defines a high-level DSL which enables optimizations that are not possible in the much lower level LLVM it produces. However, much like systems such as HyPer, the engine relies on template expansion to generate code.

2.4 Contributions

While AQP and JIT-compilation can both offer significant speedups in database systems, little work has been done on the intersection of the two. A notable exception is [24]. When a query is complex to compile, but quick to execute, the compilation time can easily dominate the total runtime, at which point interpretation is more efficient. For longer running queries compilation may be much faster. Kohn et al. present a strategy that adaptively decides to interpret a query or compile it to machine code. Compiled queries are initially not optimized by the compiler, but the system can also adaptively decide to do so if the query runs long enough.

Although it introduces adaptivity in a JIT-compiling system, it does not do so at the execution plan level. Integrating adaptivity in a JIT-compiling system is challenging because the number of variants to choose from and to compile increases exponentially in the number of adaptive choices that are made in a pipeline. Also, JIT-compilation typically processes data tuple-at-a-time, which makes it harder to measure the performance of an operator while keeping bookkeeping costs down. This research is a first attempt[2] to integrate adaptivity into Weld, a JIT-compiling data processing framework, and this is therefore our main contribution. Our main focus is on keeping compilation times down, as well as defining operator flavors in Weld. Furthermore, we introduce the concept of *loop-adaptivity*, which is an extension of micro-adaptivity that is better suited to the data-centric nature of typical query compilers.

[2] A recent paper ([33]) does introduce adaptivity in Weld, in the form of adaptive hash tables as well as adaptive predication based on measured predicate selectivity. However, the first is entirely pre-compiled and therefore not novel, while the second only considers extremely limited cases, where `for`-loops contain only a single predicate. It also only considers static cases.

Chapter 3

Weld

Weld [32] is a runtime and programming language aimed at improving the performance of data-intensive applications, by optimizing across libraries. This is done using the combination of a lazily evaluated intermediate representation (IR), a runtime API, and an optimizer. In this chapter, we will take a look at the implementation of Weld. We will summarize the IR, the optimizer, code generation and the runtime.

3.1 Overview

Weld consists of roughly three components: an IR in which users express their computations, a runtime API through which the computation is submitted, and finally an optimizer that optimizes the IR and emits machine code before evaluation. The parser and optimizer are implemented using Rust, which provides constructs such as pattern matching that eases the development of a compiler. Code is generated using LLVM [26], while the Weld runtime is mostly implemented in C++.

3.2 Internal representation

Computations in Weld are expressed using Weld's own IR. The IR was designed with three goals in mind:

1. Generality: the language should be general enough in order to be usable for a wide range of data analytics applications.

2. Optimizability: the language should be well suited for optimizations such as loop fusion and SIMDization.

3. Parallelism: the IR should be explicitly parallel so that efficient parallel machine code can be automatically generated from the IR.

The result is a small language inspired by functional programming languages. In this section we will give a brief (and incomplete) overview of the core of this IR.

Table 3.1: Value types in Weld

Type notation	Description
`bool`	A boolean value
`i8`, `i16`, `i32`, `i64`	Signed integers of 8, 16, 32 and 64 bits respectively
`u8`, `u16`, `u32`, `u64`	Unsigned integers of 8, 16, 32 and 64 bits respectively
`f32`, `f64`	32 and 64 bit precision floating point values
`vec[T]`	Vector with elements of type T
`dict[K, V]`	Dictionary with keys of type K and values of type V. Keys must be scalars or structs of scalars.
`{T1, T2, ..}`	A struct where the members are of type T1, T2, and so forth.
`simd[T]`	A SIMD register with a fixed number of values of type T

3.2.1 Data types

The Weld IR is a functional programming language based on expressions. Because it is functional, only constant types are supported. Data types can be roughly divided into "value" data types and "builder" data types. Value types contain scalars such as integers and floats, and collections in the form of vectors (growable arrays), dictionaries (hash tables) and structs. More complex collections can be formed by nesting them. For instance, the result of a group by operation would typically be a dictionary with a scalar or struct as key type and a vector as value type. Other value types include strings, which are implemented by vectors of bytes, and the SIMD type which represents a fixed number of scalars that can be used with SIMD operations, similar to the vector type in LLVM [26]. Table 3.1 shows a full list of value types supported by Weld.

Builder types are used to construct new value types in parallel, by "merging" values into them in parallel and finally producing some result. The type of result depends on the type of the builder. For instance, `appender[T]`s construct vectors of type T, while `merger[T,+]`s can be used to generate the sum of values of type `T`. Table 3.3 lists the available builders in Weld.

`merger`, `dictmerger` and `vecmerger` all apply a commutative binary operation on the values that are merged into it. Currently, Weld supports addition, multiplication, minimum and maximum operations.

3.2.2 Basic expressions

Weld supports a relatively limited number of expressions and functions. We will name the most relevant ones to us below:

- Operators: Weld supports common binary and unary operators, both arithmetic (`+`, `-`, `/`, `*`, `min`, `max`, `pow`), logical(`>`, `<`, `>=`, `<=`, `==`, `!=`, `&&`, `||`) and bitwise (`&`, `|`, `^`).
- Let-expressions: Let-expressions are of the form `let C = E1; E2`. This evaluates `E1`, assigns its value to a constant named `C`, and then evaluates `E2` and returns its result.
- Conditional expressions: Conditional expressions can be of two forms: `if(C, T, F)` which first evaluates `C`, and then branches to either `T` or `F` and returns its result, depending on the value of `C`. `select(C, T, F)` is the predicated version which always

Table 3.3: Builder types in Weld

Type notation	Description
`appender[T]`	Appends values of type T to an (initially empty) `vec[T]`.
`merger[T], binop`	Applies a commutative binary operation on values of type T, resulting in an aggregated value of type T.
`dictmerger[K, V, binop]`	Groups structs of {`K, V`} by K, and applies `binop` on the groups, resulting in a `dict[K, V]`
`groupmerger[K, V]`	Groups structs of {`K, V`} by K, resulting in a `dict[K, vec[V]]`
`vecmerger[T, binop]`	Groups structs of {`i64, T`}, where the first value is an i64 which indexes some predefined vector. Applies `binop` on each group, and results in a `vec[T]`

- Functions: functions in Weld are lambdas, i.e. they are anonymous. Functions are passed as arguments to a variety of expressions, and a full Weld program is also defined as a single lambda. The syntax is as follows: `|arg1, arg2, ..| body`, where the arguments are names that can be referred to in `body`. A function returns the value that `body` evaluates to.

3.2.3 Builder expressions

Builders are used in three expressions, which form the core of the language: `merge`, `for` and `result`. `merge(b, v)` is used to add a value `v` to a builder `b`, and returns a new builder representing the result. The old builder is consumed in the process. `for(vector, builders, func)` also consumes and produces builders. The `for`-expression applies a function of type `(builders, index, element) => builders` on each element of vector to merge values into one or more builders in parallel. Because the `for`-expression accepts an arbitrary number of builders as argument, a single `for`-loop can be used to compute multiple values, such as a vector and a sum. Finally, `result(b)` can be called on a builder to produce the value that should be produced from the previously merged values. `result` should only be called once on a builder, although this is not enforced by Weld.

3.2.4 Examples

Listing 3.1 and 3.2 show code snippets demonstrating the use of builders. Note how the the two code blocks are almost identical except for the type of builder, and therefore yield a very different result. Listing 3.3 demonstrates a `for`-loop that uses two builders: one to compute the squares of the elements of some input vector, another to compute its sum.

Listing 3.1: Building a vector.

```
1 # Evaluates to [1, 2, 3]
2 let b1 = appender[i32];
3 let b2 = merge(b1, 1);
4 let b3 = merge(b2, 2);
5 let b4 = merge(b3, 3);
6 let vec = result(b4);
```

Listing 3.2: Summing integers.

```
1 # Evaluates to 6
2 let b1 = merger[i32, +];
3 let b2 = merge(b1, 1);
4 let b3 = merge(b2, 2);
5 let b4 = merge(b3, 3);
6 let sum = result(b4);
```

Listing 3.3: Squaring a vector element wise and summing the vector in a single loop.

```
1 # Evaluates to {[1, 4, 9], 6}
2 let v = [1, 2, 3];
3 result(
4     for(
5         v,
6         {appender[i32], merger[i32,+]},
7         |bs,i,e|
8             {merge(bs.$0, e * e), merge(bs.$1, e)}
9     )
10 )
```

A more complicated example is shown in listing 3.4. This is an example of a full Weld program that joins a table `R` and `S` on `R_k` = `S_k`, and selects columns `R_k`, `R_a` and `S_b`. Note that the program is a lambda, which takes the input data as arguments. First, a hashtable mapping keys from `S_k` to values of `S_b` is created using the `groupmerger` builder. Note the `zip`-expression at line 4, which can only be used in `for`-expressions, and maps an arbitrary number of vectors into a vector of structs. Hence, `e` at line 6 is a struct with an element of `S_k` as the first member, and an element of `S_b` as the second member. When the hash table is created, we iterate over `R`. For each row, we check if there are rows in `S` with `S_k` = `R_k`. If so, the values are retrieved using the `lookup`-expression, and the group is iterated over in another `for`-expression. This `for`-expression simply appends the appropriate columns to the result. If the table does not contain any values for `R_k`, an unchanged builder is returned, which means that no values are appended during that iteration.

Listing 3.4: Basic examples of builder expressions

```
1  |R_k:vec[i32], R_a:vec[i32], S_k:vec[i32], S_b:vec[i32]|
2      # Build a dictionary that maps S_k to S_b
3      let s_table = result(for(
4          zip(S_k, S_b),
5          groupmerger[i32, i32],
6          |b,i,e| merge(b, e)
7      ));
8      result(for(
9          zip(R_k, R_a),
10         appender[{i32, i32, i32}],
11         |b,i,e_r|
12             # Check if the current R_k is in the dictionary
13             if(keyexists(s_table, e_r.$0),
14                 # Append items in group on true
15                 let group = lookup(s_table, e_r.$0);
16                 for (group, b, |b,i,e_s|
17                     merge(b, {e_r.$0, e_r.$1, e_s})
18                 ),
19                 # Return unchanged builder on false
20                 b
21             )
22     ))
```

3.2.5 Annotations

Weld also allows expressions in the IR to be annotated using the following syntax: `@(key1: value1, key2:  value2, ...)`. This can be done to give hints to the optimizer about what kind of optimizations should be performed, or the kind of builder implementation that should be used.

3.3 Optimizer

The second main component of Weld is called the optimizer, but encompasses more than just IR optimization. It is responsible for parsing strings of Weld code that are submitted through the API into an abstract syntax tree (AST). The optimizer then proceeds to apply general optimizations on the IR, such as loop fusion and SIMDization. When all the optimization rules have been applied, the Weld IR is compiled into another IR, referred to as Sequential Internal Representation (SIR), which in turn is compiled into LLVM IR. Finally, the LLVM IR is compiled by the hardware backend into executable machine code.

3.3.1 Optimization passes

The strength of Weld comes from its ability to optimize across libraries and function boundaries. These optimizations are modeled after LLVM, which applies hardware independent optimizations on its IR before passing the program to a hardware backend. Optimization happens during optimization passes, each of which can be uniquely enabled or disabled. Passes are implemented using pattern matching rules on (sub-trees of) the

AST. Passes are performed in a predefined order (which can be altered by the user), and each pass is repeated until the AST does not change anymore. The following passes have been implemented in Weld:

- Inliner: mandatory optimization pass that inlines function calls and `let`-expressions.
- Loop fusion: fuses loops horizontally and vertically. Horizontal loop fusion implies fusing two loops over the same vector into a single `for`-loop. Vertical loop fusion means fusing two loops where one loop produces the input of the second loop, such as a filter operation followed by a map operation.
- Unroll static loop: unrolls small loops into a series of `lookup`-expressions. Requires that the loop size is known at compile time.
- Infer size: infers the size of output vectors statically, allowing for more efficient memory allocation.
- Short circuit booleans: transforms `if`-expressions of the form `if(C1 && C2 .., T, F)` into `if(C1, if(C2, T, F), F)`.
- Predicate: transforms `if`-expressions annotated with the `predicate` annotation into equivalent unconditional `select`-expressions.
- Vectorize: transforms `for`-loops with simple inner bodies into `for`-loops with SIMD data types, which are later compiled into SIMD instructions.

3.3.2 Code generation

After the IR has been optimized as much as possible, IR is turned into machine code. Because the gap between LLVM IR and the rather functional style of Weld is quite large, Weld is first compiled into another IR called Sequential Internal Representation (SIR). A SIR program is composed of a series of functions, which are composed of blocks of statements and function arguments. Blocks of statements are always terminated by terminator statements, which include jumping to another function, a `for`-loop, or terminating the program and returning a result. Listing 3.5 shows an example of a simple Weld program that computes the squares of a vector `x`. Listing 3.6 shows the SIR code this compiles into.

Listing 3.5: Weld program that computes the squares of each element in a vector

```
1 |x:vec[i32]|
2     result(
3         for(
4             x,
5             appender[i32],
6             |b,i,e| merge(b, e * e)
7         )
8     )
```

Listing 3.6: SIR program that computes the squares of each element in a vector

```
1  F0:
2  Params:
3      x: vec[i32]
4  Locals:
5      fn0_tmp: appender[i32]
6  B0:
7      fn0_tmp = new appender[i32]()
8      for [x, ] fn0_tmp b i x__1 F1 F2 true
9
10 F1:
11 Params:
12     fn0_tmp: appender[i32]
13     x: vec[i32]
14 Locals:
15     b: appender[i32]
16     fn1_tmp: i32
17     i: i64
18     x__1: i32
19 B0:
20     fn1_tmp = * x__1 x__1
21     merge(b, fn1_tmp)
22     end
23
24 F2:
25 Params:
26     fn0_tmp: appender[i32]
27 Locals:
28     fn2_tmp: vec[i32]
29 B0:
30     fn2_tmp = result(fn0_tmp)
31     return fn2_tmp
```

The SIR program consists of three functions: `F0`, `F1` and `F2`. `F0` is the entry point of the program, and takes as parameters the input data of the Weld program, in this case the vector `x`. It initializes a new `appender fn0_tmp`, and then terminates with a parallel for statement. The for statement takes two functions as argument: a body function and a continuation function (`F1` and `F2` respectively). `F1` is repeatedly called on every element of the input vector, until the whole vector has been processed. Finally, the continuation function `F2` is called, which produces the vector of squares with the result statement, and returns it.

3.3.3 LLVM

After generation of the SIR program, LLVM IR can be generated. The Weld team decided against using the LLVM libraries for code generation, but instead chose to "manually" emit LLVM IR using string concatenation. The obvious downside of this approach is that this is relatively slow, as the IR string has to be parsed by the LLVM backend directly after producing it before the program can be compiled.

The LLVM program produced by Weld largely follows the same structure as the SIR

program. Each function `F`x in SIR corresponds to a function `@f`x in Weld. On top of that, some extra functions are generated that are meant as callbacks for the runtime. Every Weld program contains a function `@run(%i64 input)`. This function extracts its input data from the provided pointer, calls the runtime to start execution, and finally collects the result of the computation and returns it. After generating the LLVM IR it is passed to the LLVM backend, which performs its own optimizations on the code, and finally generates machine code.

3.4 Execution engine

The last component of Weld is the runtime API. The API is responsible for collecting Weld code fragments, passing it to the optimizer, and finally executing the resulting code. In this section we will focus on the implementation of the latter.

Every Weld program starts with a call to `weld_run_begin`, which takes a function and some data for that function as arguments. `weld_run_begin` creates a data structure of type `work_t`, which represents a task to be executed. A `work_t` has pointers to the task function and data, and tracks the lower, upper and current index of loop iterations if the task is a loop body. The task is then pushed onto the executing thread's local work queue, and popped again to start execution of the program.

