



**Vrije Universiteit Amsterdam**
Faculty of Sciences, Department of Computer Science

**Centrum Wiskunde & Informatica**

**Mihai Varga**
student number 2591442

# Just-in-time compilation in MonetDB with Weld

**Master's Thesis in**
**Parallel and Distributed Computer Systems**

Supervisor:
**Prof. Dr. Peter Boncz**
Vrije Universiteit Amsterdam
Centrum Wiskunde & Informatica

Second reader:
**Dr. Hannes Mühleisen**
Vrije Universiteit Amsterdam
Centrum Wiskunde & Informatica

July 2018

**Abstract**

Query evaluation techniques that rely on interpretation often incur overhead generated by the interpretation process itself and the materialization between the operators. These issues can be solved through the query compilation technique, in which a query is evaluated by generating and compiling a specifically tailored program. In this thesis we aim to solve MonetDB's excessive intermediate materialization overhead through just-in-time (JIT) query compilation. For this purpose we use Weld, a library which offers JIT compilation and data flow optimizations through its functional intermediate representation. We evaluate two Weld integration approaches into MonetDB, by translating individual operators from MonetDB's columnar algebra, and by producing a complete Weld program from the relational algebra using a data-centric code generation technique.

# Contents

# Chapter 1

# Introduction

In the fields of business intelligence and data mining we often encounter analytical workloads characterized by ad-hoc queries that are meant to be served by the underlying databases interactively. MonetDB [5] is a columnar store database developed at the Centrum Wiskunde & Informatica that specializes in online analytical processing (OLAP). MonetDB has been around for almost 20 years and has offered a pioneering solution to speedup the execution time of analytical queries by introducing the column-at-a-time processing model.

The MonetDB interpretation pipeline processes an incoming query in several steps, as shown in Figure 1.1 on the left side. First, the query is parsed and translated into relational algebra, a formal representation of the operations that are about to be performed on the relational data. Then, the relational algebra is transformed into MonetDB Assembly Language (MAL) which is an internal representation of the query plan in the form of operations that are applied on one or several columns at a time. MAL is an interface to the underlying kernel called GDK, which acts as the execution engine and is responsible for actually executing the operations.

Boncz et al. show in [6] that unfortunately MonetDB's implementation of the column-at-a-time processing model suffers from excessive intermediate materialization between the operators. In this thesis we investigate whether the query execution time can be improved and the amount of materialization reduced by integrating Just-In-Time (JIT) compilation into MonetDB's query processing pipeline. JIT compilation is the process of generating executable code on the fly, at runtime. The benefit of JIT is that an optimized compiled program can be created for ad-hoc queries, by taking advantage of any information that can be derived from it, from the underlying system or from the data flow of the query.

In our implementation we use Weld [14] to deploy JIT compilation inside MonetDB. Weld is a common runtime that aims to optimize data intensive applications by fusing work from different systems. It offers a functional intermediate representation (IR), which allows developers to express computations across different libraries and then rely on the runtime to optimally execute them. We identify two Weld integration opportunities. The first method works at the MAL interpretation layer of MonetDB and replaces MAL instructions that would otherwise call functions from GDK with new instructions that generate Weld code. The second idea is to produce Weld code directly from the relational algebra using the code generation technique described in [12] and which we will detail in the following chapters. The two approaches are shown side by side with the canonical MonetDB interpretation pipeline in Figure 1.1.

This thesis is structured as follows: Chapter 2 provides background information on MonetDB and Weld; we present our motivation and summarize the two integration techniques in Chapter 3; Chapter 4 provides implementation details of our work and in Chapter 5 we evaluate the performance of the new query system in its two variants; we discuss related work in Chapter 6 and, finally, in Chapter 7 we conclude our work.

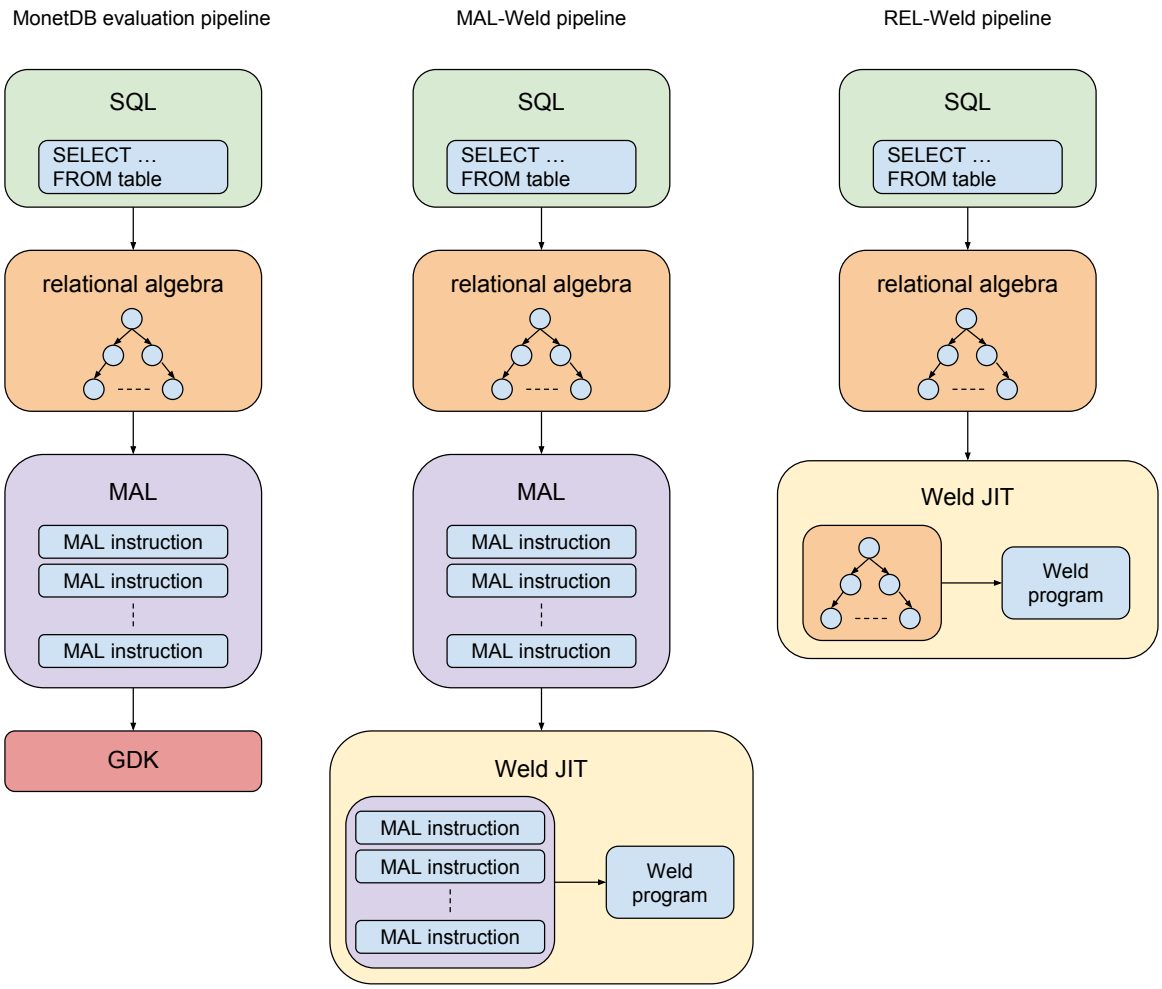


Figure 1.1: The three query evaluation pipelines considered in this thesis. Left: the standard MonetDB pipeline where the SQL query is transformed into a relational algebra, then the algebra is used to generate a MAL program, which is eventually evaluated by the GDK kernel. Center: the MAL-Weld pipeline in which we assemble a Weld program from smaller Weld statements generated for individual MAL instructions. Right: The REL-Weld pipeline in which we produce the complete Weld program based on the relational algebra.

# Chapter 2

# Background

This chapter offers background information related to our work. We first begin by discussing several query evaluation techniques, how they perform on modern hardware and what lead to the adoption of the column-at-a-time processing model in MonetDB. We then describe MonetDB's architecture and introduce the components we worked with in our Weld integration. Lastly, we discuss Weld's capabilities as a runtime and as a JIT compilation system.

## 2.1 Database systems

A Database Management System (DBMS) allows users to store, manipulate and access data in a database. A special form of DBMS is the Relational DBMS in which the databases hold records using a relational data model. The stored data is usually split into tables where columns define the attributes and each row constitutes a record. The tables can be defined by a schema which holds information about the physical structure of the tables and the relationships between them.

Data stored in a relational DBMS can be queried with dedicated domain-specific languages, with SQL being the most popular example. The interpretation engine of a DBMS uses an abstract representation of the query, called the relational algebra. The algebra expresses the query in terms of connected operators, very often in a tree-like structure, in which the internal nodes represent algebraic operators such as *Select*, *Project* or *Join* and leaves represent the relations (the data). These operators will be described in Section 2.3.1 in the context of MonetDB.

## 2.2 Query evaluation techniques

The most common approach to evaluating a query is to have a set of predefined operators that can be arranged in any order to express complex computational intents. Any SQL query can be expressed in such a tree of operators. The freedom to formulate ad-hoc queries forces the system to deal with them dynamically, typically by creating an interpreter for such operator trees.

The Volcano iterator model [8] is one of the best known processing models for such a system. With Volcano, each operator from the relational algebra tree exposes three canonical methods, *open()*, *next()* and *close()*, and the query is evaluated by recursively calling *next()* on each node and then its children, starting from the root, until all the tuples have been pulled through the tree. This type of evaluation data flow is known as *tuple-at-a-time*. Due to its simplicity the Volcano model is a popular choice for DBMSs. However it incurs a high instruction interpretation overhead. The goal of a query engine is to process data as fast as possible, which means utilizing the available hardware components, such as the

CPU or the memory, at their peak performance. On modern hardware the memory and I/O bandwidths have increased to a point where the instruction interpretation cost has become non negligible and can actually hinder the rest of the system. While the query interpretation has also become faster with the advancements in hardware, CPU cycles spent on this could instead be used to actually operate on data. Ailamaki et al. [2] show that databases can suffer from poor code and instruction locality which leads to excessive CPU stalls and ultimately suboptimal hardware usage. The poor performance can be attributed to the tuple-at-a-time evaluation technique. The recursive *next()* calls extensively use the instruction cache and generate numerous cache misses, which in turn prevent the compiler from exploiting modern CPU features such as deep pipelining and SIMD instructions. At the same time, it fails to expose the potential of processing multiple tuples in parallel, leading the CPU to achieve low IPC (instructions per cycle).