3.4.1 Parallelism

In principle, all code is run sequentially, until a `for`-statement is executed. A for statement can either be executed sequentially, by directly calling its body and continuation functions, or in parallel, by calling the runtime function `weld_rt_start_loop`. The loop is executed in parallel when either:

- the loop is bigger than the minimum *grain size*[1];
- the loop has an inner loop;
- the loop is specifically annotated to always use the runtime.

Note that only inner loops are ever run sequentially, even if an outer loop has a very small number of iterations. `weld_rt_start_loop` is responsible for creating tasks for the loop, and pushing them on the queue. If the loop is an outermost loop, two tasks are created: a loop body task and a continuation task. The body task has a pointer to the loop body function, and tracks the lower and upper bound of the vector that the function may iterate over. The continuation task has a pointer to the function that should be run after the loop has finished execution. Only the body task is pushed onto the queue, but the body task has a pointer to the continuation task so that it can be pushed onto the queue as soon as the body task is finished. This ensures that, in a multithreaded environment, the continuation task is never run before the end of the body task. If the loop is nested

[1] The grain size is a user controllable variable that defines the number of iterations that should be processed by a single sub-task, which we will discuss shortly.

inside some other loop, the current task (which relates to the outer loop) is aborted, and a new task is created for the remaining iterations. This ensures that `merge`s are correctly ordered.

To parallelize the work of a `for`-loop, a work-stealing strategy is employed. Before execution, tasks are split so that no task is larger than a user controllable grain size, and pushed on the thread's local queue before proceeding execution with the first task. When no more tasks are left on the local queue, the worker repeatedly tries to steal tasks from a random victim until it either received a new task, or the program terminated. Upon completing a task, `finish_task` is called. `finish_task` checks if there are any sibling tasks still executing. If not, the continuation task is pushed on the queue, and execution continues. If no continuation task exists, the program terminates, after which the result can be retrieved by means of another runtime call.

Chapter 4

Opportunities for Loop-Adaptivity

Micro-adaptivity was first proposed in Vectorwise [52]. In [40] Răducanu et al. defined micro-adaptivity as "the property of a DBMS to continuously choose between primitive flavors at runtime with the goal of minimizing the overall query execution time, choosing the most promising flavor based on real time statistics". A primitive in Vectorwise is a function that implements some operator (such as selection), and typically operates on a large number of tuples (a *vector*) at-a-time to amortize interpretation overhead. A primitive can have multiple *flavors*: different implementations of a primitive that compute the same result in different ways. Primitives are well-suited for micro-adaptivity: because they work on vectors rather than single tuples, the overhead of measuring is automatically amortized. Moreover, because a primitive implements a single operation, the number of alternative flavors is limited.

Unlike Vectorwise, Weld does not have the notion of primitive functions that operate on multiple tuples at-a-time. Weld's approach to eliminating interpretation overhead is JIT-compilation, allowing it to fuse many operators together into a single pipeline. Unfortunately, this prevents us from measuring the performance of a single operator, as the associated overhead would be too costly. Instead, we will look at a bigger scale: parallel `for`-loops (a concept which we call loop-adaptivity). As mentioned in the previous chapter, loop bodies are executed by tasks, which are subdivided into a large number of smaller tasks, each operating on their own part of the input data. By measuring the performance of a single subtask we can find out how a pipeline of operators is performing while keeping overhead low. Therefore, when we refer to flavors in Weld, we mean a set of different implementations for the same loop body function.

The downside of this approach is that identifying flavors becomes less obvious. Instead of a well-defined set of primitives, we can have an infinite number of different `for`-loops, and consequently an infinite number of flavors of these loops. In this chapter we will look at potential flavors for loop bodies in Weld. We are interested in flavors that perform better than other flavors in some cases, but not all, motivating the use of loop-adaptivity. We will only look at relatively simple loops, for which the number of flavors will be limited. However, the system will ultimately have to be able to deal with complex loops with a high number of flavors.

4.1 Experimental setup

Table 4.1 shows basic information about the machines that were used for the experiments. For each experiment a number of different flavors of the same query were run for varying selectivity values. Each experiment was repeated 5 times, and for each experiment only the minimum execution time was recorded, ignoring compilation times. The queries were run on data with varying scale factors on machine 1 and machine 2, while machine 3 only ran the smaller scale factors due to its limited memory size. Machine 1 has two of the CPUs listed in the table, but `numactl` was used to make sure only one was ever used.

Table 4.1: Test machine details

Name	CPU	Clock Speed	#Threads	RAM
M1	Intel Xeon E5-2650	2.0 Ghz	16	256 GB
M2	Intel Core i7-2600K	3.4 Ghz	8	16 GB
M3	Intel Core i5-6360U	2.0 Ghz	4	8 GB

4.2 Selective versus full computation

Many real-world queries project and select data. This can be done using two separate loops: the first filters the data and stores it in an intermediate array. The second loop iterates over the intermediate array, performs some computation on each element, and stores it in the final result. Weld optimizes operations like these by fusing the two loops into a single loop. This loop iterates over the input array, checks the predicate and immediately projects the tuple if it satisfies the predicate. This is generally more efficient because the intermediate results are not materialized.

However, if the selection expression cannot be predicated, the loop fusion optimization denies the possibility of SIMDizing the projection, another important optimization that Weld applies. Therefore, in some cases it may actually be worthwhile to keep the loops separate, so that the projection part can be SIMDizing. We expect this to be true when execution cost of the projection calculations is higher than the time needed to materialize the intermediate results. Another factor that should affect the effectiveness of SIMD is the size of the data type. The fewer bytes that are needed to store a value, the more values fit in a SIMD register, improving overall performance.

4.2.1 Flavors

As an example, take a query that computes the product of each element in two columns, but only if the element in the first column equates to 42. Listing 4.1 shows the Weld code for a fused loop.

Listing 4.1: Fused selection and projection loop

```
1  result(for(
2      zip(v1, v2),
3      appender[i32],
4      |b,i,e| if(e.$0 == 42, merge(b, e.$0 * e.$1), b)
5  ))
```

Because `merge`-expressions that return an `appender` cannot be predicated by Weld, the loop will keep its `if`-expression, which cannot be SIMDized. Listing 4.2 shows Weld code where the same query is implemented using two different loops: the first filters the data, while the second performs the projection.

Listing 4.2: Filter first, unvectorizable projection

```
1  let filtered = result(for(
2      zip(v1, v2),
3      appender[{i32, i32}],
4      |b,i,e| if(e.$0 == 42, merge(b, e), b)
5  ));
6  result(for(
7      filtered,
8      appender[i32],
9      |b,i,e| merge(b, e.$0 * e.$1)
10 ))
```

Unfortunately, neither of these loops can be SIMDized either. The problem here is that Weld cannot SIMDize a loop when its input is a vector of structs, but only when its input is a struct of vectors with scalar elements (using the `zip`-expression). The Weld code that achieves just this is shown in listing 4.3. Here, the first loop explicitly generates two different vectors, rather than a single vector that contains both columns. The downside of this is that we have more `merge`-expressions in the selection loop, which implies more overhead. The upside is that the projection loop can now be SIMDized.

Listing 4.3: Selection first, vectorizable projection

```
1  let builders = result(for(
2      zip(v1, v2),
3      {appender[i32], appender[i32]},
4      |bs,i,e| if (e.$0, {merge(bs.$0, e.$0), merge(bs.$1, e.$1)}, bs)
5  ));
6  result(for(
7      zip(result(builders.$0), result(builders.$1)),
8      appender[i32],
9      |b,i,e| merge(b, e.$0 * e.$1)
10 ))
```

Another approach is *full computation* [41]. Rather than performing the selection first, we can also compute the projection on the input vectors in its entirety, and selection afterwards. Although this means that the projection loop will perform some computations that are unneeded, it may be faster if the predicate selects most of the data. Moreover, this implementation does not force us to use more `merge`-expressions than strictly necessary, reducing the associated overhead. Listing 4.4 shows the corresponding Weld code. Note

that the builder in the projection loop takes the length of the input vector as an argument. This tells Weld that the vector is a fixed size array, rather than a growable one.

Listing 4.4: Full computation

```
let mapped = result(for(
    zip(v1, v2),
    appender[i32](len(v1)),
    |b,i,e| merge(b, e.$0 * e.$1)
));
result(for(
    zip(v1, mapped),
    appender[i32],
    |b,i,e| if (e.$0 == 42, merge(b, e.$1), b)
))
```

Because of Weld's limitations we do not expect the unfused filter first strategies to perform better than a fused loop. The first strategy simply cannot be SIMDized, and therefore will only suffer from the additional materialization, while the second strategy suffers from even higher materialization costs. However, full computation may be faster overall if selectivity is high enough.

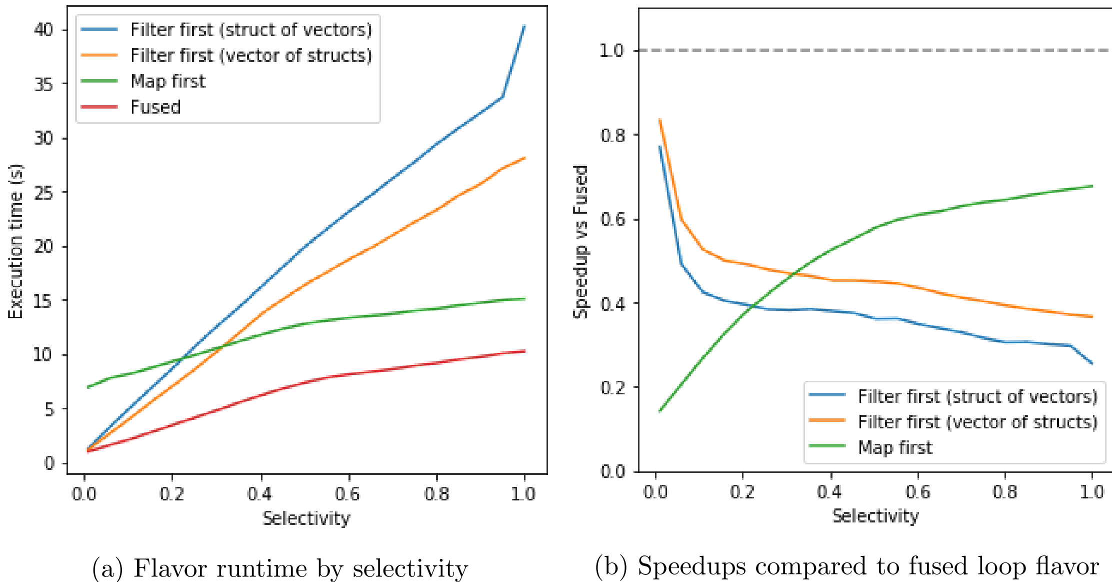
4.2.2 Experiment

Figure 4.1a shows how the execution time of the different flavors that were discussed above depend on query selectivity. The query that was run is essentially the same as the examples used above, although it now consumes 6 input columns rather than 2, and contains a more complicated projection (11 multiplications and 1 addition). Figure 4.1b shows the speedup of each flavor compared to the fused variant. As expected, the fused loop performs better than filtering first. Among the two different strategies for filtering first, outputting a struct of vectors is even worse than the outputting a vector of structs, despite the performance gains from SIMDization. This has to do with the high overhead of merges using an `appender`, which we will discuss in the next subsection. Although full computation performs better for higher selectivity values, at no point it comes close to the fused variant.

4.2.3 Discussion

Unfortunately, this case is not a candidate for adaptive execution, as the fused flavor seems to be always better than other flavors. This contradicts the result in [40]. We suspect that this has to do with poor performance of merge operations of `appender` types, which makes materializing intermediate results too expensive. We identified the following problems in the LLVM code generated by Weld that likely contribute to these materialization costs:

- For every `merge` operation, an `appender` type calls a runtime function to get pointers to its size, capacity and data. Even if this is inlined, it still appears to cost about 10 LLVM operations per `merge`. There is no reason for doing this for every loop iteration, and moving the operations out of the loop should decrease the overall merging overhead.



(a) Flavor runtime by selectivity

(b) Speedups compared to fused loop flavor

Figure 4.1: Execution time and speedup for the different select-project flavors.

- For every `merge` operation, the capacity of the vector and the current number of elements in it are compared. If the vector is full, the capacity is doubled and memory is reallocated accordingly. The capacity check is therefore done N times in the worst case, where N is the size of the vector that is iterated over. This could be improved to only $\log(N)$ iterations by creating an inner loop that, after doubling capacity, will iterate as long as the remaining capacity without performing intermediate capacity checks.

- Even when a fixed-size `appender` is used (as is the case for the map first strategy), the capacity check is performed. However, since fixed size vectors are never resized, this could be removed entirely.

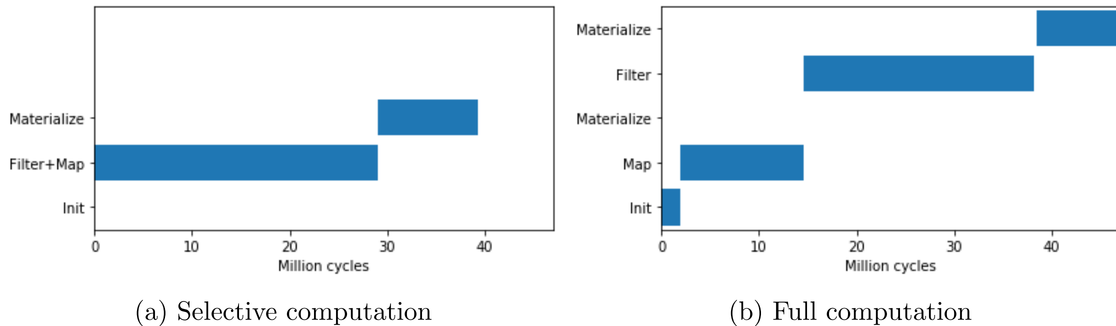
- The typical way of outputting multiple columns is by producing a vector of structs, rather than multiple vectors[1]. The problem with this approach is that in these cases a SIMD register cannot be stored in the vector by a single store operation, because the values of a single column are not stored consecutively. Instead, the SIMD register has to be unpacked, and each value has to be stored separately. This reduces the benefits of SIMDization.

- Some of the above mentioned problems could likely be solved by the compiler, but the generated LLVM code lacks the necessary compiler hints to make this possible, such as `const` or `noalias` attributes.

Figure 4.2 shows a timeline of the different tasks of the fused loop (selective computation) as well as the full computation variant, where selectivity is set to 1. The y-axis shows

[1]As we have seen, building a struct of vectors is more expensive than building a vector of structs, probably because of the aforementioned issues with `appender`.

the tasks that are run for each variant. The initial task for selective computation performs little work, as all it does is calling the runtime to start the fused selection-projection-loop. The materialize task materializes the builder into a vector. The full computation variant has more tasks. The map and filter tasks are responsible for the map and filter loops respectively. Materializing the builder from the map-loop is virtually free, because this builder is a fixed-size `appender`, and so all memory has been allocated up-front. The costs for this allocation is reflected in the additional runtime in the initial task.



(a) Selective computation (b) Full computation

Figure 4.2: Timeline of different tasks for selective computation and full computation.

We can see that, despite moving the computation out of the loop, the selection loop is barely faster than the fused loop, despite the high number of projection operators versus a single conditional operator. This again suggests that the bulk of the execution time goes towards materialization. The projection loop is considerably faster than the selection loop, likely because SIMDization actually has a positive effect on materialization overhead as well: the SIMDized loop will iterate w times less than its equivalent scalar loop, where w is the number of values that fit in a SIMD register. This results in w less load operations and capacity checks.

SIMDization

Another issue we encountered is the quality of the generated SIMD code. First of all, Weld currently only supports LLVM vector types (a datatype in LLVM that represents a SIMD register) that contain 4 values. This is a conservative approach and does not take advantage of modern hardware that support SIMD registers with widths up to 512 bits (and so can contain 4 times as many 32 bit integers). Moreover, upon inspecting the generated machine code we found many unnecessary shuffle in between the actual multiplication operations. According to the Weld team this is because Weld does not instruct LLVM to target the specific feature set that the host machine has, but instead assumes a conservative default target with a very limited number of SIMD features.