MonetDB is best known for its *column-at-a-time* processing model, in which entire columns are processed in a tight loop. Compared to the tuple-at-a-time model, the internal nodes from the query plan no longer consume a single row, but an entire column at once, and can similarly produce a whole column as their result. The main benefit of this evaluation technique is that the interpretation overhead is no longer proportional to the number of tuples, but instead to the number of operators. In MonetDB, the interpretation cost is determined by the length of the MAL program which has been generated from the relational algebra. Therefore, for MonetDB the interpretation overhead is negligible compared to the amount of useful operations.

A variant of the tuple-at-a-time model has been implemented in a new query engine for MonetDB called X100 [6]. The technique named "vectorized execution model" reduces the interpretation overhead of the tuple-at-a-time model by moving a vector of tuples through the tree of operators, instead of a single tuple.

A different query execution technique is presented by Neumann in [12]. It relies on compiling the query to machine code rather than interpreting it as in the iterator model. This new approach comes from the observation that the traditional algebraic operator model is focused on the operators themselves, thus drawing a clear boundary between the execution phases of the query. Instead, by compiling the query to native machine code, the processing becomes data-centric. This new data flow is denoted by Neumann as the *produce-consume* model in [12]. The resulting execution model is also tuple-at-a-time, but unlike in the Volcano model where data is pulled upwards in the query plan, data is now pushed from one operator to another. The advantages of this approach over interpreting the query are the lack of interpretation overhead and the fact that the generated code can be tuned to perform optimally on the given hardware.

The *produce-consume* code generation technique works by identifying operator pipelines in the relational algebra. The pipeline is defined as a chain of operators that do not materialize the data in their evaluation. At the opposite end there are pipeline-breaker operations, which require the entire data to be made available before producing any result. For example, an aggregation that computes the sum of the values in a column requires all the tuples to be evaluated before producing the result, whereas a projection that doubles the values in a column can apply the multiplication operation on individual data points and immediately make them available to the next operator. We will discuss operator pipelines in more detail in Chapter 4, in which we also provide an example in Figure 4.2 of how pipelines can be identified in the query plan. The performance gains of the produce-consume evaluation technique is that materialization is limited only to the operators that absolutely require it and the fact that during the execution of pipelined operators the tuples can remain in the fast CPU cache while several operators are applied on them. In other words, the techniques benefits from a better data cache locality.

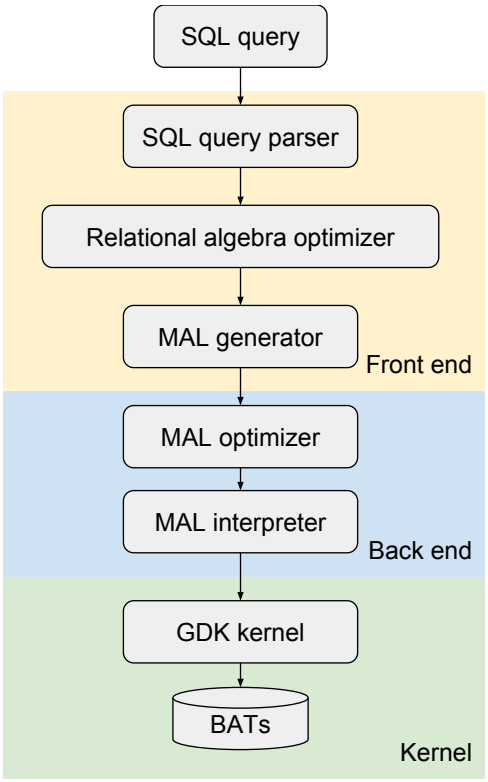


Figure 2.1: MonetDB's three layered architecture.

## 2.3 MonetDB

Relational table data in MonetDB is stored column-wise, in specialized data structures called binary association tables, or BATs. A BAT has two columns, a *head* and a *tail*. The *head* column usually column usually represents a unique object identifier (oid) and the *tail* holds the actual data values. The architecture of MonetDB is shown in Figure 2.1 and is split into the frontend, the backend and the kernel. An incoming SQL query is parsed, abstracted into an algebraic tree and subsequently transformed into MAL instructions in the frontend. The MAL instructions are then optimized in the backend. MAL represents an interface to the underlying kernel, the GDK, which offers access to the storage layer. The optimized MAL plan is passed to the kernel for execution. Complex expressions from the relational algebra can be broken down into a sequence of BAT-specific algebraic operators in the frontend, which map to MAL instructions and are implemented in the kernel as simple and CPU-friendly array operations.

MonetDB is a DBMS that specializes in providing high performance for analytical queries [5] and its main performance drives are the columnar storage and the BAT algebra operations that are used to express the query plan in simple operations applied on whole columns at a time. The points of interest for our thesis are the MAL generator and the MAL optimizer. The MAL generator is a module that emits MAL instructions based on the algebraic tree - we will replace the generated instructions with Weld code. The MAL optimizer creates an optimal set of MAL instructions right before the MAL interpreter calls into the GDK kernel for their interpretation. We will modify the optimizer's workflow and instead use it to emit Weld code from groups of MAL instructions.

### 2.3.1 Relational algebra

The relational algebra is generated after the SQL query has been parsed. It provides a logical plan for the query and it can be visualized as a tree where the nodes are algebraic operators and the edges are data

dependencies. To showcase an example of the relational algebra produced by MonetDB, we will use the SQL query from Figure 2.2 which is take from [12] as it contains the most common algebraic operations: Select, Project, GroupBy and Join. MonetDB translates the query in Figure 2.2 to the relational algebra in Listing 2.1. Later this query will be referenced again to show the data pipelines that can be identified from the plan and how we can generate Weld code that perfectly fits on the pipeline model. We will now briefly explain the basic relational algebra operators in MonetDB.

Table Is the operator used for reading data from disk. It works with a single table at a time, and as the storage model is MonetDB is columnar, it returns only those columns that are needed throughout the query execution.

Select Filters data based on one or more predicates.

Project It can select a subset of columns from the input, apply operations on them and produce new columns. It can also be used to simply rename columns. In addition, the operator can be used to sort the data given a sort key.

Group By Performs an aggregation over a set of columns using the specified aggregation function. If there is a Group By key, the result will be a set of columns, otherwise the aggregations will go into a single bucket and the result will be a set of scalars.

Join The Join operator family, including Semi Join, Anti Join, Outer Join, etc. combine tuples from the left and the right hand sides of the join relations. MonetDB can choose at runtime between implementation such as merge or hash join.

TopN The TopN operator selects only a range of values from the resulting output.

```
select  *

from    R1, R3
        (select R2.z, count(*)
        from R2
        where R2.y = 3
        group by R2.z) R2

where   R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

Figure 2.2: SQL query from [12].

Listing 2.1: Relational algebra produced by MonetDB for Figure 2.2

```
| project (
| | join (
| | | join (
| | | | select (
| | | | | table(sys.r1) [ "r1"."a" NOT NULL, "r1"."x" NOT NULL ] COUNT
| | | | ) [ "r1"."x" NOT NULL = int "7" ],
| | | | table(sys.r3) [ "r3"."b" NOT NULL, "r3"."c" NOT NULL ] COUNT
| | | ) [ "r1"."a" NOT NULL = "r3"."b" NOT NULL ],
| | | project (
| | | | group by (
| | | | | select (
| | | | | | table(sys.r2) [ "r2"."y" NOT NULL, "r2"."z" NOT NULL ] COUNT
| | | | | ) [ "r2"."y" NOT NULL = int "3" ]
| | | | ) ["r2"."z" NOT NULL ] ["r2"."z" NOT NULL, sys.count() NOT NULL as "L3"."L3"]
| | | ) [ "r2"."z" NOT NULL, "L3" NOT NULL as "r2"."L3" ]
| | ) [ "r3"."c" NOT NULL = "r2"."z" NOT NULL ]
| ) [ "r1"."a" , "r1"."x" , "r3"."b" , "r3"."c" , "r2"."z" , "r2"."L3" ]
```

### 2.3.2 MAL and GDK

MAL specifies the query's data flow. As MAL is interpreted through the GDK kernel that interfaces with the actual operating system and storage layer, the MAL-GDK execution can be likened to a virtual machine architecture. The MAL program is generated from the optimized relational algebra in the frontend and is then passed to the backend to be further optimized. A MAL program can have variables, instructions and functions, and it can specify complex computations while still maintaining a level of abstraction over GDK. BATs are at the heart of a MAL program, and as BATs are immutable in a typical read-only analytical query, each operation that is applied on a BAT will result in the creation of a new transient BAT, i.e. a BAT that is valid only for the duration of the query evaluation.

At the storage level MonetDB stores individual columns from the relational table as <head, tail> tables. The head column is the oid (object identifier) which maps to an index in the tail column. In practice the oids are consecutive values in ascending order and the head column is not actually written to disk or represented in memory, as the oids can be computed knowing just the first oid and the number of elements. On disk and in memory, a BAT is an ordinary C-array which makes reading and processing data columns trivial. A string column is handled differently from columns of other data types: a string BAT is backed up by two arrays, one containing the actual null-terminated strings separated by metadata, and the other containing indexes that tell where the individual strings start from in the string array.

As we will be dealing with MAL instructions in our Weld translation it is useful to understand the syntax. MAL is a strongly typed language and therefore every variable or parameter, BAT or scalar, has an associated type. The base types are bit (int8), bte (int8), sht (int16), int (int32), lng (int64), oid (int64), flt (float), dbl (double) or str. MonetDB supports hge (int128) as well, but as Weld doesn't have an equivalent we have disabled its support. Instructions also accept parameters with a polymorphic type named any_1 which is resolved at runtime. Namespaces, or MAL modules, are designed to improve the organization and the type and function resolutions. An instruction's signature consists of the MAL module and the instruction name. For example, *algebra.thetaselect* is the notation for the *thetaselect* function that resides in the *algebra* module. Listing 2.2 shows shows the signature of *thetaselect*, which takes as arguments a data column *b* with lng values, a candidate list which is also a BAT, a constant *val* - the right operand and a string constant called *op* that represents the comparison operation. The instruction returns the oids of the values that satisfy the $x <$ *op* predicate.

Listing 2.2: MAL instruction showcase. The *thetaselect* instruction is part of the *algebra* module, it takes four arguments (two BATs and two constants) and returns a single BAT