4.3 Branching versus predication

The second potential case for loop-adaptivity that we examined was conditional branching versus predicated operators. Branching operators first evaluate a predicate, and then

execute one of two branches, depending on whether the predicate is true or not. Branching is implemented in Weld by the `if`-expression. Its predicated counterpart is `select`. Unlike `if`, `select` always executes both branches, but only returns the value of one of them, again depending on a predicate. The major advantage of `select`-expressions is that they can be SIMDized, unlike `if`-expressions.

4.3.1 Branch prediction and selectivity

The execution of a CPU instruction goes through a number of phases. Among other things, an instruction needs to be fetched, executed, and a result is written back to a register. To increase throughput, modern CPUs try to perform as many of these tasks at the same time, e.g. while some instruction is executing, the next is already being fetched. This technique, referred to as pipelining, works well as long as the CPU knows which instruction will be executed next. However, this is no longer true when future instructions depend on the outcome of earlier instructions, as is the case for conditional branching. In this case the CPU has to predict which branch is most likely to be taken. If it is right, it will benefit from pipelining, but if it made a misprediction the incorrect instructions will have to be rolled back, causing a delay.

Branch prediction is based on the history of the branches taken. Branch prediction will be accurate if a branch is rarely taken or taken most of the time. However, when a branch's selectivity is around 50% it becomes hard for the CPU to predict the next branch that will be taken, resulting in relatively poor performance. Unlike branching expressions, the performance of predicated expressions does not depend on the selectivity of its predicate. Therefore, when the CPU has a hard time predicting the correct branch, a predicated expression may be faster than a branching one. On the other hand, when a CPU can accurately predict a branch, the predicated variant may perform worse because it is doing more work than necessary. The choice between branching and predication could thus very well be something that should be done adaptively.

4.3.2 Flavors

In this case, we are dealing with two simple flavors, where each flavor corresponds to a different expression: if and select. Listing 4.5 shows Weld code that counts the number of elements with a value of 42 in a vector using the classical branching approach, while listing 4.6 shows its predicated variant.

Listing 4.5: Counting 42s using branching

```
1  result(for(
2      input,
3      merger[i32,+],
4      |b,i,e| if (e == 42, merge(b, 1), b)
5  ))
```

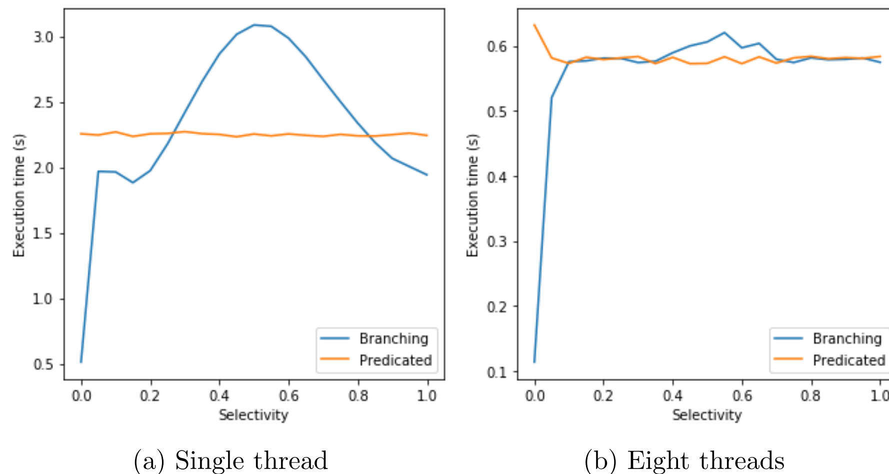
Listing 4.6: Counting 42s using predication

```
1 result(for(
2     input,
3     merger[i32,+],
4     |b,i,e| merge(b, select(e == 42, 1, 0))
5 ))
```

The branching variant simply checks the predicate, and merges the value if it is satisfied. If not, the unchanged builder is returned, effectively bypassing the merge operation entirely. In the predicated version a merge is always performed, but the value that is merged into the builder depends on the predicate. If the predicate is true, this value is the element itself. If it is false, 0 is merged, which has the same effect as not calling the merge operation. Although this clearly performs more work, it does not suffer from branch mispredictions and it can be SIMDized.

4.3.3 Experiment

The flavors of the previous subsection were ran for data with a number of different selectivities, ranging from 0 to 1. The query was slightly altered to increase the execution time of the true branch: the query loops over 6 columns, and if an element of the first column satisfies the predicate, all columns are projected to a single value. The projection performs 8 additions and 3 multiplications.



(a) Single thread (b) Eight threads

Figure 4.3: Total execution time of branching and predicated flavors of a query as a function of selectivity.

Figure 4.3 shows the execution time of these queries on machine 1. The queries were run on a single thread as well as using 8 threads, shown left and right respectively. We can clearly see that for low and high selectivities the branching expression is preferred, while for other values the predicated version performs better. Moreover, the point at which predication becomes better than branching differs depending on the number of

threads used. We can see that predication is better than branching in fewer cases in a multithreaded environment than in a single threaded one.

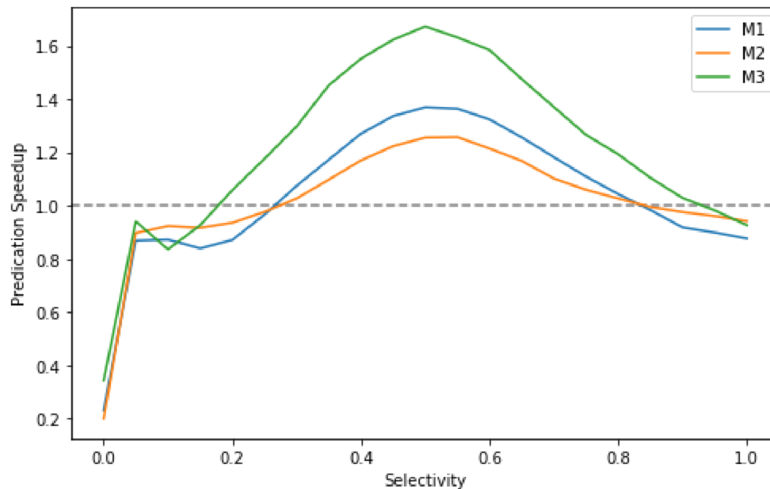


Figure 4.4: Speedup of using predication versus branching depending on selectivity for three different machines.

Figure 4.4 shows the speedup caused by using predication over branching for different machines. We can see that, while the benefit of using predication is potentially higher for machine 1 than for machine 2, the points at which predication is preferred to branching are the same. However, for machine 3 predication is beneficial for a wider range of selectivity values than for machine 1 and 2.

We have seen that the choice of using predicated expressions over branching expression depends on the data, the execution environment as well as the platform itself. This makes it an interesting case for loop-adaptivity.

4.4 Hash joins with Bloom filters

One way of improving the performance a hash join is by using a Bloom filter. A Bloom filter is a data structure that can be used to check if an element is in a set or not. Although false positives are possible, it is typically much cheaper to perform a Bloom filter check than a hash table lookup. Because false negatives are not possible, Bloom filters can be checked before performing the hash table lookup. If the Bloom filter returns false, we know we do not have to perform the more expensive hash table lookup. If the Bloom filter returns true, the hash table still needs to be checked.

It is easy to see that the choice of using a Bloom filter or not for a hash join depends on the selectivity of the join. If the hit ratio of the hash table is low, it can be worthwhile to build a Bloom filter on its keys and check the Bloom filter before testing the hash table. However, if the tested keys are typically in the hash table, building the Bloom filter and testing it only makes the operation slower.

4.4.1 Hash tables and Bloom filters in Weld

A hash table is implemented in Weld by the `dict` data type. To check if a key is in the table, the `keyexists`-expression should be used, which is typically followed by a `lookup`-expression if the key is in the set. Bloom filters on the other hand are not currently implemented in Weld, so we decided to implement our own version. Listing 4.7 shows an example of how this Bloom filter in Weld is used.

Listing 4.7: Creating and using a Bloom filter in Weld

```
1 # Create a new Bloom filter builder
2 let b1 = bloombuilder[i32](256L);
3 # Add 42 to the Bloom filter
4 let b2 = merge(b1, 42);
5 # Materialize the Bloom filter
6 let bf = result(b2);
7 # Check if the Bloom filter contains 42 (returns true)
8 bfcontains(bf, 42)
```

The newly introduced `bloombuilder` is a builder similar to `appender` which can be used to construct bloom filters in parallel. Note that `bloombuilder` takes an integer as argument. This number is the expected number of keys to be stored in the bloom filter, and is used to allocate an appropriate number of bits. Calling result on it creates a value of type `bloomfilter`, which is a read-only bloom filter. Testing the Bloom filter is done using the `bfcontains`-expression.

An alternative way of creating the Bloom filter is to pass a dictionary directly as argument to the `bloombuilder`, in the form of `result(bloombuilder[T](length, dict))`. This initiates a runtime call that directly extracts the hashes from the dictionary, and uses an optimized batch insertion function to quickly build the bloom filter in parallel. Because most of the work is implemented in the runtime, this minimizes compilation times. This follows the philosophy of [29], which states that code should only be JIT-compiled for code that is both performance critical, and specific to the query. Building a Bloom filter from a dictionary is performance critical, but not specific to a query, and it can therefore be pre-compiled.

4.4.2 Flavors in Weld

With the Bloom filter implemented, we can now implement the different flavors for a hash join. In principle, the number of flavors for a hash join grows exponentially in the number of different tables that are joined within a loop. For now, we will look at the simplest example: a join of two tables. Listing 4.8 shows an example of a hash join without Bloom filters, that joins tables R and S on $R.b = S.b$, and selects a, b and c. We assume here that $S.b$ is unique, so we can safely use a `dictmerger` instead of a `groupmerger`. Listing 4.9 shows Weld code for the same join, but now includes a Bloom filter check. Note the additional loop to construct the Bloom filter as well as the `bfcontains`-expression preceding the `keyexists`-expression.

Listing 4.8: Regular hash join of two tables in Weld

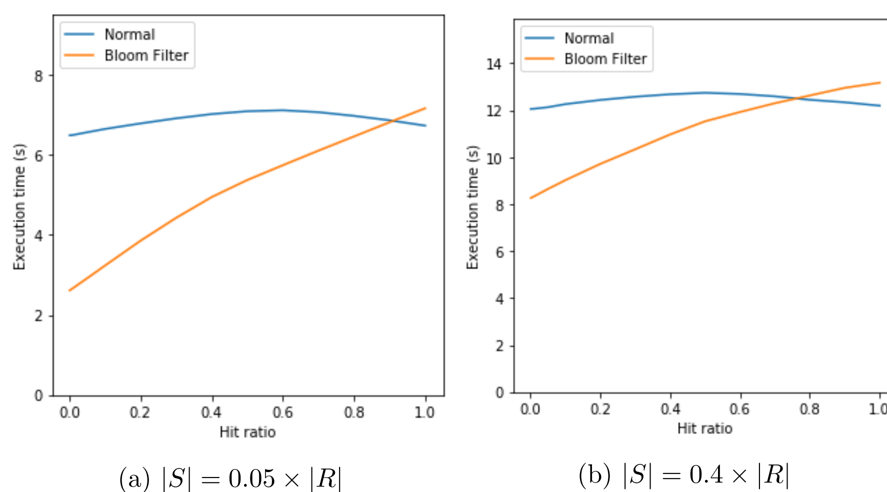
```
1  let S_ht = result(for(
2      zip(S_b, S_c),
3      dictmerger[i32,+],
4      |b,i,e| merge(b,e)
5  ));
6  result(for(
7      zip(R_a, R_b),
8      appender[{i32,i32,i32}],
9      |b,i,e|
10         let a = e.$0;
11         let b = e.$1;
12         if (keyexists(S_ht, b),
13             let c = lookup(S_ht, b);
14             merge(b, {a, b, c}),
15             b
16         )
17 ))
```

Listing 4.9: Hash join of two tables using Bloom filters in Weld

```
1  let S_ht = result(for(
2      zip(S_b, S_c),
3      dictmerger[i32,+],
4      |b,i,e| merge(b, e)
5  ));
6  let S_bf = result(for(
7      S_b,
8      bloombuilder[i32](len(S_b)),
9      |b,i,e| merge(b, e)
10 ));
11 result(for(
12     zip(R_a, R_b),
13     appender[{i32,i32,i32}],
14     |b,i,e|
15         let a = e.$0;
16         let b = e.$1;
17         if (bfcontains(S_bf, b),
18             if (keyexists(S_ht, b),
19                 let c = lookup(S_ht, b);
20                 merge(b, {a, b, c}),
21                 b
22             ),
23             b
24         )
25 ))
```

4.4.3 Experiment

Again, we would like to know if there are cases where one flavor outperforms another, and vice versa. To this end, the query from the previous section was run on synthetic data on which we vary the hit ratio for the join, as well as the size of S relative to R. Figure 4.5 shows the results for machine 1. We can see that the speedups that can be gained by using Bloom filters can be very high, but only for lower hit ratios. As expected, when most of rows in R can join on some row in S, using a Bloom filter is actually detrimental for the overall execution time. Moreover, the size of S has a significant effect on when using a Bloom filter becomes beneficial. When S is relatively small, building a Bloom filter is cheap, and so the investment is worth it in most cases. As S becomes bigger, building a Bloom filter pays off in less cases.



(a) $|S| = 0.05 \times |R|$ (b) $|S| = 0.4 \times |R|$

Figure 4.5: Total execution time of join R and S with and without Bloom filter, for different sizes of S and different selectivity values.

Figure 4.6 shows the speedup of using a Bloom filter for hash joins for each machine, where $|S| = 0.4 \times |R|$. We can see that the selectivity at which it pays off to use a Bloom filter highly depends on the platform on which the query is run.

All in all, we have shown that the choice whether to use Bloom filters or not for a hash join depends on the selectivity of the join, the size of the smaller relation, as well as the platform on which the query is run. This makes it an interesting case for loop-adaptivity.

4.5 Conclusion

In this chapter, we have looked at three different cases that could potentially benefit from loop-adaptivity. Fused selection-projection loops always perform better than any other flavor, and so there is no case for loop-adaptivity here. This likely has to do with the high costs associated with materializing as well as poorly generated SIMD code. The choice of using branching versus predicated expressions were far less obvious, and at the very least depends on predicate selectivity, number of threads as well as hardware. Although

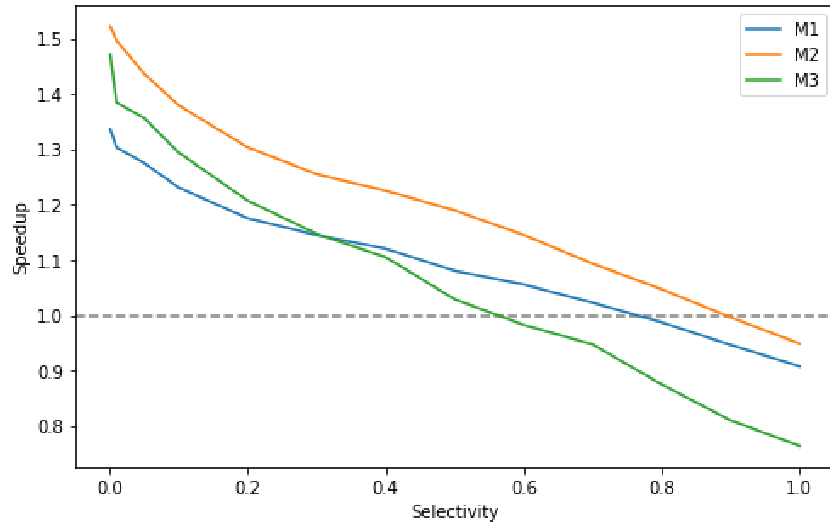


Figure 4.6: Speedup of using Bloom filters for a hash join depending on selectivity for three different machines.

not shown here, it is easy to see that it can also depend on query execution time, e.g. when the selectivity in the first half of the data is significantly different than in the latter. Finally, whether it pays off to use a Bloom filter for hash joins is also hard to predict: using hash joins makes sense for lower join selectivity values, but the exact cross over point depends on factors such as the relative size of the smaller relation as well as the platform on which the query is run. All in all, we have found some cases motivating the integration of loop-adaptivity in Weld.

Chapter 5

Loop-Adaptivity Implementation

The main objective of this research is implementing loop-adaptivity in Weld, which we will discuss in this chapter. Section 5.1 will give a high level overview of the different components that contribute to the implementation of loop-adaptivity in Weld. In the following sections we will explore each component in more detail. Section 5.2 and section 5.3 explain how flavors can be defined using Weld IR and how they are generated by the optimizer respectively. Section 5.4 will briefly discuss the code generated by the Weld compiler to support loop-adaptivity, while section 5.5 discusses the changes that were made to the runtime. Finally, section 5.6 presents the algorithm used to explore and exploit flavors, which is based on vw-greedy.

5.1 Overview

To make loop-adaptivity possible, a form of batched execution needs to be present, in which computations are done on a few thousand tuples at a time. Unlike Vectorwise, Weld does not have the notion of primitives that perform a simple computation on vectors at a time. Instead, it relies on the `for`-expression, which perform a series of computations on batches of data in parallel. Therefore, we introduce adaptivity in Weld by adapting entire loops.

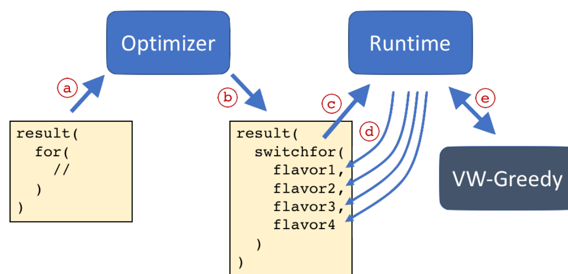


Figure 5.1: Overview of loop-adaptivity in Weld.

Figure 5.1 shows a general overview of the loop-adaptivity architecture in Weld. Making loops adaptive is done in Weld IR by means of the `switchfor`-expression, which takes

an arbitrary number of parallel loops as argument that we will refer to as flavors. The flavors are typically generated from normal IR (a) by optimization passes that recognize opportunities for adaptivity (b), but could in principle also be written by hand. A `switchfor`-expression calls the runtime (c) to spawn a new task, much in the same way that a `for`-expression would. It differs from a `for`-expression in that a `switchfor` task does not have a single function pointer, but one for each flavor defined in the `switchfor`. It is up to the runtime to actually explore and exploit the most promising flavors (d), using an algorithm based on vw-greedy (e).

Because the number of flavors in a `switchfor` grows exponentially in the number of adaptive choices, one of the major challenges is to keep code explosion (and thus compilation times) at a minimum. This is done by (optionally) compiling only the bare minimum flavors up front, and compiling other flavors during runtime as needed. Other mechanisms that are introduced are lazy evaluation of `let`-expressions as well as instrumented loops to measure specific features of the data.

5.2 Internal representation

The central idea of introducing loop-adaptivity in Weld is to run different flavors of a loop. These loops are defined in the newly introduced `switchfor`-expression. Listing 5.1 shows an example of a `switchfor`-expression that counts the number of 42s in a list. The first flavor does this using the branching `if`-expression, while the second uses the predicated `select`-expression.

Listing 5.1: Counting 42s with adaptive branching or predication using the switchfor-expression.

```
1  let sw_bld = merger[i32,+];
2  result(switchfor(
3      |lb1,ub1|
4          for(xs, sw_bld, |b,i,x|
5              if (x == 42, merge(b, 1), b)
6          ),
7      |lb2,ub2|
8          for(xs, sw_bld, |b,i,x|
9              merge(b, select(x == 42, 1, 0))
10         )
11 ))
```

The `switchfor`-expression returns the same result as any of its flavors would if it were executed in isolation, provided that all flavors are equivalent, with some exceptions (which we will go into later in this section). A flavor can contain multiple `for`-loops, allowing `switchfor` to adapt between entire pipelines. Note that each flavor is a lambda function that takes two parameters as argument. These parameters hold the value for the lower and upper bound of the current batch. Further, each `for`-loop has the same builder as argument, which is the builder that is ultimately returned by `switchfor`. This is a general requirement of `switchfor` to ensure consistent results, and makes sure we do not have to merge the results of the different flavors at the end of execution.

Under the hood, `for`-loops contained by `switchfor`-expressions behave differently from

vanilla `for`-expressions. As an example, see listing 5.2, which shows a `switchfor` containing a flavor that computes the sum of two dimensional list of integers. If the `for`-expression were executed as a regular `for`-loop, a task would be created for the outer loop and split up in subtasks that are scheduled among the workers. When a subtask for the outer loop is then executed, it will encounter the inner loop, which invokes another runtime call. During this call, the current task for the outer loop is canceled, and new tasks are created for the inner loop and the remaining iterations of the outer loop (in that order). Because the outer task then finishes executing after only a single iteration, we cannot use it to measure the performance of the flavor, as all that we would be measuring then is the time required to create new tasks rather than the actual computation. Therefore, any `for`-expression contained by `switchfor` will not call the runtime, and are thus executed completely sequentially. Parallelism is introduced by calling the `for`-loops themselves in parallel, rather than the loop body. While this eases the implementation of loop-adaptivity, the downside of this approach is that it may be a source of load imbalance. If all vectors in lists are small except for one, one of the subtasks would be doing most of the work. `switchfor`-expressions are therefore currently not suitable for these kind of queries.

Listing 5.2: Example of a nested for loop in a switchfor-expression.

```
1  switchfor(
2      |lb,ub|
3          for(lists, merger[i32,+], |b,i,xs|
4                  for(list, b, |b,i,x|
5                          merge(b, x)
6                  )
7          ),
8      ...
9  )
```

The fact that entire pipelines are parallelized, rather than just the function body, also has an important implication for loops that take the result of another loop as argument. Listing 5.3 shows an example of this, where the first loop projects a column y and then selects its values based on corresponding values in column x. In regular Weld, the projection loop is executed in its entirety, materialized, and then the selection loop is evaluated. In the example however, the projection loop is only run on a partition of input data at a time. This means that the `result`-expression will not return a vector as long as y, but only as long as the partition of y. The second loop needs to account for this by only iterating over the corresponding partition of x. This can be done using the lower and upper bound parameters that are passed to the flavor function. Finally, because the flavor function is called multiple times, any `result`-expression in the function is also called multiple times. Weld states that `result` may only be called once on a builder, and therefore only new builders can be used as argument of a `result`-expression in a `switchfor`.

Listing 5.3: Example of pipelined for loops in a switchfor-expression.

```
switchfor(
    |lb,ub|
        for(
            zip(
                iter(xs, lb, ub, 1L),
                result(for(ys, appender[i32], |b,i,y|
                    merge(b, y*y)
                ))
            ),
            sw_bld,
            |b,i,e|
                if (e.$0 == 42, merge(b, e.$1), b)
        ),
    ...
)
```

5.2.1 Instrumentation and deferred let-expressions

Deciding to use a flavor is cheap when a flavor can be completely defined in a single `for`-loop, such as the flavors for the branching versus predication case. In the case of Bloom filters it becomes more complicated: if selectivity is low, we would like to use a Bloom filter, but that means that one should have been built in the first place. However, building a Bloom filter when selectivity is too high could be detrimental to overall performance. We therefore need a system that defers evaluating some expression until we actually decide (through instrumentation) that it is required.

A naive way of implementing this would be to split the query in several parts: first an instrumented loop runs over a sample of the data, and measures the hit rate of the dictionary. If the hit ratio is low enough, a Bloom filter is built and a `switchfor`-expression is evaluated with flavors for a regular join and a join using Bloom filters. If the hit ratio is too high, a regular join is performed using a normal `for`-expression. The problem with this approach is the amount of additional code that this requires: instead of a single `switchfor`-expression and a loop for the Bloom filter creation, we now have an additional `for`-loop for instrumentation, a `for`-loop for the regular join, as well as an `if`-expression that steers control flow depending on the result of instrumentation. The amount of code gets exponentially bigger as more dictionaries are added to the same loop, which could drive compilation times to unacceptable levels.

We therefore took a different approach, using Weld's annotation system. `let`-expressions may now be annotated with a `deferred_until` annotation that specifies that the right hand side of the expression should not be evaluated until some condition is true. The condition can only contain global variables (more on the use of global variables in section 5.2.1). Instrumentation is performed using the `switch_instrumented` and `count_calls` annotations. `switch_instrumented` denotes that a flavor is instrumented, which tells the runtime to call that flavor before running any others, and to run it only once so that instrumentation costs are amortized. It also informs the runtime which global variables that hold the instrumentation data are updated. `count_calls` can be annotated on any

expression, and simply increments a global variable by one each time the expression is evaluated.

Listing 5.4: Example of an instrumented flavor, a deferred let expression, as well as a flavor that depends on the deferred let. Parts of the code were left out for brevity.

```
1  @(deferred_until: @ht_hit == 0 || @ht_try / @ht_hit < 0.5)
2  let bf = result(bloombuilder[i32](len(ht), ht));
3  switchfor(
4      @(switch_instrumented:@ht_try,@ht_hit)
5      |lb,ub|
6          for(xs, sw_bld, |b,i,e|
7              @(count_calls:@ht_try)
8              if (keyexists(ht, e),
9                  @(count_calls:@ht_hit)
10                 merge(b, {e, lookup(ht, e)}),
11                 b
12             )
13         ),
14     ..., # Regular flavor
15     @(switch_if_initialized: bf)
16     |lb,ub|
17         for(xs, sw_bld, |b,i,e|
18             if (bfcontains(bf, e),
19                 # Join
20                 ...
21             )
22         )
23 )
```

Listing 5.4 shows an excerpt of an adaptive hash join using instrumentation and a deferred `let`-expression. The right hand side of line 1, the creation of the Bloom filter, is not evaluated immediately. Instead, it is ignored and `switchfor` is evaluated. The runtime first runs the instrumented loop to measure the hit ratio of ht. When it is done, it will check any deferred `let`-expressions that depend on the global variables that are given as arguments to switch_instrumented, i.e. bf. If the condition is true, control flow is passed to the creation of the Bloom filter, after which the `switchfor`-expression continues execution on the remaining data, and all flavors (except the instrumented) can be explored. If the condition is false, `switchfor` continues executing, but will ignore any flavors that depend on the deferred `let`-expression. A flavor can be annotated using switch_if_initialized to inform the runtime on which values it depends.

Global variables

To support instrumentation we also introduced global variables, which are preceded by the `@` symbol. The global variables allow us to count the number of times an expression was evaluated, without having to resort to more heavyweight mergers. They are only allowed in the annotation system, because introducing them in the language itself would break the static single assignment (SSA) property of Weld. Global variables can be defined using the run_vars annotation, which is only allowed at the beginning of the Weld program.

5.3 Flavor generation

With the new expressions for defining loop-adaptivity in place, we can now generate the actual flavors. This is implemented by new Weld IR optimization passes that detect opportunities for adaptivity, and generate corresponding `switchfor`-expressions. In this section we will give an overview of the three implemented passes. Note that, due to time constraints, the passes are exclusive: any `for`-expression already contained by a `switchfor`-expression will not be further adapted. However, users can specify in which order the adaptive optimizations should be performed.

5.3.1 Selection-projection loop reordering

Although we have not found any benefits of full computation versus a fused selection-projection loop (see section 4.2), we did implement an optimization pass that generates a `switchfor`-expression with both flavors from a fused loop. This was done partly because it still allows us to verify if the system can find the fastest running flavor, but also because the optimization may actually be beneficial if the cost of materialization is reduced in future updates of Weld.

Listing 5.5 shows the general shape of `for`-loops that are matched by the optimization pass. Listing 5.6 shows the resulting `switchfor`-expression. The first flavor is a copy of the original fused loop, and remains unchanged. The second flavor first executes a loop, which is the input of the outer loop. Note the use of the lower and upper bound parameters on line 12, for reasons as discussed in the previous section.

Listing 5.5: Shape of a for loop that is eligible for adaptive selection-projection reordering.

```
1  for(data, bld, |b,i,e|
2      if (condition(e),
3          merge(b, project(e)),
4          b
5      )
6  )
```

Listing 5.6: Shape of the switchfor expression resulting from the adaptive selection-projection reordering optimization.

```
switchfor(
    |lb1,ub1|
        for(data, bld, |b,i,e|
            if (condition(e),
                merge(b, project(e)),
                b
            )
        ),
    |lb2,ub2|
        for(
            zip(
                iter(data, lb2, ub2, 1L),
                result(for(data, appender[?], |b,i,e|
                    merge(b, project(e))
                ))
            ),
            bld,
            |b,i,e|
                if (condition(e.$0),
                    merge(b, e.$1),
                    b
                )
        )
)
```

5.3.2 Branching versus predication

The second adaptive optimization pass is adaptive predication. The pass recognizes `if`-expressions that could be transformed into `select`-expressions, and annotates them with `@(predicate:true)`. The regular predication pass will then transform them into actual `select`-expressions. Listing 5.7 shows the general shape of expressions that are matched by the pass, while listing 5.8 shows the result. Note that whatever the number of `if`-expressions in the original loop is, there will always only be two resulting flavors. The reason is that `if`-expressions are quite numerous in the TPC-H queries that we tested, mostly for null checks, and so compilation times became too high when we allowed a flavor for every combination. A way around this would be to annotate which `if`-expressions in the generated code should not be adapted (for instance, null-checks) to reduce the number of options. However, because of limited access to the code generator this was not feasible.

Listing 5.7: Shape of a for loop that is eligible for adaptive predication.

```
for(data, bld, |b,i,e|
    ...
    if (c1, t1, f1)
    ...
    if (cn, tn, fn)
    ...
)
```

Listing 5.8: Shape of a switchfor expression resulting from the adaptive predication pass.

```
switchfor(
    |lb1,ub1|
        for(data, bld, |b,i,e|
            ...
            if (c1, t1, f1)
            ...
            if (cn, tn, fn)
            ...
        ),
    |lb2,ub2|
        for(data, bld, |b,i,e|
            ...
            @(predicate:true)
            if (c1, t1, f1)
            ...
            @(predicate:true)
            if (cn, tn, fn)
            ...
        ),
)
```

5.3.3 Adaptive Bloom filters

The last adaptive optimization pass is the adaptive Bloom filters pass. It works in two phases. The first phase matches `for`-expressions that contain `keyexist`-expressions, and finds the set of dictionaries that are probed in that `for`-loop (including nested loops). In the second phase, a Bloom filter expression is created for each dictionary, and `for`-expressions are made for each possible flavor, as well as an instrumented `for`-expression. These `for`-expressions are encapsulated by a `switchfor`-expression, which is preceded by deferred `let`-expressions that build the Bloom filters. Listing 5.9 shows the general shape of loops that are matched by the optimizer, while listing 5.10 shows the final Weld IR. In the latter, $N = 2^n$.