```
/* Instruction signature */
algebra.thetaselect(b:bat[:any_1], s:bat[:oid], val:any_1, op:str):bat[:oid]

/* The instruction used in a MAL program */
X_3:bat[:oid] := algebra.thetaselect(X_1:bat[:lng], X_2:bat[:oid], 2400:lng, "<":str);
```

As MAL instructions are executed one-by-one, control is passed for each to the GDK library where the actual data processing part takes place. Certain MAL instructions, such as *algebra.join*, have several implementations to choose from. The GDK library can determine at runtime which one to use, for example which join strategy to choose, hash or merge join, depending on whether the input columns are sorted or not. Then, if for example the hash join algorithm is chosen, GDK can decide on which of the two columns to build the hash table and which to probe with. Choosing the algorithm implementation dynamically is what allows MonetDB to execute the core part of the algorithm in a tight loop over the data. Listing 2.3 shows the main part of the MAL program which is generated from the query in Figure 2.2.

Listing 2.3: MAL program generated from the relational algebra in Listing 2.1

```
X_20:bat[:int] := sql.bind(X_6:int, "sys":str, "r1":str, "x":str, 0:int);
C_7:bat[:oid] := sql.tid(X_6:int, "sys":str, "r1":str);
C_29:bat[:oid] := algebra.thetaselect(X_20:bat[:int], C_7:bat[:oid], 7:int, "==":str);
X_10:bat[:int] := sql.bind(X_6:int, "sys":str, "r1":str, "a":str, 0:int);
X_31:bat[:int] := algebra.projection(C_29:bat[:oid], X_10:bat[:int]);
C_33:bat[:oid] := sql.tid(X_6:int, "sys":str, "r3":str);
X_35:bat[:int] := sql.bind(X_6:int, "sys":str, "r3":str, "b":str, 0:int);
X_41:bat[:int] := algebra.projection(C_33:bat[:oid], X_35:bat[:int]);
(X_49:bat[:oid], X_50:bat[:oid]) := algebra.join(X_31:bat[:int], X_41:bat[:int], nil:
    BAT, nil:BAT, false:bit, nil:lng);
X_42:bat[:int] := sql.bind(X_6:int, "sys":str, "r3":str, "c":str, 0:int);
X_58:bat[:int] := algebra.projectionpath(X_50:bat[:oid], C_33:bat[:oid], X_42:bat[:int
    ]);
X_61:bat[:int] := sql.bind(X_6:int, "sys":str, "r2":str, "y":str, 0:int);
C_59:bat[:oid] := sql.tid(X_6:int, "sys":str, "r2":str);
C_77:bat[:oid] := algebra.thetaselect(X_61:bat[:int], C_59:bat[:oid], 3:int, "==":str)
    ;
X_68:bat[:int] := sql.bind(X_6:int, "sys":str, "r2":str, "z":str, 0:int);
X_80:bat[:int] := algebra.projection(C_77:bat[:oid], X_68:bat[:int]);
(X_81:bat[:oid], C_82:bat[:oid], X_83:bat[:lng]) := group.groupdone(X_80:bat[:int]);
X_84:bat[:int] := algebra.projection(C_82:bat[:oid], X_80:bat[:int]);
X_85:bat[:lng] := aggr.subcount(X_81:bat[:oid], X_81:bat[:oid], C_82:bat[:oid], false:
    bit);
(X_87:bat[:oid], X_88:bat[:oid]) := algebra.join(X_58:bat[:int], X_84:bat[:int], nil:
    BAT, nil:BAT, false:bit, nil:lng);
X_97:bat[:lng] := algebra.projection(X_88:bat[:oid], X_85:bat[:lng]);
X_96:bat[:int] := algebra.projection(X_88:bat[:oid], X_84:bat[:int]);
X_95:bat[:int] := algebra.projection(X_87:bat[:oid], X_58:bat[:int]);
X_94:bat[:int] := algebra.projectionpath(X_87:bat[:oid], X_50:bat[:oid], X_41:bat[:int
    ]);
X_93:bat[:int] := algebra.projectionpath(X_87:bat[:oid], X_49:bat[:oid], C_29:bat[:oid
    ], X_20:bat[:int]);
X_92:bat[:int] := algebra.projectionpath(X_87:bat[:oid], X_49:bat[:oid], X_31:bat[:int
    ]);
sql.resultSet(X_142:bat[:str], X_143:bat[:str], X_145:bat[:str], X_147:bat[:int],
    X_149:bat[:int], X_92:bat[:int], X_93:bat[:int], X_94:bat[:int], X_95:bat[:int],
    X_96:bat[:int], X_97:bat[:lng]);
```

### 2.3.3 Parallelism in MonetDB

To achieve parallelism MonetDB splits the columnar data horizontally and applies the same query plan on each fragment. Parallelization is carried out at MAL level and it is handled by three MAL optimizers. *Mitosis* partitions the data horizontally, *Mergetable* pushes the column fragments through the operators as much as possible without having to recombine them and *Dataflow* identifies independent parts of the MAL program and runs them in parallel. One of the operations that require synchronization and which is often encountered in analytical queries is the Aggregation. A first aggregation step can be run in parallel, where each thread computes the aggregation on its own data, but then the thread local partial results are merged together sequentially. Joins can also be run in parallel, if the two join columns are each split into N horizontal partitions then each thread must perform N joins in which it matches its own left partition with the other threads' N right partitions.

## 2.4 Weld

Weld [14] is a runtime library which improves the performance of data intensive applications by allowing the users to express computation intents throughout a data pipeline specified in Weld's IR, which is in turn compiled into an efficient program. The generated Weld program is compiled and executed only when the computation expression has been completely defined and a materialization of the results is required. Therefore, by having a broad overview of the data flow, several optimization passes can be applied to create an optimized version of the program in terms of memory usage and CPU instructions. An example of how Weld can be used is shown in Figure 2.3. The runtime collects pieces of Weld code from three different library functions and compiles them into an optimized program that can work on the input data.

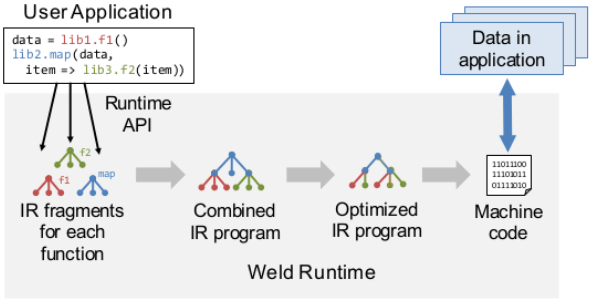


Figure 2.3: Weld collects and combines fragments of IR from different libraries and then produces optimized machine code. Figure 2 from [14].

Weld's IR is constructed around two main concepts: nested parallel for-loops and builders, which indicate what operations are applied on the data. Weld supports basic primitives, such as integers: i8, i16, i32, i64, floats: f32, f64, structures and vectors. Structures are similar to the ones in the C language and follow the same padding rules between structure members. A vector (vec) in Weld is actually just a structure with two members: a pointer to the data C array and an associated length of type i64. Weld also supports type inference; the *?* placeholder can be used to define a builder and the type will be inferred during compilation. At the time of writing there are 5 builders available in Weld, which support the +, *, *min* and *max* merging operations:

appender[T] Creates a new vector by appending values of type T to itself.

merger[T, op] Combines values of type T using the *op* operations. The result is a scalar or a struct of scalars.

dictmerger[K, V, op] Builds a dictionary K -> V and aggregates multiple values associated to the same key using the operator *op*.

groupmerger[K, V] Builds a dictionary K -> vec[V]. Unlike the dictmerger, values are no longer coalesced and are instead added to a vector.

vecmerger[T, op] Combines structures {index:i64, T} into a vec[T] at the given index.

A Weld program consists of functions and expressions. The syntax of a function in Weld is `|a_1: type_1, a_2:type_2, ...| expr_1; expr_2; ...`, in which the function takes a list of typed input parameters a_i:type_i, has several expressions in its body and returns the result obtained by evaluating last expression. As an example, the function `|a:i32, b:i32| let c = a + b; c * c` has a and b of type i32 as input parameters. It will save the sum a + b in the variable c and then return c * c as its result. The function can contain any number of expressions followed by the *;* keyword and a single one at the end, without *;*, which acts as a return statement. New immutable values can be introduced using the *let* statement and blocks of expressions can be contained inside parentheses.

The for-loop in Weld takes three parameters `for(vec, builder, update)` and applies the *update* function on every element of the vector *vec*, possibly merging a value in the builder. At the end of the iterations, the for-loop returns a new builder which incorporates all the updates that were carried out inside the for-loop body. The third parameter in the for-loop, the *update* function, also takes three parameters `|b, i, n|`, where *b* is a builder in which it can merge the result, the iteration number *i* and n represents the value *vec[i]*. The *update* function can either return the result of the merge operation or the unmodified builder.

An important design feature of Weld is that only a single merge operation can be applied on a builder, and every such an operation returns a new builder. Builders are write-only data structures and the *result* operations transforms them into read-only data structures. For example calling *result* on an *appender* will return an immutable vector. Builders are considered linear types, which means that each builder should be used in a linear sequence of operations (several *merge* operations followed by a single *result*). This restriction allows the runtime to apply the *merge* operations on the same memory location instead of forking it, as it would be necessary if a builder would be used twice.

Listing 2.4 shows a Weld program that takes as input a vector of i64 integers. It filters the elements of the vector and appends those that pass the filter into the *appender* builder, and then it computes the cumulative sum of the remaining numbers. At the end, *result* is called on the builder and the final result is returned to the user.

Listing 2.4: An example Weld program which first filters a vector and then computes the sum of the numbers that pass the filter.

```
|x:vec[i64]|
let filtered = for(x, appender[?], |b, i, n|
    if(n < 3L, merge(b, n), b)
);
let sum = for(result(filtered), merger[?], |b, i, n|
    merge(b, n)
);
result(sum)
```

### 2.4.1 Compilation and optimizations

The Weld runtime can perform various optimizations on the given Weld program, such as loop tiling or vectorization, but the optimizer we are most interested in is the loop fusion. There are two types of loop fusion, hereby named vertical and horizontal loop fusion. The former combines two for-loops when the second loop iterates over the result produced by the first, while the latter combines two loops that iterate over the same data, but produce different results. The effects of the optimizer can be seen in Figure 2.4.

The final step in the Weld pipeline is to compile the optimized program. After the Weld code has been optimized, it is transformed into LLVM IR [1] which is then compiled into a binary. The binary is dynamically linked to the main process as a library, and the Weld program can then be executed.

### 2.4.2 Memory management

Weld distinguishes between two types of memory allocations: owned by the user or by the runtime. The user owns the input data and is responsible for bookkeeping it. Any memory that is allocated during the execution of the Weld program is owned by the runtime, including the results. Therefore additional overhead is incurred when copying the result data into an area owned by the user. Ideally we should be

[1] https://llvm.org/docs/LangRef.html

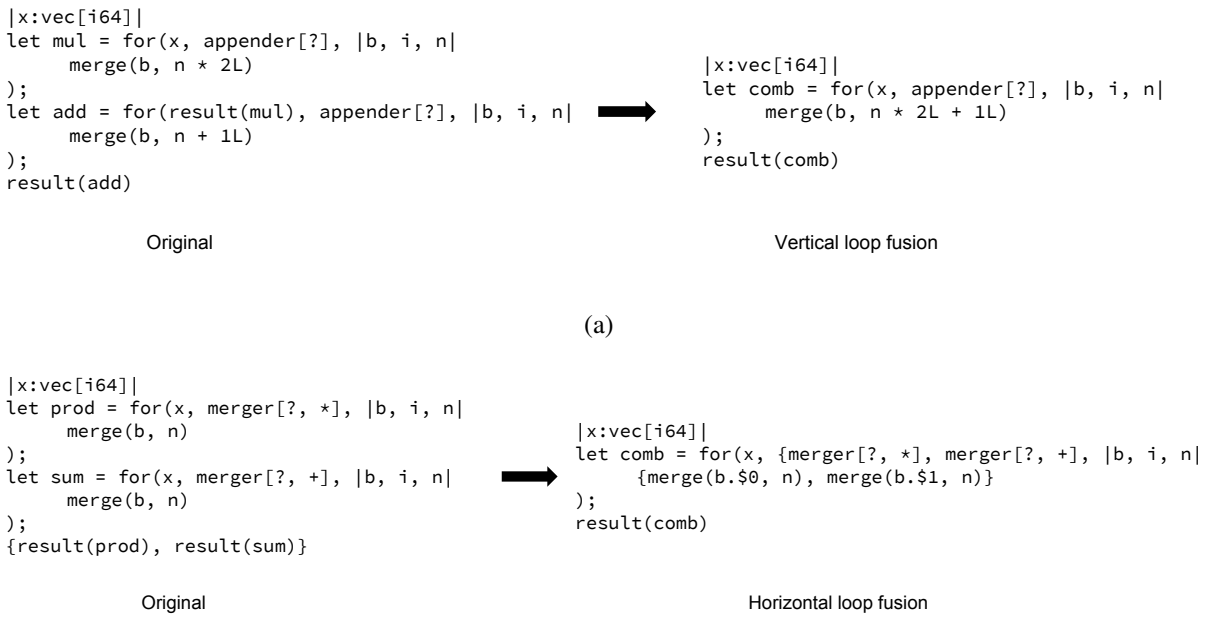


Figure 2.4: Loop fusion in Weld

able to just reference the memory addresses returned in the result, but as Weld does not free any memory during its execution, we need to deallocate everything at the end in order to restore the system to its previous memory state.

# Chapter 3

# Motivation and design

The column-at-a-time execution model from MonetDB solves the main problem of the tuple-at-a-time model by better utilizing the CPU and reducing the interpretation overhead, but it relies heavily on materialization between operators. This excessive materialization is the side effect of the query interpretation being operator-oriented, in that the evaluation is centered around individual operators that exchange information through materialized data. This technique affects the performance of the evaluation, as it becomes limited by the memory bandwidth

We presented the architecture of MonetDB's query engine in Figure 2.1 and we saw that SQL queries are expressed internally first as a relational algebraic tree and then as a sequence of MAL instructions. On both representations optimizations are applied in certain stages in the interpretation pipeline. As the execution is columnar, the input and output of each MAL operator consists of materialized columns. For example, an expression such as `a * (b + c)`, in which two columns are added and subsequently multiplied with a third, will become two MAL operators: the addition `b + c` will be stored into a result (a column), which will then be passed to the multiplication with `a`.

Boncz et al. show in [6] that the amount of materialization has a significant impact on MonetDB's performance. In this thesis we aim to reduce the materialization and thus improve the query execution time by introducing JIT compilation of (or parts of) the query to native machine code into MonetDB's interpretation pipeline. In this way, we hope to bring the performance closer to the the hardware limits. A JIT compiled query will benefit from a data-centric evaluation, as the boundaries between operators are removed.

To this end, we identify two modules in Figure 2.1 that can act individually as the entry point of the added JIT logic: the MAL generator, a module that emits MAL instructions based on the algebraic tree and the MAL optimizer, which creates an optimal set of MAL instructions right before the MAL interpreter calls into the GDK kernel for their interpretation. Either one of these modules can become the incision point into the interpretation pipeline and we will use them in two different approaches to JIT integration. Our first approach will be to change the MAL optimizer's workflow to create new MAL instructions that emit Weld code instead of using the default instruction implementation from GDK. We will henceforth denote this process as the *MAL-Weld* method. The second approach is to replace the MAL instructions created by the MAL generator with Weld code. We will refer to this method as the *REL-Weld* implementation. The two approaches are depicted in Figure 1.1, side-by-side with the canonical MonetDB interpretation pipeline.

The following sections will briefly explain the design of the two approaches and in Section 3.3 we pose our research questions.

## 3.1 MAL-Weld

For a given query the MAL program expresses the computation intent in terms of BAT operators. The MAL instructions are usually function calls that map directly to a BAT operator implementation in the GDK kernel. Conceptually, the MAL instructions are chained together by variables, instructions take input parameters and return their results in one or more variables which will act as the input to other instructions. As a consequence of the interpreted operator-centric column-at-a-time execution model, the results return by MAL instructions are always materialized BATs.

Our approach to solve this problem is to translate the GDK implementations of BAT operators into Weld's IR. The idea is to produce snippets of Weld statements, that correspond to the involved MAL instruction, so that in the end we obtain a coherent Weld program that preserves the semantics of the MAL program. Most BAT operators perform simple operations over one or two columns and usually involve a single for-loop, so the Weld translation will as well mostly consist of a for-loop. We then rely on Weld's compiler to identify chained operators and apply loop fusion to combine multiple for-loops into a single one and thus reduce the amount of materialization.

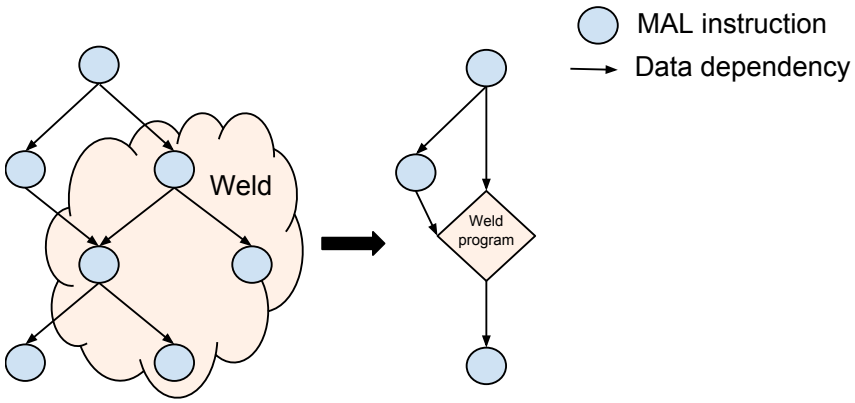


Figure 3.1: The MAL program is a directed graph, in which vertices are MAL instructions and the edges data dependencies between them. Groups of Weld-compatible MAL instructions that form a sub-graph can be pulled out into a single node, which generates Weld code, compiles it to machine code and runs it. The input and output of this node are BATs, therefore it can be seamlessly integrated into the existing instruction graph.

An important observation is that, as the translation is performed MAL instruction by MAL instruction, we can choose which instructions to translate and for which preserve the original GDK call. This is important because certain MAL instructions are highly particular or optimized, so they may not have a Weld equivalent. Moreover, sometimes it can be useful for us to delegate a task to the GDK kernel, if it can be solved better and faster through the native call. The MAL program can be viewed as a directed graph where the nodes are the instructions and the edges are the data dependencies between them. In our translation we identify which MAL instructions can be translated to Weld and then rearrange the MAL program so that translatable instructions form a subgraph that can be collapsed into a single single node without causing a data dependency cycle. Each Weld-able subgraph will result in a complete Weld program which is constructed by the individual nodes that belong to it. As there is a large number of MAL instructions, this approach allows us to have a modular integration of Weld into MonetDB which can operate seamlessly with the standard MAL instructions. Figure 3.1 depicts the described process.

## 3.2 REL-Weld

For a given query the relational algebra offers a broader view over the data flow than the generated MAL program. Our second approach to improve the query evaluation time in MonetDB is to completely replace the column-at-a-time execution model with the produce-consume model described in Section 2.2. The relational algebra is represented as a tree, in which the internal nodes are the operators and the leaves are the relations. Using the produce-consume model we can generate a Weld program based on the relational algebra that contains the entire query evaluation logic. In [12] Neumann categorizes certain operators as being pipeline breakers - they either remove the data from the CPU cache or, in the worst case, require materializing the input. The goal is to determine the operator pipelines in the algebraic tree and to fuse them together. The fused sequence becomes one or several nested for-loops in which multiple operations are performed on a tuple in a pipelined fashion.

The MAL program is generated from the relational algebra as well, so in the MAL-Weld translation presented above we are basically asking Weld to recreate the algebra through loop fusion. Due to the nature of the MAL program and the intermediate and auxiliary results that MAL instructions produce we do not expect Weld to be able to identify all the fusion opportunities, which is why we believe that the REL-Weld approach will generate a better query plan.

## 3.3 Research questions

MonetDB solves the problem of high interpretation overhead through its column-at-a-time approach by increasing the ratio of effective data computations over the interpretation cost. By doing so, MonetDB materializes intermediate results between operators and thus the performance becomes limited by the memory bandwidth. The research questions we are trying to answer in our work are:

- Can the query compilation into native machine code reduce the amount of intermediate materialization by removing the boundaries between operators and fusing them together?
- Is Weld able to perform loop fusion given the data flow generated by the MAL program? If not, can we add new optimization rules to Weld?
- Can we express all relevant operations using Weld's IR and parallel builders?
- Is the data-centric compilation approach in REL-Weld superior to MAL-Weld?

# Chapter 4

# Implementation

We will now discuss the technical aspects of our Weld integration into MonetDB.

## 4.1 MAL-Weld

The first approach to integrate Weld into MonetDB happens at the MAL level. A MAL program consists of multiple MAL instructions that together express the data flow. Given Weld's ability to optimize the data flow we see the MAL instructions at the core of our translation to Weld. Throughout this section we are going to discuss the implementation of some of the most common operators and their MAL instructions.

The MAL program is generated from the optimized relational algebra and then it passes through another series of (MAL) optimizers. To identify the instructions that can be translated to Weld we implemented a new optimizer that modifies and reorganizes the MAL program. Our component is part of a new optimizer pipeline in which we disabled the *dataflow* and *mitosis* modules that enable parallelism in MonetDB.

The first step in the Weld optimizer is to build an instruction dependency graph. Then, while maintaining a list of Weld translatable instructions, we identify subgraphs that contain only such instructions. The end result is a MAL program where the Weld instructions belonging to the same subgraph are grouped together and surrounded by standard GDK backed MAL instructions that connect those subgraphs. Listing 4.1 shows how our Weld optimizer identifies two subgraphs that are connected by an instruction which does not have a Weld implementation (*algebra.likeselect*). A MAL instruction that can be implemented in Weld is replaced with another MAL instruction from the *weld* module so that at run time, instead of calling a GDK function, the new instruction will generate Weld code. We kept the new instruction's name and signature unchanged so that it is easy to understand a MAL-Weld program.For example, *batcalc.add* becomes *weld.batcalcadd*. As the MAL program is executed, we want each Weld instruction to append its part of the program to a shared buffer. For that, we initialize a buffer called *wstate* and we pass a pointer to it to every Weld instruction belonging to the same subgraph. When all the MAL instruction from the subgraph have been executed, meaning that the Weld program is complete, the *weld.run* instruction compiles the program, runs it on the input that and sets the results in the corresponding MAL variables.

Listing 4.1: Part of an original MAL program and the one produced by the Weld optimizer