Listing 5.9: Shape of a for expression eligible for adaptive Bloom filters.

```
for (data, bld, |b,i,e|
        ...
        if (keyexists(d1, k1),
        ...
        if (keyexists(dn, kn),
        ...
)
```

Listing 5.10: Shape of a switchfor expression resulting from the adaptive Bloom filters pass.

```
@(deferred_until: @d1_try > 0 && @d1_hit / @d1_try > 0.2)
let d1_bf = result(bloombuilder[?](len(d1), d1));
...
@(deferred_until: @dn_try > 0 && @dn_hit / @dn_try > 0.2)
let dn_bf = result(bloombuilder[?](len(dn), dn));
```

```
6  switchfor(
7      @(switch_instrumented: @d1_try, @d1_hit, ..., @dn_hit, @dn_try)
8      |lb0,ub0|
9          for (data, bld, |b,i,e|
10             ...
11             @(count_calls: @d1_try)
12             if (keyexists(d1, k1), @(count_calls: @d1_hit) ...
13             ...
14             @(count_calls: @dn_try)
15             if (keyexists(dn, kn), @(count_calls: @dn_hit) ...
16             ...
17         ),
18     |lb1,ub1|
19         for (data, bld, |b,i,e|
20             ...
21             if (keyexists(d1, k1),
22             ...
23             if (keyexists(dn, kn),
24             ...
25         ),
26     @(switch_if_initialized: d1_bf)
27     |lb2,ub2|
28         for (data, bld, |b,i,e|
29             ...
30             if (bfcontains(d1_bf, k1) && keyexists(d1, k1),
31             ...
32             if (keyexists(dn, kn),
33             ...
34         ),
35     ...,
36     @(switch_if_initialized: d1_bf, ..., dn_bf)
37     |lbN,ubN|
38         for (data, bld, b,i,e|
39             ...
40             if (bfcontains(d1_bf, k1) && keyexists(d1, k1),
41             ...
42             if (bfcontains(dn_bf, k1) && keyexists(dn, kn),
43             ...
44         )
45 )
```

5.4 Code generation

After the adaptive optimization passes have produced adaptive Weld code, LLVM code needs to be generated for it. In this section we will give an overview of the code that is generated for `switchfor`, deferred `let`-expressions and instrumentation and global variables.

5.4.1 Switchfor

The code generated for a `switchfor`-expression is similar to that of a `for`-expression. A wrapper function is generated that computes the size of the input vectors and performs

bounds checking. If it passes the check, information about every flavor is collected in runtime structures that are then passed to a final runtime function call that creates tasks for the `switchfor`. Further, it generates callbacks for each flavor that are used by the runtime to execute them. Unlike the `for`-expression, this callback calls the function for the first `for`-expression defined in a flavor, rather than the function for its loop body. Because the `switchfor` wrapper already checks the size of the input vectors, the wrapper function of each `for`-loop inside a `switchfor` does not perform this check (unlike regular `for`-loops). Finally, a callback function is generated for the continuation of the `switchfor`.

5.4.2 Deferred let

When the compiler encounters deferred `let`-expressions, it generates a *build* function for the right hand side of the `let`-expression as well as a *condition* function that evaluates the condition. These functions are meant to be called by the runtime. The build function terminates with a runtime call (`weld_rt_set_deferred`) that sets the result of the deferred expression. At the point in the program where normally the assignment would be performed, a runtime function call (`weld_rt_register_deferred`) is generated that registers the deferred `let` so that the runtime is aware of its existence. The left hand side is set to null, and otherwise treated as if it were a regular value in the rest of the code. During SIR generation the control flow graph (CFG) is traversed to find the first usage of the deferred value. Just before the value is used, another runtime call (`weld_rt_get_deferred`) is inserted that overwrites the null value with the actual value, provided the runtime has decided to call the build function.

5.4.3 Instrumentation and global variables

Global variables are implemented using LLVM's global variables, and are initialized at the start of the `run` function. Whenever a `count_calls`-annotation is encountered, the value of the associated global variable is loaded, incremented by one, and written back again. This is not optimal because these counters are not kept in registers as long as strictly possible, but is sufficient because the instrumentation overhead is still negligible compared to the overall execution time.

5.4.4 Lazy function compilation

To decrease compilation times, not all functions are compiled up front. Recall that before generating LLVM IR, the Weld IR is first compiled into SIR. During SIR generation, functions can be marked as *lazy*: while SIR is generated for it, the LLVM generator will ignore it. The execution engine will keep an handle to the SIR code during execution which it can pass to the compiler if it decided to run a flavor which requires a function to be compiled. For now, only flavors that depend on a deferred `let`-expression are marked as lazy, because the the outcome of instrumentation makes the choice of whether compiling the function is worth it relatively easy. The build function of a deferred `let`-expression could in principle also be a candidate for lazy compilation. However, in our case deferred `let`-expressions are only used for building Bloom filters. This is handled by a pre-compiled runtime function, and therefore the cost of compilation is very low.

5.5 Runtime

While the IR and code generation is responsible for the definition and implementation of flavors, the runtime is responsible for exploring and exploiting the flavors. In this section we will look at the most important changes that were made to the runtime to integrate loop-adaptivity in Weld.

5.5.1 Switchfor

Adaptive execution is kicked off when `weld_rt_start_switch` is called. This function is similar to `weld_rt_start_loop`, but takes multiple flavors as arguments as well as metadata about each flavor, such as whether a flavor is instrumented or not. If an instrumented flavor exists, a separate *instrumented task* is created for it and pushed on the queue. The instrumented task will be configured so that it will only run the maximal number of iterations that fit in a single subtask (referred to as the *grain size*). Another *body task* is created that has pointers to all the remaining flavors, as well as a *continuation task* for when `switchfor` has finished executing. If an instrumented task was created the body task is not pushed on the queue, but set as the continuation of the instrumented task to ensure that the instrumented task is completed entirely before the remaining flavors are explored. For the remainder of this section, we will assume an instrumented task was created.

Instrumentation and lazy compilation

When the task that invoked the `weld_rt_start_switch` call finishes executing, the instrumented task is popped off the queue and executed by a worker thread. It is run like any other task, but upon its completion the execution engine will inspect the deferred `let`-expressions that are associated with the global variables that were updated by the instrumented task. For each of those deferred `lets`, its condition function is evaluated. If it returns true, a new *build task* is created with a pointer to the build function, and pushed on the queue. The build task is similar to a parallel loop task, and can thus be split up and parallelized. The execution engine then proceeds to check if there are any flavors that should be compiled. If so, a *compilation task* is created that compiles the relevant flavors. The compilation task is pushed first to ensure that a worker will execute it as early as possible. The build tasks will be executed by the remaining workers, and are enforced to finish them before executing the body task. If this were not the case we would risk the scenario where the majority of workers are already executing the `switchfor`, but cannot execute the most promising flavors because a single worker is still busy evaluating the deferred `let`. By moving the build tasks first, the total runtime should decrease (provided that the flavors depending on the deferred expressions are faster). The body task will not wait for completion of the compilation task. This task cannot be parallelized, so time would be wasted by waiting for its result.

Executing the body task

When the build tasks have finished executing, the body task is pushed to the queue and split in subtasks that can be stolen by other workers. Before execution of each subtask the worker has to find out which flavor it is supposed to run, which is stored in a central data structure. After executing the task it will update statistics about the flavor, and periodically choose a new flavor to explore or exploit. We will examine the adaptive algorithm in more detail in section 5.6.

5.5.2 Deferred let-expressions

Deferred `let`-expressions are implemented in the runtime by the `weld_rt_defer_build`, `weld_rt_set_deferred_result` and `weld_rt_get_deferred_result` functions. The first function is responsible for registering a deferred expression, and keeps track of the build and condition functions, the global variables that are evaluated in the condition function and the result of the build function (if any). `set_deferred_result` is called by the build function, and stores its result, while `get_deferred_result` is called by functions that depend on a deferred `let`-expression. Checking if the build function should be called, as well as building it, is the responsibility of an instrumented task, as described in section 5.5.1.

5.6 VW-greedy

The key component of a micro-adaptivity (or loop-adaptivity) system is the adaptive optimizer, which selects the flavors to run. In Vectorwise, this is implemented using the vw-greedy algorithm, which alternates between exploration and exploitation phases. In the exploration phase a random flavor is tested, while during the exploitation phase the flavor that performed best thus far is run. Only recent performance statistics are kept to ensure that the statistic reflect the current data, and are not influenced by factors that may not be true anymore (such as the selectivity of a predicate).

To implement loop-adaptivity in Weld, we use the vw-greedy algorithm, as it has proven its performance in Vectorwise. However, [40] does not specify how the algorithm works in a parallel setting. Particularly, it is not specified what happens if the explore period is shorter than the number of worker threads. In this case, if all threads run the flavor that is to be explored at the start of the phase, we will end up having executed too many instances of an exploration flavor, possibly resulting in suboptimal performance. To solve this problem only one worker (the *explorer*) will be responsible for exploring new threads, while all other workers (the *exploiters*) are restricted to exploiting the best flavor. The exploring thread is not fixed, but is set to be the thread who finishes the last task in an explore period. This is done to ensure that the thread who is chosen to explore is actually available, which may not be the case for all threads (e.g. when a worker is still compiling a flavor in the background).

Listing 5.11 shows the adaptive algorithm used in Weld. Before a task is run the worker checks if it is the explorer, and depending on the result it runs the appropriate flavor. After the task has finished executing `updateFlavor` is called, which updates flavor

statistics. If the thread was an explorer, and just finished executing `EXPLORE_LENGTH` tasks, it becomes an exploiter, and the average cost of the explorer flavor is updated. If the thread was an exploiter, and the exploiters finished executing `EXPLOIT_PERIOD` tasks since the phase began, the average cost of the exploited task is updated, and the best flavor is reevaluated. Finally, if the total number of calls since the last explore phase exceeds `EXPLORE_PERIOD`, the current thread becomes the explorer, and the least currently used flavor will be explored. Note that this is different from vw-greedy, which picks a random flavor for exploration. Using the least recently used flavor minimizes the time before all flavors are explored.

Another difference with vw-greedy is that we do not explore all flavors before exploiting, for cases of code explosion where there are too many flavors to explore. However, because we explore the least recently used flavor rather than a random one, the order in which flavors are explored for the first time is deterministic. This can be exploited by ordering flavors in such a way that the most promising flavors will be explored first. Other than that, this algorithm boils down to vw-greedy if it is executed by a single thread. In that case, exploration and exploitation phases effectively alternate. In multithreaded settings however, exploration phases overlap with exploitation phases.

Listing 5.11: Loop-adaptive algorithm in Weld in pseudo code, accounting for multithreading.

```
1  def runTask(task):
2      // Check whether the current thread is an explorer. If so,
3      // run the explore flavor. Otherwise run best flavor.
4      imExplorer = (myId == task.exploreThread)
5      if imExplorer:
6          task.localFlavor = task.exploreFlavor
7      else:
8          task.localFlavor = task.exploitFlavor
9
10     // Run flavor and measure time
11     start = getCycles()
12     task.localFlavor.run()
13     end = getCycles()
14
15     // Update statistics
16     updateFlavor(task, imExplorer, end - start)
17
18 def updateFlavor(task, imExplorer, cycles):
19     // Increase global counter
20     task.calls++
21
22     // Update statistics on the flavor we just executed
23     myFlavor = task.localFlavor
24     myFlavor.totCycles += cycles
25     myFlavor.totTuples += (task.upper - task.lower)
26
27     // Update average cost and stop exploring when explore length is
28     // over
29     if imExplorer and --task.exploreRemaining == 0:
30         myFlavor.avgCost = (myFlavor.totCycles - myFlavor.prevCycles) /
31                            (myFlavor.totTuples - myFlavor.prevTuples)
```

```
32          task.exploreThread = -1
33
34      // Update average cost and find best flavor when exploit period is
35      // over
36      if !imExplorer and --task.exploitRemaining == 0:
37          myFlavor.avgCost = (myFlavor.totCycles - myFlavor.prevCycles) /
38                             (myFlavor.totTuples - myFlavor.prevTuples)
39          task.exploitFlavor = getBestFlavor(task)
40          initFlavor(task.exploitFlavor, task.calls)
41          task.exploitRemaining = EXPLOIT_PERIOD
42
43      // Check whether it is time to explore again. If so, the current
44      // thread will be the explorer for EXPLORE_LENGTH calls. The least
45      // recently used flavor will be explored.
46      if task.calls > task.exploreStart:
47          task.exploreFlavor = getLruFlavor(task)
48          initFlavor(task.exploreFlavor, task.calls)
49          task.exploreThread = myId
50          task.exploreRemaining = EXPLORE_LENGTH
51          task.exploreStart += EXPLORE_PERIOD
52
53  def initFlavor(flavor, currentCall):
54      flavor.lastChosen = currentCall
55      flavor.prevCycles = flavor.totCycles
56      flavor.prevTuples = flavor.totTuples
```

Chapter 6

Results and Discussion

To evaluate the performance of the loop-adaptivity system, we ran a variety of micro-benchmarks as well as the TPC-H benchmark suite [46]. In most cases we measure both execution times as well as compilation times, and see at which point (if any) the increased performance of loop-adaptivity makes up for the increased compilation times. Section 6.1 covers the setup that was used to perform the experiments. Sections 6.2, 6.3 and 6.4 discuss micro-benchmarks that specifically test how well the three adaptive optimization passes perform. In section 6.5 we look at the performance of various methods of creating Bloom filters. In section 6.6 we examine the overhead of instrumentation, while section 6.7 covers the results of the TPC-H experiments.

6.1 Experimental setup

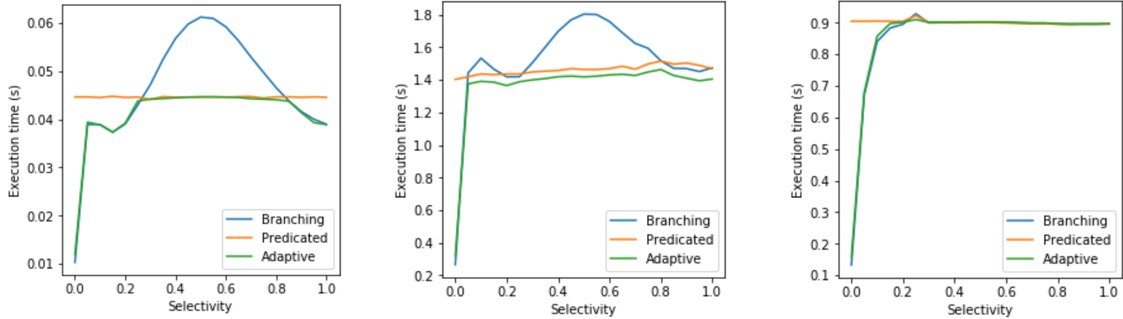
Table 6.1 shows basic information about the machines that were used for the experiments. In general, each experiment was repeated 5 times, and for each experiment both the compilation time and execution time was recorded. The queries were run on data with varying scale factors. The TPC-H queries were run on M1 only, while all machines were used for the micro-benchmarks.

Table 6.1: Test machine details

Name	**CPU**	**Clock Speed**	**#Threads**	**RAM**
M1	Intel Xeon E5-2650	2.0 Ghz	16	256 GB
M2	Intel Core i7-2600K	3.4 Ghz	8	16 GB
M4	Intel Xeon E5-4657L v2	2.4 Ghz	24	1024 GB

6.2 Adaptive predication

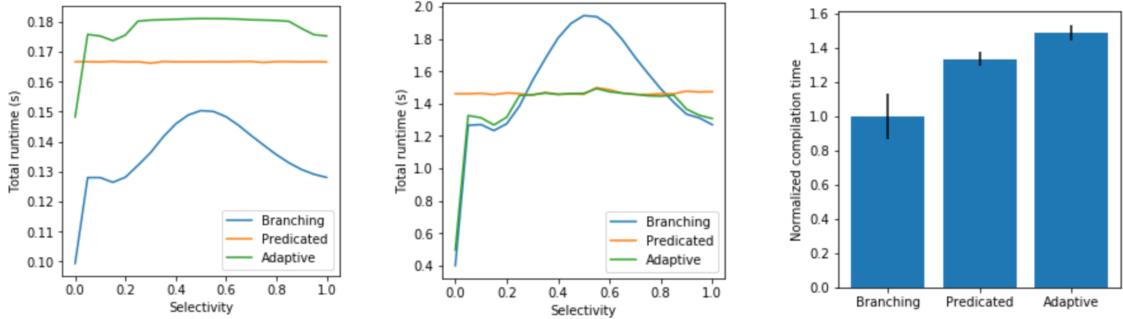
The first micro-benchmark aims to evaluate the performance of code produced by the the adaptive predication optimization pass. We used the same query and data that were discussed in section 4.3, but now also run the adaptive algorithm.