```
/* Original MAL program */
X_2:bat[:int] := sql.bind(X_1:int, "sys":str, "part":str, "p_partkey":str, 0:int);
X_3:bat[:str] := sql.bind(X_1:int, "sys":str, "part":str, "p_mfgr":str, 0:int);
C_4:bat[:oid] := sql.tid(X_1:int, "sys":str, "part":str);
```

```
X_5:bat[:oid] := algebra.thetaselect(X_2:bat[:int], C_4:bat[:oid], 15:int, "==":str);
X_6:bat[:oid] := algebra.likeselect(X_3:bat[:str], X_5:bat[:oid], "%BRASS":str, "":str
    , false:bit);
X_7:bat[:int] := algebra.projection(X_6:bat[:oid], X_2:bat[:int]);
X_8:bat[:int] := batcalc.add(X_7:bat[:int], 100:int);

/* The MAL program after the Weld optimizer */
X_2:bat[:int] := sql.bind(X_1:int, "sys":str, "part":str, "p_partkey":str, 0:int);
X_3:bat[:str] := sql.bind(X_1:int, "sys":str, "part":str, "p_mfgr":str, 0:int);
C_4:bat[:oid] := sql.tid(X_1:int, "sys":str, "part":str);
wstate:ptr := weld.initstate();
X_5:bat[:oid] := weld.algebrathetaselect(X_2:bat[:int], C_4:bat[:oid], 15:int, "==":
    str, wstate:ptr);
X_5:bat[:oid] := weld.run(wstate:ptr, X_2:bat[:int], C_4:bat[:oid], 15:int, "==":str);
X_6:bat[:oid] := algebra.likeselect(X_3:bat[:str], X_5:bat[:oid], "%BRASS":str, "":str
    , false:bit);
wstate:ptr := weld.initstate();
X_7:bat[:int] := weld.algebraprojection(X_6:bat[:oid], X_2:bat[:int], wstate:ptr);
X_8:bat[:int] := weld.batcalc.add(X_7:bat[:int], 100:int, wstate:ptr);
X_8:bat[:int] := weld.run(wstate:ptr, X_6:bat[:oid], X_2:bat[:int], X_7:bat[:int],
    100:int);
```

In the remainder of this section we will show the Weld translation of several MAL instructions that belong to one of the canonical algebraic operators.

### 4.1.1 Select

There are two types of MAL Select instructions, *algebra.select* in which the data is fitted inside or outside a range of values as dictated by the combination of *li* and *hi* parameter flags, and *algebra.thetaselect* which selects the data values that satisfy a relation *dataValue OP VAL*. The implementation of these operation is a simple *for-loop* statement , as shown in Listing 4.2, and we expect that Weld's loop fusion optimization pass will pick up these instructions and coalesce them into a single loop.

Listing 4.2: algebra.select and algebra.thetaselect MAL instructions and the Weld code

```
algebra.select(col:bat[:any_1], low:any_1, high:any_1, li:bit, hi:bit, anti:bit):bat[:
    oid]
algebra.select(col:bat[:any_1], candid:bat[:oid], low:any_1, high:any_1, li:bit, hi:
    bit, anti:bit):bat[:oid]

/* Example Weld code for algebra.select with a candidate list */
let v1 = result(
    for(candid, appender[i64], |b, i, oid|
        if (low < lookup(b, oid) && high >= lookup(b, oid),
            merge(b, oid),
            b
        )
    )
);

algebra.thetaselect(col:bat[:any_1], val:any_1, op:str):bat[:oid]
algebra.thetaselect(col:bat[:any_1], candid:bat[:oid], val:any_1, op:str):bat[:oid]
/* Example Weld code for algebra.thetaselect without a candidate list */
let v1 = result(
    for(col, appender[i64], |b, i, n|
        if (n == val,
            merge(b, i),
```

```
            b
        )
    )
);
```

### 4.1.2 Project

Project operations usually produce a column that is the result of applying an operation on several input columns. Most projections are unary or binary operations and can take a column, two columns or a column and a constant value as arguments. There are also more complex projections such as *batcalc.ifthenelse* which is the equivalent to the conditional ternary operator *cond ? then : else*. Besides these operations, MAL also has an instruction called *algebra.projection* that, using a candidate list, extracts data values from a column. This is usually the last operation in a series of other instructions that work on candidate lists. Again, we expect that the for-loops generated by projections to be successfully picked up by the Weld optimizer. Two examples of project operations can be seen in Listing 4.3.

Listing 4.3: algebra.projection and algebra.+ MAL instructions and the Weld code

```
batcalc.+(colA:bat[:lng], colB:bat[:lng]):bat[:dbl]
/* Example Weld code for algebra.+ without a candidate list */
let v1 = result(
    for(zip(colA, colB), appender[i64], |b, i, n|
        merge(b, n.$0 + n.$1)
    )
);

algebra.projection(candid:bat[:oid], col:bat[:any_1]):bat[:any_1]
/* Weld code for algebra.projection */
let v1 = result(
    for(candid, appender[?], |b, i, oid|
        merge(b, lookup(col, oid))
    )
);
```

### 4.1.3 Aggregation

In MonetDB complex relational algebra operators are split into multiple simple instructions and the Aggregation is one of those operations. MAL instructions can take a variable number of arguments and also return a variable number of results, but as MonetDB aims to process columns in a tight for-loop, such instructions have to be split into smaller instructions with a fixed number of parameters for which there exists an optimized implementation in GDK. The Aggregation is split into two phases: MonetDB's approach is to first determine the unique groups in the input, assign a groupID to each row and then compute the aggregates based on the groupIDs. For example, let's look at the query `SELECT SUM(d) , SUM(e)FROM table GROUP BY a, b, c`. The core generated MAL program is shown in listing 4.4 (the instructions' signature will be explained below). The Aggregation operation can have N key and M aggregation columns. The query has N = 3 grouping instructions and M = 2 aggregations. We can observe that the groupIDs (groupIDsC and groupIDsCD) are "artificial" intermediate results, these BATs are produced only to be used during the aggregations' computation and are not part of the final result.

Listing 4.4: MAL code generated for `SELECT SUM(d), SUM(e)FROM table GROUP BY a, b, c`

```
(groupIDsA, extentsA, histoA) = group.group(colA);
```

```
(groupIDsAB, extentsAB, histoAB) = group.subgroup(colB, groupIDsA);
(groupIDsABC, extentsABC, histoABC) = group.subgroup(colC, groupIDsAB);
colDSums = aggr.subsum(colD, groupIDsABC, extentsABC, true, true);
colESums = aggr.subsum(colE, groupIDsABC, extentsABC, true, true);
```

Ideally we would want Weld to transform the instructions from Listing 4.4 into a single for-loop which has no intermediate results. In the Weld syntax it should look looks like *for(zip(a, b, c, d, e), dictmerger[?, ?, ?, ?, +] ....* Unfortunately at the time of writing Weld does not have an optimizer which can detect that the groupIDs produced by the *group.*group* instructions represent unique identifiers for each group, and that grouping by the groupIDs would be equivalent to grouping by the data tuples. The groupIDs are a functional dependency of the aggregation columns, each tuple from the input columns is associated with exactly one groupID. In other words, Weld does not track functional dependencies. The problem is best described in Figure 4.1 where we look at a more simple query: `SELECT SUM(B)FROM R GROUP BY A`. In this aggregation we have two MAL instructions *group.group* which determines the *groupIDs* based on column *A* and *aggr.subsum* which computes the aggregated sum of *B* by summing each value in the bucket indicated by the *groupID*. By translating individual MAL instructions to Weld, we need to reproduce the output of the MAL instructions in the Weld code. In the right side of the Figure 4.1 we have the equivalent Weld code for the two instructions, and there is no Weld optimizer that can fuse the three for-loops.

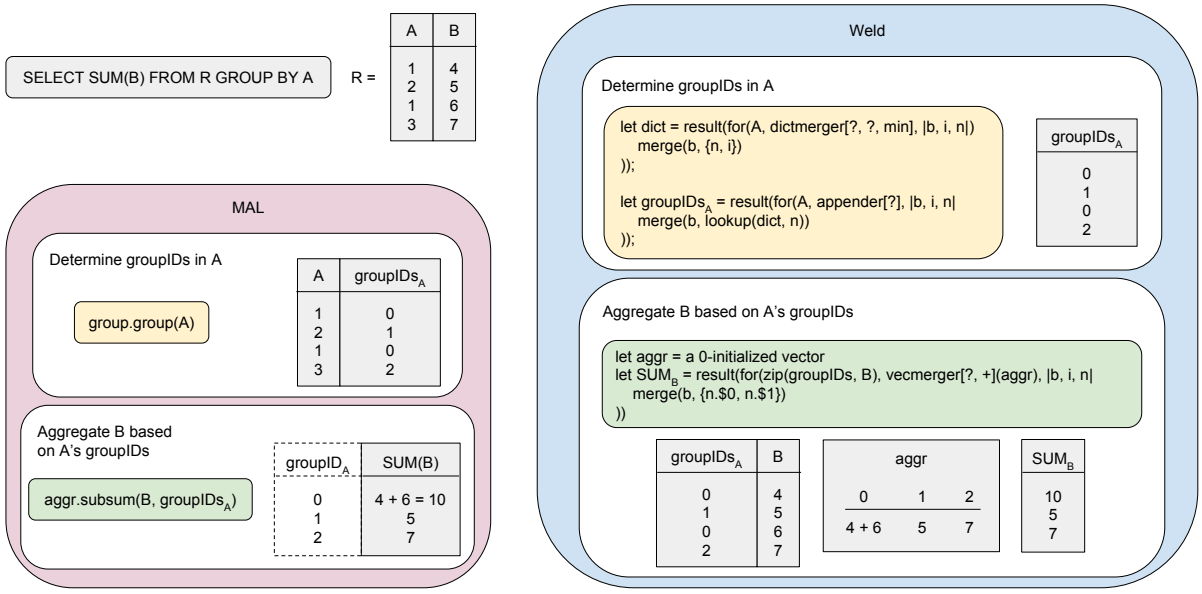


Figure 4.1: Two MAL instructions which are using during an Aggregation in MonetDB and their Weld implementations. The generated Weld code cannot be optimized by the runtime.

In order to bypass this limitation of the optimizer, we attempt to implement the operation in a way that slightly breaks the one-instruction-at-a-time MAL-Weld translation. We notice that each *group.subgroup* MAL instruction builds on top of a previous grouping instruction: *group.subgroup(col, oldGroupdIDs)*. Therefore when generating the new set of groupIDs, instead of using the current column and the oldGroupIDs, we can use the current column and all the other columns that have been used to generate oldGroupIDs. The second and third grouping operations from the example above now become become *group.subgroup(colB, colA)* and *group.subgroup(colC, colB, colA)*. The advantage of this approach is that we construct a single hash table and we reduce the number of intermediate materializations. Furthermore, Weld is now able to tell that only the final groupIDs (*groupIDsABC* and *extentsABC*) are used in the aggregation which means that the first two grouping operations are now redundant and can be removed from the optimized program. A further optimization in this direction, which we did not implement, would be to also keep track of the aggregation MAL instructions which use the result of the

grouping phase and manually generate the optimally fused code.

The groupIDs in our GroupBy Weld implementation are consecutive values ranging from 0 to the total number of groups. In Weld it is not possible to access an incomplete data structure, which means that while building the groupIDs dictionary, we cannot probe it to check whether the key already exists or whether we have encountered a new key for which we need to generate a new unique groupID. As a consequence, we first generate a set of unique group identifiers using a *dictmereger* and the *min* operator to create a mapping between a key (a group) and a groupID (the smallest row number at which we encounter this key). We then use the row indexes to create a new set of groupIDs that are consecutive and start from 0. The final step is then to produce the three result columns containing the groupID for each row, indexes for data representatives (an index in the input columns to show what the value looks like) and counts for each group. The aggregation operations use the groupIDs to determine the aggregation bucket. Because the groupIDs are 0 indexed consecutive values we can now use Weld's *vecmerger* instead of the hash map backed *dictmeger*.

The Aggregation MAL instructions and the complete generated Weld code for them can be seen in Listing 4.5.

### Aggregations not optimized by Weld

The GroupBy MAL instructions explicitly create a functional dependency towards the groupIDs column. Weld is currently lacking an optimizer that detects the fact that an aggregation by a column which is a functional dependency could be rewritten into an aggregation by the original columns. This optimization pass in combination with the horizontal loop-fusing one could rewrite the aggregation operation using a single for-loop and a single dictionary.

Listing 4.5: group.group, group.subgroup and aggr.subsum MAL instructions and the Weld code

```
group.group(col:bat[:any_1]) (groups:bat[:oid], extents:bat[:oid], histo:bat[:lng])
group.subgroup(col:bat[:any_1], g:bat[:oid]) (groups:bat[:oid], extents:bat[:oid],
    histo:bat[:lng])

/* Example Weld code for group.subgroup with three columns */
/* Dictionary key -> min row index
let idxMap = result(
    for(zip(colA, colB, colC), dictmerger[{?, ?, ?}, i64, min], |b, i, n|
        merge(b, {n, i})
    )
);
let idxVec = tovec(idxMap);
/* Dictionary key -> groupID
let groupIDMap = result(
    for(idxVec, dictmerger[{?, ?, ?}, i64, min], |b, i, n|
        merge(b, {n.$0, i})
    )
);
/* Vector to be used with vecmerger */
let zeros = result(
    for(rangeiter(0L, len(idxVec), 1L), appender[i64], |b, i, n|
        merge(b, 0L)
    )
);
let groups = result(
    for(zip(colA, colB, colC), appender[i64], |b, i, n|
        let groupID = lookup(groupIDMap, n);
        merge(b, groupID)
```

```
    )
);
/* We compute the counts in a separate loop because the counts are actually rarely
 * used, so Weld will remove this computation */
let histo = result(
    for(zip(colA, colB, colC), vecmerger[i64, +](zeros), |b, i, n|
        let groupID = lookup(groupIDMap, n);
        merge(b, {groupID, 1L})
    )
);
let extents = result(
    for(idxVec, vecmerger[i64, +](empty), |b, i, n|
        merge(b, {i, lookup(idxMap, n.$0)})
    )
);

aggr.subsum(col:bat[:lng], groups:bat[:oid], extents:bat[:any_1], skip_nils:bit,
    abort_on_error:bit):bat[:lng]
aggr.subsum(col:bat[:lng], groups:bat[:oid], extents:bat[:any_1], candid:bat[oid],
    skip_nils:bit, abort_on_error:bit):bat[:lng]

/* Example Weld code for aggr.subsum without candidate list */
let zeros = result(
    for(rangeiter(0L, len(col), 1L), appender[?], |b, i, n|
        merge(b, 0?)
    )
);
let v1 = result(
    for(zip(groups, col), vecmerger[?, +](zeros), |b, i, n|
        merge(b, {n.$0, n.$1})
    )
);
```

### 4.1.4 Join

The join MAL instruction in Listing 4.6 takes as parameters two column, two optional candidate lists and two other optional arguments and returns two columns *X_0* and *X_1* which contain the join matches. The hash join algorithm can be implemented in Weld using the *groupmerger* builder, which is a dictionary that holds of list of values for any given key, or the more lightweight *dictmerger* if we know that one of the columns only contains unique values. Unfortunately during the implementation we ran into a Weld bug [1] which forced us to delegate the join operation to the default implementation in GKD.

Listing 4.6: MAL join instruction