(a) M1, scale factor 1, single threaded.

(b) M4, scale factor 30, single threaded.

(c) M2, scale factor 30, eight threads.



(d) M1, scale factor 1, single threaded.

(e) M1, scale factor 30, single threaded.

(f) Normalized compilation times.

Figure 6.1: Results of the adaptive predication benchmark. The top row shows execution time depending on selectivity for the branching and predicated flavors as well as adaptive Weld for different machines, scale factors and numbers of threads. The bottom row shows total runtimes including compilation times, as well as the isolated compilation times.

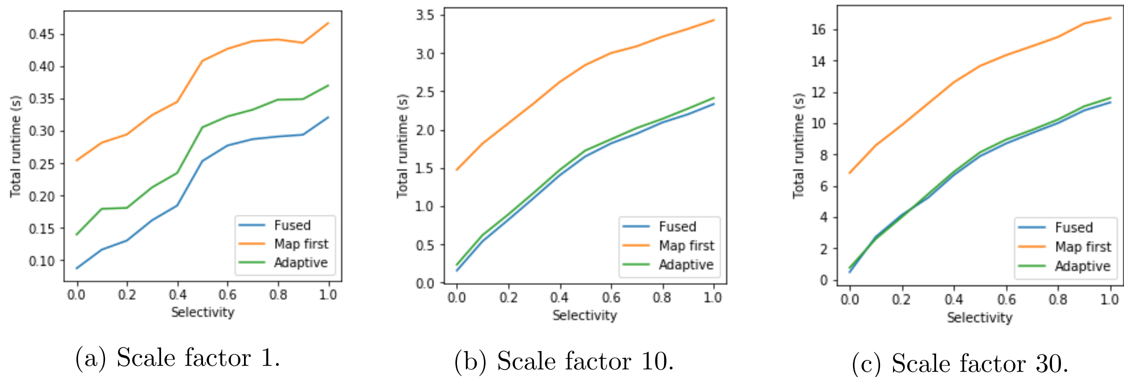
The top row of figure 6.1 displays the execution time for each variant for varying machines, scale factors and numbers of threads. We can see that the adaptive variant accurately tracks the fasted flavor, even though the best flavor depends on multiple factors. In figure 6.1b we can even see that adaptive Weld is faster than any other variant. This might have to do with the fact that `switchfor` tasks use a smaller grain size than `for` tasks, which could lower the number of cache misses.

However, execution time is only part of the full picture. Figure 6.1d shows the total runtimes (including compilation times) of the query on machine 1, for a scale factor of 1. Although we have just seen that the adaptive variant always matches the fastest flavor in terms of raw execution speed, its benefit is lost because of the increased compilation time, which dominates the total runtime. Predication is also never faster than branching in this case, because the predicated version is SIMDized by Weld. This requires significantly more LLVM code and thus takes longer to compile. Figure 6.1e shows results of the same query, on the same machine, but for a dataset that is 30 times larger. Because the total runtime is now dominated by execution time, we can see that at any time the adaptive version

is only slightly slower than the faster flavor. Finally, figure 6.1f shows the normalized compilation time of each variant. The adaptive variant takes roughly 50% more time to compile than branching only. All in all this suggests that adaptive predication works well, provided that the execution time dominates the total runtime.

6.3 Adaptive selection projection order

The second micro-benchmark evaluates adaptive full computation. As we have seen in section 4.2, full computation in Weld never outperforms a fused loop, so all we can hope for here is that loop-adaptivity recognizes this and chooses the fused variant with minimal overhead. We used the same query and data as described in section 4.2 for this experiment.



(a) Scale factor 1. (b) Scale factor 10. (c) Scale factor 30.

Figure 6.2: Total runtime depending on selectivity for the fused and full computation flavors as well as adaptive Weld on machine 4, for varying scale factors.

Figure 6.2 shows the total runtime of the query, depending on selectivity and for different scale factors. Again, we see that adaptivity performs better for higher scale factors, as compilation times get amortized. Although the adaptive code takes about 8% longer to compile than the map first strategy, it outperforms it even for the smallest scale factor. Across all machines and scale factors, the slowdown of the adaptive variant compared to the fused variant ranged from 0.77% in the best case to 40.17% in the worst case. Speedups compared to the full computation variant ranged from 2.08% to 213.51%.

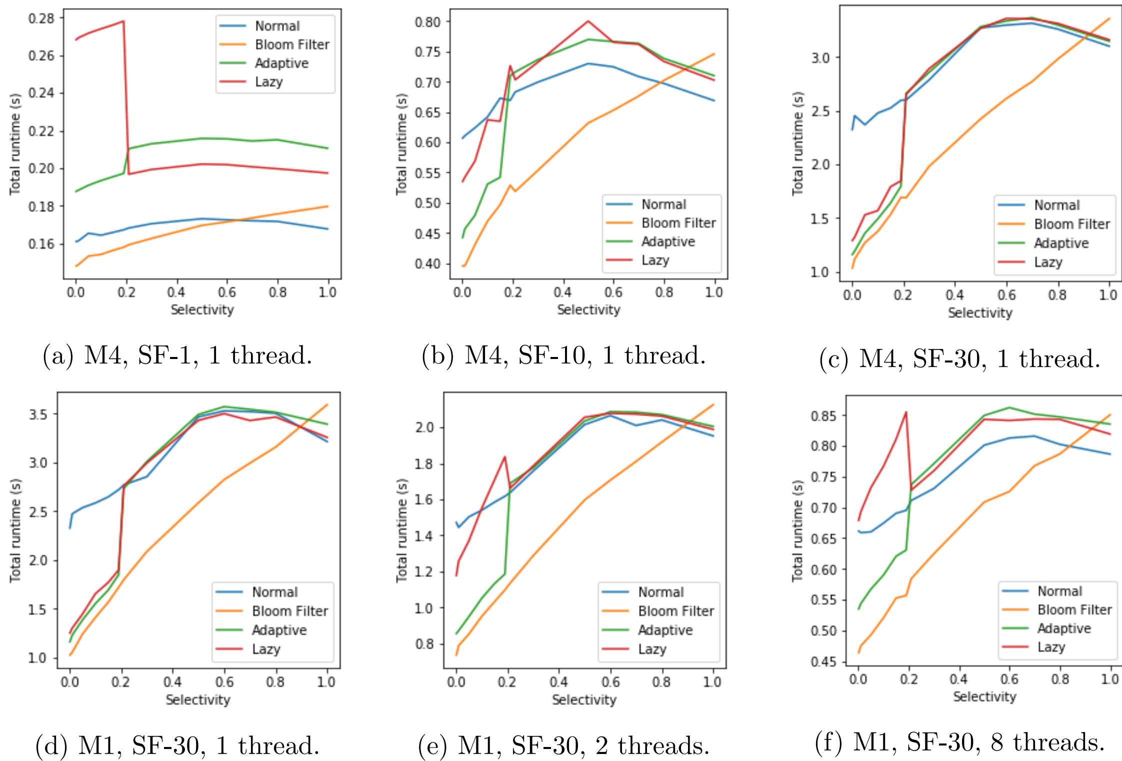
6.4 Adaptive Bloom filters

To evaluate the performance of the adaptive Bloom filters optimization, we ran a number of benchmarks that use different join queries: a simple `dictmerger` join of two tables, a three-way join using `dictmergers` and a three-way join using `groupmergers`.

6.4.1 Simple join

The first benchmark is a simple join of the form $R \bowtie S$, which we have also seen in section 4.4. Figures 6.3a, 6.3b and 6.2c show the total runtime of this query for varying scale

factors. Again, for the lowest scale factor loop-adaptivity never pays off. Lazy function compilation shows a modest improvement when the join selectivity is above 20%, which we set as the threshold for building Bloom filters. However, below that point lazy compilation is actually worse: it seems that lazy compilation is relatively expensive, and makes more sense when the number of flavors is much higher than the expected number of flavors to be explored. Loop-adaptivity performs better for higher scale factors, but eager compilation outperforms lazy compilation in all cases. Also, the threshold for building a Bloom filter is rather pessimistic. This is an unavoidable characteristic of using heuristics, but can surely be improved by using better cost models. This is outside of the scope for this research, and left for future work.



(a) M4, SF-1, 1 thread.

(b) M4, SF-10, 1 thread.

(c) M4, SF-30, 1 thread.

(d) M1, SF-30, 1 thread.

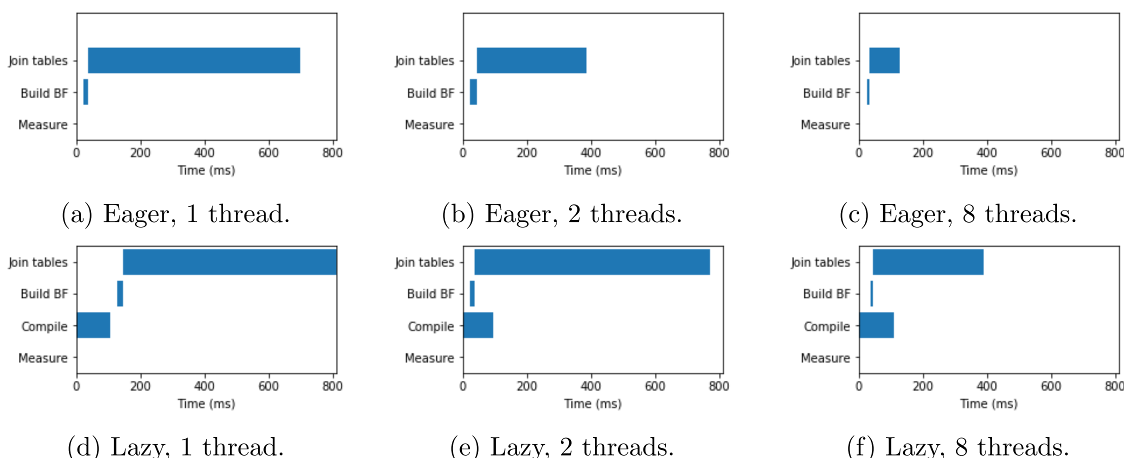
(e) M1, SF-30, 2 threads.

(f) M1, SF-30, 8 threads.

Figure 6.3: Results of the simple join micro-benchmark, showing total query runtimes depending on selectivity for the regular hash join, Bloom filter hash join, adaptive Weld and finally lazily compiling adaptive Weld. The top row shows the singlethreaded runtimes on machine 4, for varying scale factors. The bottom row shows runtimes on machine 1, scale factor 30, and varying numbers of threads.

Figures 6.3d, 6.3e and 6.3f show how the total runtime of the query varies for different numbers of threads. All variants scale relatively well, except lazy function compilation, making it worse than regular hash joins for even the lowest selectivity values. This contradicts our expectation that lazy compilation would especially benefit from multithreading, as the compilation can be performed in the background while build tasks are being executed.

To get a better of understanding what is going on, we profiled the tasks that are executed by both eagerly and lazily compiling adaptive Weld, which can be seen in figure 6.4. We can see that joining the tables scales quite well for the regular adaptive variant. However, when lazy compilation is introduced, this is not the case. When using two threads we do get the benefits of latency hiding for the build task, but the runtime of the join task does not improve at all. For 8 threads we do see some improvement, but not nearly as much as it does for eagerly compiling Weld. Logs show that the Bloom filter flavor is used as soon as it is compiled, so that cannot explain the poor performance. Ultimately the cause appeared to be a bug, where any thread that executes a compilation task stops working after the task is finished, except in single-threaded scenarios. The issue is now solved, but this is not reflected in the results.



(a) Eager, 1 thread. (b) Eager, 2 threads. (c) Eager, 8 threads.

(d) Lazy, 1 thread. (e) Lazy, 2 threads. (f) Lazy, 8 threads.

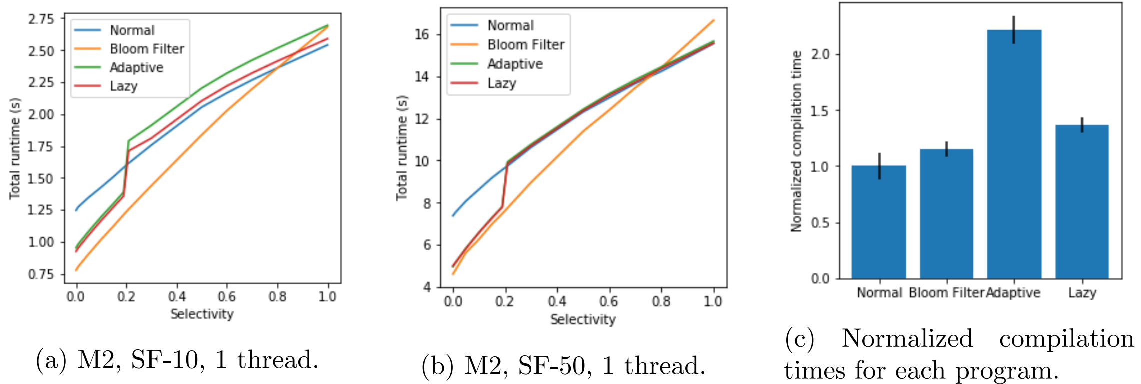
Figure 6.4: Timeline of tasks executed by the adaptive variations of the join query. The upper row shows eagerly compiled adaptive Weld, while the bottom row shows lazy compilation. The creation of the dictionary is omitted.

6.4.2 Three-way dictmerger join

To demonstrate the use of lazy compilation, we ran a micro-benchmark joining four tables, of the form $R \bowtie S \bowtie T \bowtie U$. We again vary the selectivity of the join of R on S, but keep the hit rates of $S \bowtie T$ and $T \bowtie U$ fixed at 0.55. Because this is higher than the threshold to build a Bloom filter, a maximum of only two out of eight flavors should be explored. Data was generated such that the right hand side of each join is unique, so each key in each dictionary maps to a single value, allowing us to implement the join with a single `for`-loop.

Figures 6.5a and 6.5b show the runtimes of each program for scale factor 10 and 50, respectively. For scale factor 10 the benefit of lazy function compilation is more convincing. Although it still shows a performance penalty over non-adaptive flavors, it is significantly less than what we see for eagerly compiling Weld. For scale factor 50 compilation times are ammortized, and eager and lazy compilation perform similarly. Figure 6.5c shows the normalized compilation times for each program. While lazy loop-adaptive Weld takes

significantly longer to compile than normal Weld, it still shows a large improvement over compiling all flavors up front.



(a) M2, SF-10, 1 thread.

(b) M2, SF-50, 1 thread.

(c) Normalized compilation times for each program.

Figure 6.5: Results of the three-way dictmerger join micro-benchmark, showing total query runtimes depending on selectivity for each program, for varying scale factors, as well as normalized compilation times.

6.4.3 Three-way groupmerger join

Another micro-benchmark we ran is similar to the one from section 6.4.2, but using `groupmergers` to construct the hash tables instead of `dictmergers`. This means that the dictionary values will not be single elements, but `vector` types, and therefore we need nested loops to perform the join. Because the data generated is the same as for 6.4.2, each `vector` in fact only contains a single element, and so the join will showcase the worst case overhead of the task creation that happens for nested loops in Weld.

Figure 6.6 shows the execution time of each program. We can see that adaptive Weld performs significantly better than even the best strategy for normal Weld, despite adaptive Weld running either the normal join or the Bloom filter join. The improved performance can therefore not be the result of loop-adaptivity. Instead, it is a side-effect of the fact that the runtime is disabled for any `for`-loop in a `switchfor`-expression. In normal Weld, every `for`-expression that has an inner loop invokes a runtime call each time it is evaluated, which evidently is the cause of significant overhead costs for this type of query.

To be able to properly measure the effects of loop-adaptivity, we added a configuration setting that disables the runtime for any nested loop. Figure 6.7 shows the performance of the same join query using this configuration, as well as a regular nested loop, a nested loop in a `switchfor`, and one using only a single loop (where the hash tables were constructed using `dictmergers`). We can see that disabling the runtime for nested loops indeed performs similarly to a nested loop in a `switchfor`-expression. Although much faster than a regular nested loop, it is still outperformed by a single loop. This makes sense, as a `for`-loop in Weld is implemented by a function call in LLVM, which is still expensive (although not as much as the runtime call that creates an additional task).

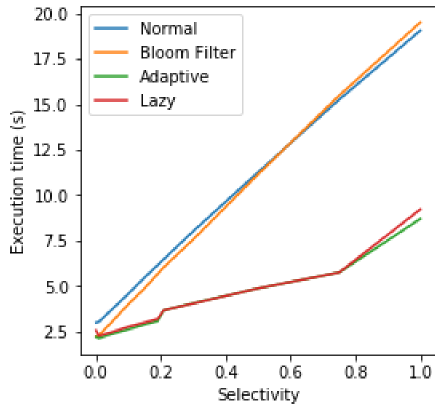


Figure 6.6: Execution time of normal flavors and adaptive variants for the three way groupmerger join.

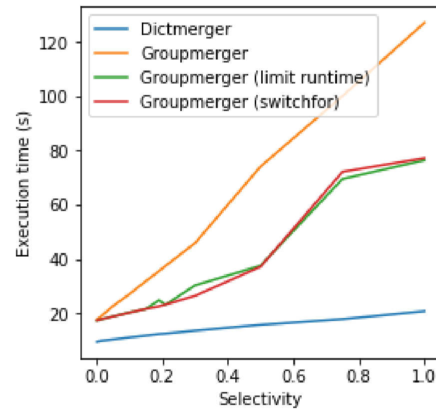
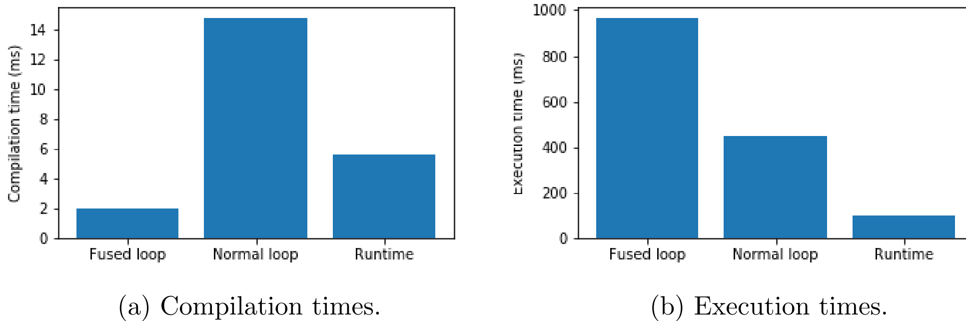


Figure 6.7: Execution time of a three way join comparing dictmerger, groupmerger in regular Weld, groupmerger and a switchfor, and groupmerger with the runtime limited to creating tasks for outer loops only.

6.5 Bloom filter creation

In Weld, there are three ways of creating a Bloom filter. The Bloom filter can either be created while the dictionary is being constructed (fused loop), in a separate loop or by passing a dictionary as argument to the `bloombuilder`-expression (which we will refer to as batch insert). We expect the first to be the fastest in terms of execution time, as it minimizes data movement and can be implemented using only few additional operations. However, for adaptive queries this is not possible, as the Bloom filter will be created only after we know the hit rate of a join operation, for which the dictionary should already be materialized. In this case the batch insert method is used, which should minimize compilation times as it is mostly implemented by a runtime call, and also perform better than a regular loop as it only requires a single function call for a batch rather than one for each insertion. To verify this, we ran a benchmark that, for each variant of Bloom filter creation, creates a dictionary of 25 million keys as well as a Bloom filter for the dictionary. As a baseline we run a program that only creates the dictionary. It took 62 ms to compile, and 7958 ms to execute.

Figure 6.8a shows the additional compilation time for each method of building a Bloom filter. Interestingly, the fused loop compiles even faster than the runtime call, although both versions are a lot more efficient than a separate loop. Figure 6.8b shows the performance of each variant in terms of execution time. Contrary to the philosophy of Weld, which always tries to fuse loops, a separate loop actually performs significantly better than a fused loop. This may be related the findings in [40], where in some cases *loop fission* (splitting a loop into two separate loops) outperforms the compound loop, due to improved locality of reference. Finally, the batch insert (runtime) method outperforms the



(a) Compilation times.

(b) Execution times.

Figure 6.8: Additional compilation and execution times for various methods of building a Bloom filter in Weld, with 25M insertions.

other two by a large margin. This is because the batch insert method only needs a single function call to insert its elements in the Bloom filter, rather than one for each element. Moreover, it is implemented in pure C++ and compiled by Clang [25], which typically generates more efficient machine code than Weld does.

6.6 Instrumentation overhead

Our final micro-benchmark evaluates the overhead caused by instrumentation. In particular, we look at the execution cost of the `count_calls` annotation, which increments a global variable whenever the expression that is annotated is evaluated. We evaluated a Weld program with a single `if`-expression, and a `merge` and multiplication in each branch. The `merge`-expressions were done with an `appender`, which (as we have seen in section 4.2) is relatively expensive. The `if` and `appender` ensure the loop is not SIMDized. Both the `if` and `merge`-expressions were annotated with `count_calls`, for a number of two `count_calls` for each iteration. Each `count_calls` increments a different variable.

Table 6.2: Cost of instrumented loop compared to its regular variant, for different machines.

Machine	Type	Cycles/Tuple	Cycles/`count_calls`
M1	normal	20.39	
	instrumented	21.48	**0.54**
M2	normal	23.82	
	instrumented	24.79	**0.48**
M4	normal	24.10	
	instrumented	24.64	**0.27**

Table 6.2 shows the results of the experiment. We can see that the impact of the `count_calls` annotation is relatively limited. In the worst case (M1), the slowdown per annotation is $(20.39 + 0.54)/20.39 = 1.03$. Considering that `count_calls` are meant to be used in instrumented `switchfor`-flavors that are only run on a fraction of the complete

data, this should be negligible.

6.7 TPC-H benchmarks

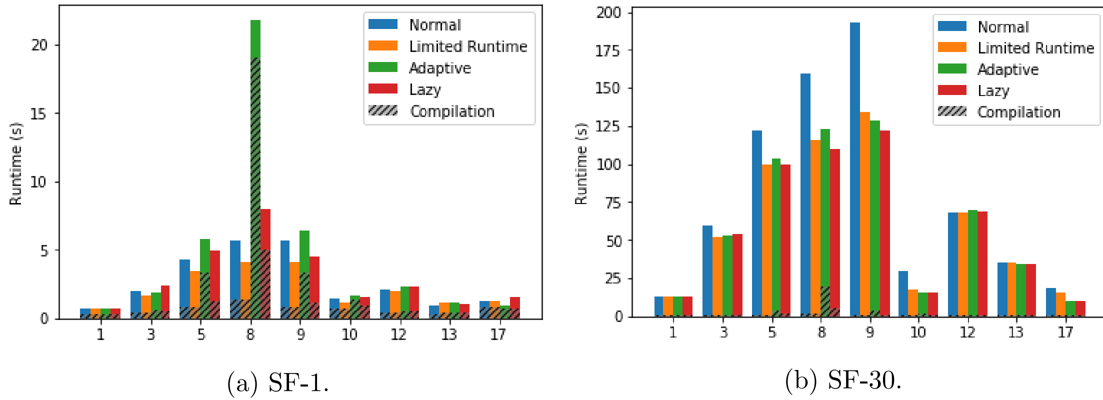
Next to the micro-benchmarks we ran a selection of queries from the TPC-H benchmark suite to see how loop-adaptivity in Weld performs on more realistic data. The Weld IR was generated using a Weld implementation in MonetDB [7] (which was the result of a separate research project on Weld). The selected queries are 1, 3, 5, 8, 9, 10, 12, 13 and 17. The reason for selecting these queries was that they were the only ones for which the MonetDB implementation could generate valid Weld IR. MonetDB does not produce Weld IR for query 17 that produces a correct result, but the IR itself is otherwise still executable and valid, and so we decided to keep it in anyway. Finally, none of the TPC-H queries include `order by` statements, because of limitations in Weld. Scale factors 1, 10 and 30 were used, both single threaded and using eight threads, and the benchmarks were run exclusively on machine 1.

We benchmarked four different versions of Weld. The first, *Normal*, is unmodified Weld. The second, *Limited Runtime*, is normal Weld, with the exception that nested loops are never parallelized and thus do not call the runtime. This is used as the baseline to which we compare our adaptive versions, to measure only the effects of loop-adaptivity and not the fact that nested loops in `switchfor`-expressions are never parallelized. The other two versions are *Adaptive* and *Lazy*, both using the adaptive optimization transformations, and the latter also using lazy flavor compilation.

6.7.1 Varying scale factors

Figure 6.9a and 6.9b show the results of single-threaded execution for SF-1 and SF-30, respectively. As we can see, for SF-1 compilation times often dominate the total runtime and so adaptive execution provides little benefits in this case. The exception is query 17, which has a highly selective join and thus benefits greatly from using Bloom filters. The lazy variant is slower however, because there is only a single dictionary so there is no real code explosion. As we have seen in section 6.4, lazy compilation only starts paying off for large numbers of flavors. Query 8 shows a clear effect of code explosion: the adaptive variant takes nearly 20 seconds to compile, much longer than the execution time. The code explosion comes from a `switchfor`-expression that has 128 flavors, one for every combination that can be made with seven Bloom filters. The lazy variant brings compilation times down by a large amount, but it's not enough to get it close to normal levels. All in all we can conclude that adaptive execution is not desirable for this scale factor.

Things look much better for SF-30. For the largest part the speedups must be attributed to disabling the runtime for nested loops, as we can see by the difference in runtime between normal Weld and the limited runtime version. However, the adaptive versions do provide some additional (albeit mostly marginal) speedups. In query 1 the optimizer applied adaptive predication, which resulted in no change in runtime. There are only two flavors so there is little increase in compilation time, and branching was picked about as often as predication by vw-greedy, possibly indicating that none of the flavors



(a) SF-1.

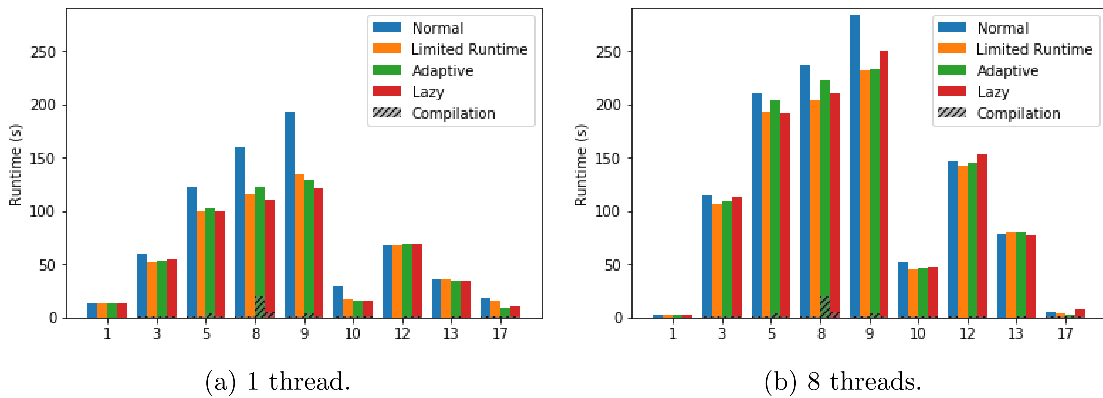
(b) SF-30.

Figure 6.9: Total single-threaded runtime of selected TPC-H queries for different versions of Weld, for varying scale factors.

provided any real benefit over the other. For all the other queries the optimizer applied adaptive Bloom filters, of which the best showcase is query 17. In most cases we can see that lazy compilation provides a performance increase over eagerly compiled adaptive Weld. Most notably, for query 8 adaptivity does not pay off when all flavors are compiled up front, while the lazy variant does.

6.7.2 Varying thread counts

Figure 6.10 shows the results for SF-30 using a single thread as well as using eight threads. With the exception of query 1 and 17, all queries actually take longer to finish rather than shorter. It appears that this has to do with the fact that all these queries use `groupmergers`, which apparently parallelize poorly. Unfortunately, adaptivity aggrevates the situation, especially in the case of lazy function compilation. We have discussed the cause of this in section 6.4.1.



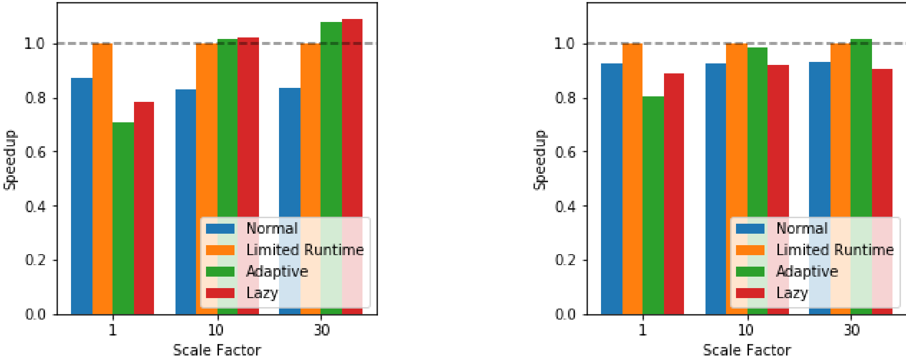
(a) 1 thread.

(b) 8 threads.

Figure 6.10: Total runtime of selected TPC-H queries for different versions of Weld, for varying numbers of threads, on SF-30.

6.7.3 Overall speedups

Figure 6.11 summarizes our findings with the TPC-H benchmarks. First of all, taking into account single-threaded execution, adaptivity clearly does not pay off for the lowest scale factor, but shows a moderate yet significant increase for higher scale factors. In general, lazy compilation shows a slight performance gain over eager compilation. Unfortunately the results are less bright when we execute the queries using multiple threads. We already discussed the poor performance of lazy compilation in multi-threaded environments due to a bug, but also eagerly compiling adaptive Weld performs worse than in the single-threaded case. This may be attributed to the fact that there is some additional locking whenever runtime statistics are updated, as well as the fact that we use slightly lower grain sizes for `switchfor`-tasks than for `for`-tasks. We have paid no specific attention to optimizing loop-adaptivity for parallelization, so there is likely plenty to gain. That said, using multi-threading in Weld rarely seems wise for the TPC-H queries anyway.



(a) 1 thread.

(b) 8 threads.

Figure 6.11: Geometric mean of the speedup of the selected TPC-H queries, for varying scale factors and numbers of threads. Limited runtime is used as the baseline.

Chapter 7

Conclusion

For this thesis we integrated loop-adaptivity in Weld, a JIT-compiling data processing framework. While it shows that (loop-)adaptivity is definitely possible in a JIT-compiling system, it is much less straightforward than doing this in a vectorized-interpreted DBMS like Vectorwise. We recall the research questions of this research.

Where are the opportunities for loop-adaptive execution in Weld, if any?

We have seen that opportunities for loop-adaptive execution in Weld are somewhat limited to those found in Vectorwise. Full computation never pays off, because of the poor code generation for `appender`s, as well as poor SIMD code. More promising however, is switching between branching and predicated expressions, as well as adaptively using bloom filters in hash joins. However, compared to Vectorwise we have only explored a limited number of opportunities.

Should Weld be adapted, and how do we design loop-adaptive execution in Weld?

We have seen at least two opportunities for loop-adaptivity in Weld, so its integration makes sense from a performance standpoint. Its design is the most challenging part of this research, for a number of reasons. The first two problems relate to code explosion: compilation times may increase to unacceptable amounts, as well as the time it takes to home in on the best variation. The third problem is instrumentation: how do we measure runtime statistics in a data-centric loop, while minimizing overhead?