```
algebra.join(left:bat[:any_1], right:bat[:any_1], leftcand:bat[:oid], rightcand:bat[:
    oid], nil_matches:bit, estimate:lng) (X_0:bat[:oid], X_1:bat[:oid])
```

### 4.1.5 Summary

In this section we presented the one-instruction-at-a-time translation from MAL to Weld. In order to beat MonetDB's performance we rely on the Weld optimization passes to perform loop fusion and eliminate

[1] https://github.com/weld-project/weld/issues/363

as much intermediate materialization as possible. We have seen how the Selection and Projection operators can be successfully picked up the Weld's optimizers, while the Aggregations can only be partially improved and the Joins cannot yet be implemented.

## 4.2 REL-Weld

The second approach to adding just-in-time compilation in MonetDB is to construct a Weld program based on the relational algebra, using the produce-consume model described by Neumann in [12]. The idea behind this technique is to identify pipelines (chains of operators that do not materialize the result) and push tuples through them. In this way a tuple can remain in the CPU registers while several operators are applied on it, which leads to an optimal execution plan.

Weld's IR is quite expressive and allows us to generate code that is both efficient and easy to read and understand. The query in Figure 2.2 is the one used by Neumann in [12]. Similar to the referenced paper, we show its representation in the relational algebra in Figure 4.2, but adapted to MonetDB's translation. The Figure shows how the plan is also split into 4 pipeline. The pipeline breakers in our example are the Group By operation and the two materializations from the Joins.

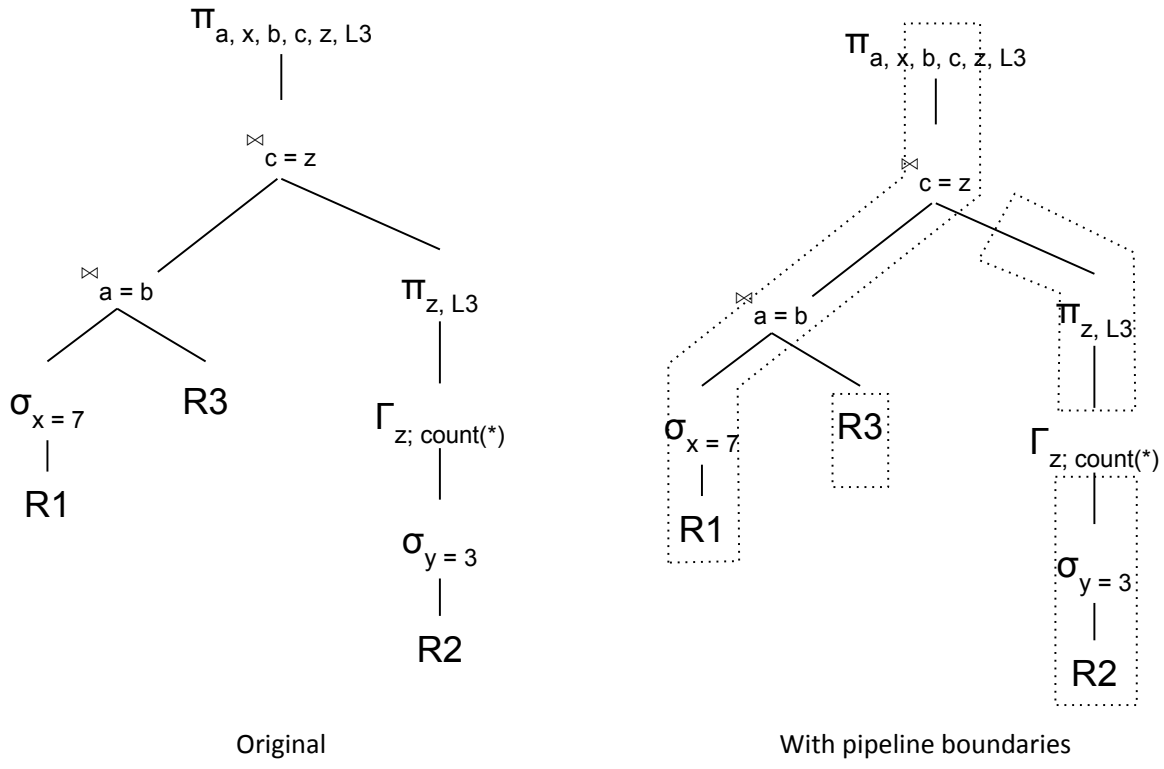


Figure 4.2: Query plan for Figure 2.2

Listing 4.7 shows the generated Weld code for the query in Figure 2.2. The program contains 6 for-loops; the first loop constructs the Group By hash table, the next two materialize the tuples in the Joins' hash tables while the last 3 loops iterate over the R1 column and probe the join hash tables.

In MonetDB, expressions (the sql_exp structure) are at the core of the relational algebraic operators and encode the information needed to process the operator. As expressions are the base type for all algebraic operators, they do not necessarily operate on columns, but on literals as well. In order to handle operations involving complex literals without rewriting the code already available in GDK, we

Table 4.1: The produce-consume implementation of the most common algebraic operators

| | | |
|---|---|---|
| Scan | produce | start the for-loop: **for(zip(...** |
| Select | produce | call input.produce |
| | consume | add the filters: **if(condition, ..., ....** |
| Project | produce | if sort:<br>new_builder = **{appender[?], ...}**<br>call input.produce |
| | consume | if need sorting:<br>materialize arrays: **{merge(vec1, x), merge(vec2, y), ...**<br>sort the arrays: **sort(vecs, fn)**<br>start a new for-loop: **for(zip(...**<br>apply projection operations or rename: **let x = y + z;** |
| Group By | produce | new_builder = **dictmerger[..., ...., op]**<br>call input.produce |
| | consume | materialize: **merge(key, value)**<br>start a new for-loop: **for(tovec(dict), ...** |
| Join | produce | new_builder = **groupmerger[..., ...]**<br>call right.produce |
| | consume right | materialize hashtable: **merge(key, value)**<br>call left.produce |
| | consume left | start a new for-loop: **for(lookup(hashtable, key), ...** |

offload the handling of the column-less expressions to MonetDB and pass the result as a parameter to our Weld program, for example, a date will be translated into an integer by MonetDB. An expression in the relational algebra provides us with data types and column names. In our Weld translation we make use of the types to generate builders typed correctly and we name the variables using a pattern known by all operators such that a block of Weld code generated by multiple translation nodes remains semantically correct. For example, we see how in Listing 4.7 the Scan operator generates a for-loop and then a Select operator can apply a filtering predicate on the variables created by the scan.

In MonetDB string columns are backed by two arrays, one containing the actual strings and the other indexes in the strings array. If there are any string columns in the output column set, we need our Weld program to also return pointers to both arrays in order to reconstruct a BAT. Therefore extra care is needed to rename and reference the string columns in the Weld program correctly.

The algorithm that builds the Weld program works on the relational algebra tree. It starts at the root node, it visits every node in the tree and then it traces back to the root operator. A node will usually generate code or initialize state in the *produce* phase, will call *child.produce*, and when the function call ends, the *consume* phase begins and more Weld statements are produced. The code is appended to a shared buffer and every operator expects that the ancestors and the descendant nodes will leave the program in a coherent state. In the following subsections we will discuss the implementation of the main relational algebra operators and show which parts they contributed with to the final Weld code in listing 4.7.

Listing 4.7: Generated Weld for query 2.2

```
1 let v0 = (
2     let v1 = result(
3         let v2 = result(
4             for(zip(r2y, r2z), dictmerger[i32, i64, +], |b1, i_1, n1|
5                 let r2_y = n1.$0;
6                 let r2_z = n1.$1;
7                 if((r2_y == 3) == false,
8                     b1,
```