Compilation times. We demonstrate two methods of keeping compilation times down: pre-compilation and lazy flavor compilation. The first method reduces compilation times by compiling functions for common tasks up-front. This should only be done for tasks that are both generic and pipeline breakers, so that nothing can be gained by JIT-compilation, as is the case for adaptively constructing Bloom filters after an instrumentation pass. By compiling flavors only as we need them, we can further reduce compilation times. However, compiling a single flavor in Weld is relatively expensive compared to compiling it up-front, because of shared functions that have to be compiled each time a Weld module

is compiled. It is therefore important to find a balance between eager and lazy compilation.

Exploiting the best flavor. The decision to implement adaptivity at the level of a `for`-expression rather than the operator level does mean we can only measure the aggregated performance of a pipeline rather than that of a single operator, which makes it harder to choose the right flavor. In the case of adaptive Bloom filters we are aided by an instrumented loop which measures the hit rate of dictionaries and can therefore often eliminate most flavors, but for adaptive predication we chose to simplify the optimization pass to never produce more than two flavors, which may be too simplistic. In all, we have not answered the question of how to timely find the best variation in the face of flavor explosion. Answering the remaining questions appeared to take so much time that it fell outside the scope of the project.

Instrumentation. For loop-adaptivity we need to measure the performance of operators while keeping the associated overhead down. Like many JIT-compiled systems Weld fuses operators into a single tuple-at-a-time pipeline, but this did not pose a challenge because the tasks for the `for`-loops in which they are implemented are split into sub-tasks by Weld, of which we can easily measure the execution time.

For instrumented loops we do measure on a tuple-at-a-time basis, but this is only done on a fraction of the data, amortizing the measuring overhead. We introduced global variables in Weld to be able to count the number of hits and misses in a dictionary. Weld is a functional language, so global variables do not fit well with the language. Our solution is implementing it only in annotations, which means that the core language keeps its functional properties.

What are the potential speedups gained by loop-adaptivity in Weld?

Looking at the micro-benchmarks, the speedups of loop-adaptivity are significant, but never dramatic. This is similar to the findings in Vectorwise. Lazy flavor generation shows clear benefits of compiling everything up-front, but only if the number of flavors that are never tried is reasonably larger than the number of explored flavors. Also, the query size needs to be sufficiently large so that the execution time dominates the compilation time. Similar findings come from the TPC-H benchmarks, where adaptivity only starts paying off at scale factor 10 and higher. Most queries gain some speedups from loop-adaptivity, but there are also queries that in fact perform worse, especially when lazy flavor compilation is not used.

Other challenges

Although not explicitly includes as a research question, another challenge was defining the flavors. In an interpreted system there is a predefined number of functions that implement each operator, for which we can then define a limited number of flavors. However, because we have to base adaptivity on the `for`-expression level, we have a virtually infinite number of loops for which there can then be an infinite number of flavors. We therefore resort to defining the various flavors during the Weld IR optimization stage. We have defined three

different optimization passes that can detect different opportunities for loop-adaptivity, and that can produce adaptive Weld IR accordingly, using the newly introduced `switch-for`-expression.

The final challenge is Weld itself. We have seen that Weld suffers from a number of problems that make it harder to implement loop-adaptivity. One of them is the high costs of using `appenders`, which makes that full computation never outperforms a fused loop. Also, the machine code that is generated in general is quite poor, mostly due to lacking compiler hints. Another problem is the poorly scaling implementation of dictionaries, in particular `groupmerger`, tainting the performance picture of parallel experiments. Finally, the high compilation times of Weld in general, which means that adaptivity only pays off for relatively high scale factors, and that lazy flavor generation can take up a large portion of execution time.

7.1 Future Work

A number of topics are left for future of work. First of all, we have only looked at code explosion with regards to the increase of compilation times, but we still have to tackle the increased search space, which vw-greedy cannot comfortably deal with. In [39], Rosenfeld et al. extended micro-adaptivity to deal with larger search spaces, which may also work for loop-adaptivity in Weld.

Compilation times are already significantly reduced by lazily generating and compiling LLVM for some flavors, but this could be further brought down by also lazily generating its Weld IR. Also, compilation tasks can only be run sequentially, but if we could make the compiler thread safe we could do this in parallel to decrease the latency of lazy compilation. Moreover, we would like to investigate how compilation times could be further reduced by exploiting the similarity between flavors. For instance, two flavors of a join may be highly similar, and only differ in the existence or absence of a Bloom filter check. If we could somehow make sure that the code for the similar part is generated just once, we might keep compilation times much closer to that of normal Weld.

Another topic for future work is combining loop-adaptivity with cost modeling. The decision to build a bloom filter is based on an extremely simple threshold for now, and could be much improved by using more advanced cost models. Although perhaps conflicting with the philosophy of micro-adaptivity, in some cases it seems inevitable to resort to these kind of heuristics.

Finally, there are more opportunities for loop-adaptivity in Weld to be discovered. For instance, we have seen that building the Bloom filter in the same loop as building a dictionary is often more costly than building the Bloom filter in separate loop. Yet, in most other cases fused loops seem to outperform separate loops. This, and other situations, may be a case for loop-adaptivity.

Bibliography

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[2] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[3] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 119–130. ACM, 2005.

[4] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418, 2004. ISSN 07308078. doi: http://doi.acm.org/10.1145/1007568.1007615.

[5] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. Adaptive and big data scale parallel execution in oracle. *Proceedings of the VLDB Endowment*, 6(11):1102–1113, 2013. ISSN 21508097. doi: 10.14778/2536222.2536235.

[6] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[7] Peter Alexander Boncz et al. *Monet: A next-generation DBMS kernel for query-intensive applications.* Universiteit van Amsterdam [Host], 2002.

[8] Francis Chu, Joseph Y Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 138–147. ACM, 1999.

[9] John Cieslewicz and Kenneth a. Ross. Adaptive aggregation on chip multiprocessors. *Proceedings of the 33rd International Conference on Very Large Data Bases - VLDB '07*, pages 339–350, 2007.

[10] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[11] Amol Deshpande and Zachary Ives. Adaptive Query Processing : Why , How , When , What Next ? pages 1426–1427, 2007.

[12] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007. ISSN 1931-7883. doi: 10.1561/1900000001.

[13] Craig Freedman, Erik Ismert, and På Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin*, pages 22–30, 2014. URL `ftp://131.107.65.22/pub/debull/A14mar/p22.pdf`.

[14] Yoshihiko Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Compurters, Controls*, 25:45–50, 1971.

[15] Sumit Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, volume 98, pages 228–238, 1998.

[16] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[17] Joseph M Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS)*, 23(2):113–157, 1998.

[18] Yannis E Ioannidis and Stavros Christodoulakis. *On the propagation of errors in the size of join results*, volume 20. ACM, 1991.

[19] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, page 395, 2004. ISSN 07308078. doi: 10.1145/1007568.1007613. URL `http://portal.acm.org/citation.cfm?doid=1007568.1007613`.

[20] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[21] Alfons Kemper and Thomas Neumann. HyPer : A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. *Icde*, pages 195–206, 2011. ISSN 1063-6382. doi: 10.1109/ICDE.2011.5767867. URL `http://www.cs.albany.edu/ jhh/courses/readings/kemper.icde11.memory.pdf`.

[22] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10): 853–864, 2014.

[23] Christoph Koch. Abstraction Without Regret in Database Systems Building: a Manifesto. *IEEE Data Engineering Bulletin*, 37(1):70–79, 2014. URL `http://infoscience.epfl.ch/record/197359`.

[24] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive Execution of Compiled Queries. *Icde*, (iii), 2018. doi: 10.1109/ICDE.2018.00027.

[25] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[26] LLVM Language Reference Manual. LLVM language reference manual. `https://llvm.org/docs/LangRef.html`. Accessed: 2018-06-19.

[27] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670. ACM, 2004.

[28] Microsoft Data Platform. Microsoft Data Platform. `https://www.microsoft.com/en-us/sql-server/`. Accessed: 2018-08-03.

[29] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011. ISSN 21508097. doi: 10.14778/2002938.2002940. URL `http://dl.acm.org/citation.cfm?doid=2002938.2002940`.

[30] Thomas Neumann and César A Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*, pages 73–92. Citeseer, 2013.

[31] Thomas Neumann and Viktor Leis. Compiling Database Queries into Machine Code. *IEEE Data Engineering Bulletin*, pages 3–11, 2014.

[32] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the Interface Between Data-Intensive Applications. 2017. URL `http://arxiv.org/abs/1709.06416`.

[33] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, Matei Zaharia, S Palkar, J Thomas, D Narayanan, P Thaker, R Palamuttam, and P Negi. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Pvldb*, 11(9):1002–1015, 2018. doi: 10.14778/3213880.3213890. URL `http://www.vldb.org/pvldb/vol11/p1002-palkar.pdf`.

[34] Li Quanzhong, Shao Minglong, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. Adaptively reordering joins during query execution. *Proceedings - International Conference on Data Engineering*, (May 2014):26–35, 2007. ISSN 10844627. doi: 10.1109/ICDE.2007.367848.

[35] Vijayshankar Raman, Amol Deshpande, and Joseph M Hellerstein. *Using state modules for adaptive query processing*. IEEE, 2003.

[36] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. Compiled query execution engine using jvm. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 23–23. IEEE, 2006.

[37] Herbert Robbins. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, pages 169–177. Springer, 1985.

[38] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[39] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The Operator Variant Selection Problem on Heterogeneous Hardware. *Adms@Vldb*, 2015.

[40] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in a Vectorized Database System. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 2012.

[41] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in Vectorwise. *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, (June 2013):1231, 2013. ISSN 07308078. doi: 10.1145/2463676.2465292. URL `http://dl.acm.org/citation.cfm?doid=2463676.2465292`.

[42] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1907–1922. ACM, 2016.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[44] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 307–322. ACM, 2018.

[45] The Scala Programming Language. The Scala Programming Language. `https://www.scala-lang.org/`. Accessed: 2018-08-02.

[46] TPC-H. TPC-H. `http://www.tpc.org/tpch/`. Accessed: 2018-08-02.

[47] Joannes Vermorel and Mehryar Mohri. Multi-Armed Bandit Algorithms and Empirical Evaluation BT - Machine Learning: ECML 2005. *Machine Learning: ECML 2005*, pages 437–448, 2005. doi: 10.1007/11564096_42.

[48] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1):31–37, 2014. URL `http://dblp.uni-trier.de/db/journals/debu/debu37.html#Wanderman-MilneL14`.

[49] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[50] Annita N Wilschut and Peter MG Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[52] Marcin Zukowski, Peter A Boncz, et al. Vectorwise: Beyond column stores. 2012.

Appendix A

Literature Study

Table A.5 displays the selected literature for the literature research. Table A.1 and table A.3 show the criteria that were used to include or exclude tables respectively. For each paper we note whether we read only the title (**T**), abstract (**A**) or the full text (**F**). For papers that present some query compiler, we also note whether the system uses multiple levels of abstraction (**MA**), and whether it follows the abstraction without regret principle (**AWR**) [23]. We use a slightly more loose definition than Koch: we do not explicitly require the system to use multiple levels of abstraction, but the system should be built using a high-level programming language, while retaining maximum performance.

Table A.5: Literature inclusion table.

#	Rel.	Authors	Title	Year	Topic	Incl.				E.	Read			JIT systems	
						1	2	3	4	1	T	A	F	MA	AWR
1	-	G. Graefe	Volcano - An Extensible and Parallel Query Evaluation System	1994	Other				x		x				
2	+	S. Babu et al.	Adaptive ordering of pipelined stream filters	2004	AQP		x					x			
3	+	Z. Ives et al.	Adapting to source properties in processing data integeration queries	2004	AQP							x			
4	+	J. Vermorel et al.	Multi-Armed Bandit Algorithms and Empirical Evaluation BT	2005	MA				x				x		
5	+	J. Rao et al.	Compiled query execution engine using JVM	2006	JIT				x		x			no	no
6	++	A. Dasphande et al.	Adaptive Query Processing	2007	AQP		x		x				x		
7	++	J. Cieslewicz et al.	Adaptive aggregation on chip multiprocessors	2007	AQP		x						x		

8	++	L. Quanzhong et al.	Adaptively reordering joins during query execution	2007	AQP		x						x		
9	+	A. Dasphande et al.	Adaptive Query Processing: Why, How, When, What Next?	2007	AQP		x						x		
10	++	T. Neumann	Efficiently compiling efficient query plans for modern hardware	2011	JIT	x							x	no	no
11	-	A. Kemper et al.	HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots	2011	Other				x				x		
12	++	B. Raducanu	Master Thesis - Micro Adaptivity in a Vectorized DBMS	2012	MA		x	x					x		
13	++	B. Raducanu et al.	Micro adaptivity in Vectorwise	2013	MA		x	x					x		
14	+	T. Neumann et al.	Taking the Edge off Cardinality Estimation Errors using Incremental Execution	2013	AQP		x					x			
15	+	S. Bellamkonda et al.	Adaptive and big data scale parallel execution in oracle	2013	AQP		x						x		
16	++	C. Koch	Abstraction with regret in data management systems	2013	JIT	x							x		
17	++	C. Freedman et al.	Compilation in the Microsoft SQL Server Hekaton Engine	2014	JIT	x							x	yes	no
18	++	T. Neumann et al.	Compiling Database Queries into Machine Code	2014	JIT	x							x	no	no
19	+	S. Wanderman-Milne et al.	Runtime Code Generation in Cloudera Impala	2014	JIT	x							x	no	no
20	++	C. Koch	Abstraction Without Regret in Database Systems Building: a Manifesto	2014	JIT	x							x		
21	-	A. Dutt et al.	Plan Bouqets: Query Processing without Selectivity Estimatin	2014	AQP		x			x		x			
22	++	V. Rosenfeld et al.	The Operator Variant Selection Problem on Heterogeneous Hardware	2015	MA		x					x			

23	-	M. Lee et al.	A JIT compilation-based unified SQL query optimization system	2016	JIT	x							x	?	?
24	++	A. Shaikhha et al.	Building Efficient Query Engines in a High-Level Language	2016	JIT	x							x	yes	yes
25	++	A. Shaikhha et al.	How to Architect a Query Compiler	2016	JIT	x							x	yes	yes
26	++	S. D. Viglas	Processing declarative queries through generating imperative code in managed runtimes	2017	JIT	x						x		?	?
27	++	S. Palkar et al.	Weld: Rethinking the Interface Between Data-Intensive Applications	2017	Weld/JIT	x							x	yes	yes
28	++	S. Palkar et al.	Evaluating End-to-End Optimization for Data Analytics Applications in Weld	2018	Weld/AQP	x	x						x	yes	no
29	++	A. Kohn et al.	Adaptive Execution of Compiled Queries	2018	AQP/JIT	x	x						x		
30	++	R. Tahboub et al.	How to Architect a Query Compiler, Revisited	2018	JIT	x							x	no	yes

Table A.1: Inclusion criteria.

	Criterion	Rationale
1	A study that directly proposes software techniques or strategies to implement query compilers.	We are interested in bridging the gap between adaptivity and query compilers. Approaches taken to implement the latter are therefore relevant to us.
2	A study that directly proposes software techniques or strategies to implement some form of adaptive query processing.	We are interested in bridging the gap between adaptivity and query compilers. Approaches taken to implement the first are therefore relevant to us.
3	A study that uses or extends the concept of micro-adaptivity.	We will bridge the gap between adaptivity and query compilers by implement a form of micro-adaptivity in a query compiler. We are therefore particularly interested in articles on micro-adaptivity, as well as any extensions to it.
4	A study that provides context for at least one of the above.	Some articles may not directly discuss any of the above, but provide some context for them anyway. These will not be discussed in detail, but may be referenced.

Table A.3: Exclusion criteria.

	Criterion	Rationale
1	An AQP study that are only suitable for inter-query adaptivity, or only work in streaming environments.	We are only interested in intra-query adaptivity in mainstream data processing.
2	A query compiler that is relatively old.	We are mostly interested in the state-of-the-art. Papers about older query compilers will be referenced, but not explored in detail.