```
9                   merge(b1, {r2_z, 1L})
10              )
11          )
12      );
13      for(tovec(v2), groupmerger[i32, {i32, i64}], |b1, i_1, n1|
14          let r2_z = n1.$0;
15          let L3_L3 = n1.$1;
16          let r2_L3 = L3_L3;
17          merge(b1, {r2_z, {r2_z, r2_L3}})
18      )
19  );
20  let v3 = result(
21      for(zip(R3b, R3c), groupmerger[i32, {i32, i32}], |b1, i_1, n1|
22          let r3_b = n1.$0;
23          let r3_c = n1.$1;
24          merge(b1, {r3_b, {r3_b, r3_c}})
25      )
26  );
27  for(zip(R1a, R1x), {appender[i32], appender[i32], appender[i32], appender[i32],
        appender[i32], appender[i32], appender[i64]}, |b1, i_1, n1|
28      let r1_a = n1.$0;
29      let r1_x = n1.$1;
30      if((r1_x == 7) == false,
31          b1,
32          for(lookup(v3, r1_a), b1, |b2, i_2, n2|
33              let r3_b = n2.$0;
34              let r3_c = n2.$1;
35              for(lookup(v1, r3_c), b2, |b3, i_3, n3|
36                  let r2_z = n3.$0;
37                  let r2_L3 = n3.$1;
38                  {merge(b3.$0, r1_a), merge(b3.$1, r1_x), merge(b3.$2, r3_b), merge(
                        b3.$3, r3_c),
39                      merge(b3.$4, r2_z), merge(b3.$5, r2_L3)}
40              )
41          )
42      )
43  )
44  );
45  {result(v0.$0), result(v0.$1), result(v0.$2), result(v0.$3), result(v0.$4), result(v0.
        $5)}
```

### 4.2.1 Scan

The Scan, or Table, as named in the MonetDB algebra, operator is not a canonical algebraic operator, but is nonetheless part of the relational algebra produced by MonetDB. It can only be a leaf in the query plan, and every leaf of the plan must be a scan. Scans used to generate MAL instructions for reading BATs into memory. In our Weld translation we keep the BAT-reading MAL instructions as they are, and use their result as input to our Weld translation of the Scan, where we generate a for-loop to iterate over the data fetched with GDK. This for-loop uses a builder received from an ancestor node and also creates loop variables which are named by a pattern known to all the nodes such that these variables can be directly referenced. As the for-loop statement is produced, control is returned to the parent, which continues to build its own statements. Lines 4-6, 21-23 and 27-29 from Listing 4.7 are produced by the Scan operator.

### 4.2.2 Select

In relational algebra the Select operator filters the incoming tuples based on a series of predicates. At this step it is decided which tuples are pushed forward in the pipeline. The output consists of an *if* statement with several chained conditions. The Weld syntax for the *if* statement is *if(cond, then, else)* and the *else* block usually only contains the unmodified builder. Therefore in order to allow other operators to continue to build the program, we reverse the condition so that the *then* block is at the end, and the rest of the program can be appended to the share buffer: *if(cond == false, unmodifiedBuilder, ....* Lines 7 and 30 from Listing 4.7 are produced by Select.

### 4.2.3 Project

The Project operator is used to apply an operation on a column or to rename it in the consume phase. Weld supports basic arithmetic operations which are enough for the most common queries. Other more complex operations can be applied through UDFs, which have to be declared beforehand in our implementation. Line 16 from Listing 4.7 is produced by Select.

In MonetDB Project can also function as an Order By operator. So while usually the Project operation is not a pipeline breaker, if the Order By functionality is activated then the data has to be materialized, sorted and the pipeline is resumed by generating a new for-loop over the sorted columns. The Order By operation is performed in the root node of the relational algebra, therefore materializing at this point does not incur any overhead as it is also the final step in the pipeline.

### 4.2.4 Aggregation

Aggregation (Group By) is a pipeline breaking operation which requires the materialization of input tuples in order to produce its output. The operation involves aggregating a set of columns while grouping by a key determined by another set of columns. Fortunately, Weld provides us with the *dictmerger* builder which can handle grouping and the aggregations at the same time. The first step in the operator's produce phase is to generate a new builder which is passed to the child operator. Then, the consume phase means actually aggregating the tuples in the dictionary (merge(dict, key, value)) and then resuming the pipeline by using the old builder to emit a new for-loop over the finalized dictionary. The *dictmerger* generated in the produce phase can be seen on line 4 in Listing 4.7, while the consume phase generated lines 9-15.

### 4.2.5 Join

This subsection covers the Join, Semijoin and Antijoin operations as their implementations are almost identical. The join algorithm that we implemented in Weld is the hash join and it is backed by Weld's *groupmerger* builder. The algorithm consists of building a dictionary on the right hand side relation and then using the left one to probe the dictionary and emit new tuples when there is a match.

In the produce phase we generate a *groupmerger* builder where the key is the join attribute and the value is a list of matches that belong to the same group. Lines 13 and 21 show such a builder. The builder is passed to the *right.produce* function and is used to begin a new pipeline that ends with the materialization of the tuples in the hash table. In the consume phase associated with the right relation we add the tuples in the dictionary and we materialize it, as seen on lines 17 and 24. After we call the produce function on the left operator, the second consume phase begins and we probe the hash table using tuples from the left relation. The code produced will result in a nested for-loop over the matches in the dictionary, as shown on lines 32-37. Semijoin and Anti join work in a similar way, except that the former can produce 0 or

1 matches and the latter will emit tuples when there are actually no matches. To speedup the execution of Semijoins and Anti joins we use a *dictmerger* instead of a *groupmerger* as we are not interested in the match itself, but in whether there is one or not.

In practice we observed that the performance of the *groupmerger* builder is far worse than that of the *dictmerger*. A difference in performance is expected as the *dictmerger* dictionary holds a single value for a given key whereas *groupmerger* maintains a list of values associated to a key. Therefore, if we know that during a join operation one of the join key columns contains unique values we might swap the left and the right join side and always build the hashtable on that column. This process will thereby transform the Left-deep join into a Bushy join and introduce more materialization. Figure 4.3 shows the difference between the Left-deep Join that would result from the pipeline model and the Bushy Join in which the result of a Join is materialized into a hashtable. In the TPC-H benchmark, on which we test our implementation, the foreign key constraints are part of the specification, which makes this optimization an important aspect of our Weld translation. In this new join process we also need to separately materialize the other columns from the side on which the hash table is built on, which can result in slightly improved memory utilization as consecutive values of the same type can be packed more densely in memory. Using the *dictmerger* builder for the joins in Query 3 from TPC-H resulted in a speedup of over 100.

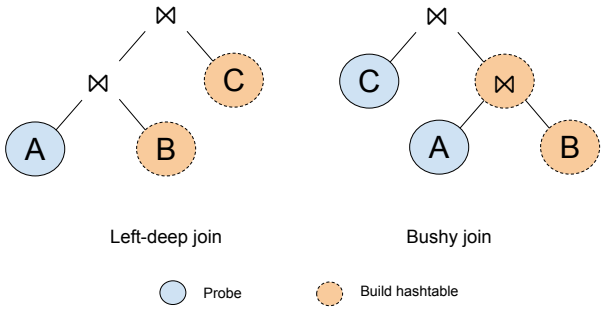


Figure 4.3: Left-deep join vs Bushy join

### 4.2.6 TopN

The TopN operator limits the number of rows that are returned by the query. If present, it is always at the root of the tree where we can just return a slice of the resulting vectors.

### 4.2.7 Summary

In this section we presented our implementation of the produce-consume query evaluation model using Weld. Weld's IR is very expressive and has allowed us to easily implement the query evaluation logic.

# Chapter 5

# Evaluation

In this chapter we evaluate the two implementations of our Weld integration into MonetDB: relational algebra to Weld (REL-Weld) and MAL to Weld (MAL-Weld). We will compare the performance of the two approaches against MonetDB and show where Weld can outperform MonetDB and where Weld lags behind. We ran our experiments on a Intel Xeon E5-2650 CPU with 256 GB RAM, 32 cores running at 2.8 GHz and a shared L3 cache of 20MB. We used MonetDB 11.29 and Weld 0.2.

To showcase the performance of the Weld translation we use the TPC-H benchmark. TPC-H contains a set of 22 ad-hoc SQL queries that are considered to be a good workload representation of a business oriented application. We ran the queries on different scale factors between 1 and 100, where a scale factor of 1 corresponds to a overall database size of 1GB. Our Weld translations does not yet work for all the 22 queries, and at the time of writing Weld does not support sorting by composite keys (structs) and as a consequence we removed the *ORDER BY* clause from all the queries. We ran each query three times and we recorded the best (smallest) run time.

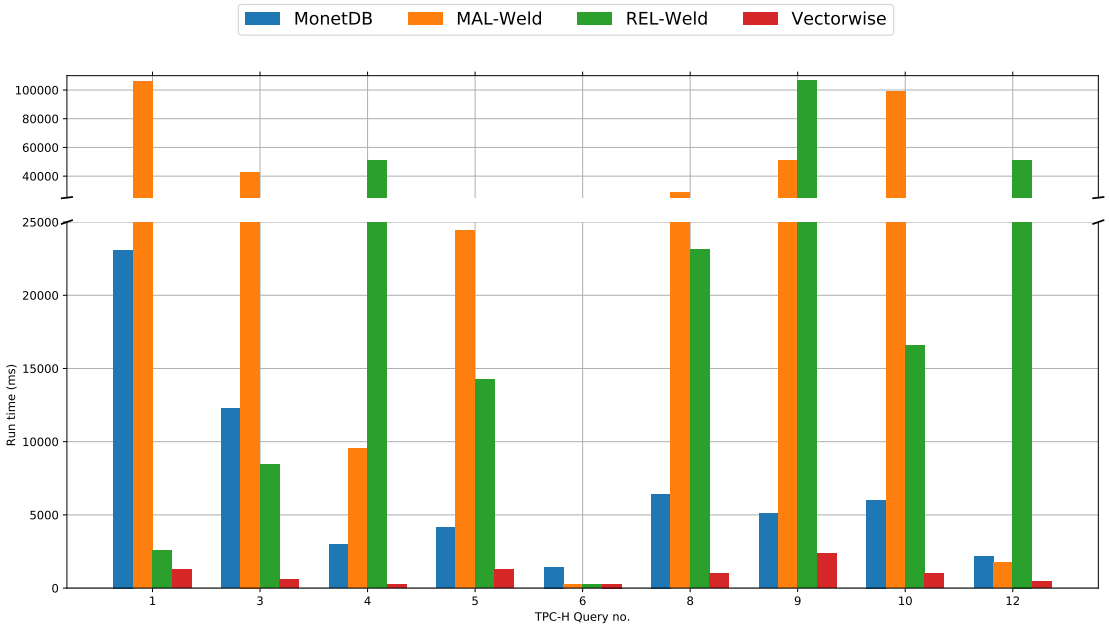


Figure 5.1: MonetDB vs MAL-Weld vs REL-Weld vs Vectorwise on SF-100 and 32 threads.

Figure 5.1 shows the runtime of 10 of the 22 TPC-H queries. We compare the execution times recorded from MonetDB, MAL-Weld, REL-Weld and Vectorwise. Vectorwise held the TPC-H benchmarking record [19] and acts here as a reference for our Weld integrations. We can observe that for most of the

queries the Weld translations are slower than MonetDB. Query 6 is an exception where both MAL-Weld and REL-Weld beat MonetDB and Vectorwise. In the following sections we focus on evaluating two queries that exhibit the best speedup (Query 1 and Query 6) and two other queries with no or small performance gains (Query 3 and Query 4).

## 5.1 MonetDB + Weld vs MonetDB

The main goal of our work is to improve MonetDB's response time for a given query which means that the most important metric of our benchmarks is the run time. To compare the performance of our implementations against MonetDB we plot the speedup in Figure 5.2, which is determined by the formula: speedup $= \frac{T_{MonetDB}}{T_{Weld}}$. The Figure consists of 16 plots arranged in a gird where the row represents the query and the column the scale factor. The plots show the speedup of the the MAL-Weld and the REL-Weld translation, and a horizontal line *(y = 1)* showing MonetDB's reference performance. The run time recorded for the Weld versions do not include the compilation time.

### TPC-H Query 1

In Query 1 The REL-Weld translation shows an impressive performance gain with the speedup ranging between 4 and 10. In this query we are able to generate a Weld program that optimally groups and aggregates the data in a single for-loop. The best relative performance is achieved on larger inputs, where the materialization cost in MonetDB is higher and the REL-Weld implementation can gain ground through its pipelined execution model.

In MonetDB, however, grouping and the aggregations are done in two phases. The MAL-Weld translation therefore also generates code that is separated into these two phases: determining the groupIds and then aggregating the other columns based on the groupId which was assigned to each tuple. In the second phase we generate a hash grouping for each aggregate, which Weld is unable to fuse into a single one. For Query 1, the MAL-Weld translation performs roughly the same operations on the data and also suffers from the same amount of intermediate materialization as MonetDB due to the lack of loop fusion (as also shown in Table 5.1). However, it is expected that given the same execution plan Weld's generic data structures perform worse than MonetDB's highly optimized ones.

### TPC-H Query 3

Query 3 contains two inner joins and in both of the joins one of the columns is a foreign key to the other. This means that in REL-Weld a *dictmerger* can be used to build the hash table on the column with unique values. For this query in particular it happens that by building the hash table on unique column we break the operator pipeline identified by the produce-consume model and we introduce an additional intermediate materialization. However, this materialization choice resulted in an improved run time by a factor of 100 over the pipelined execution as we were able to use the faster and more lightweight dictionary. In this case Weld appears to scale better than MonetDB and REL-Weld shows a small improvement over MonetDB when running on multiple threads.

MAL-Weld yields a degraded performance for the same reasons as with Query 1 and the fact that the two joins are now performed by MonetDB sequentially.

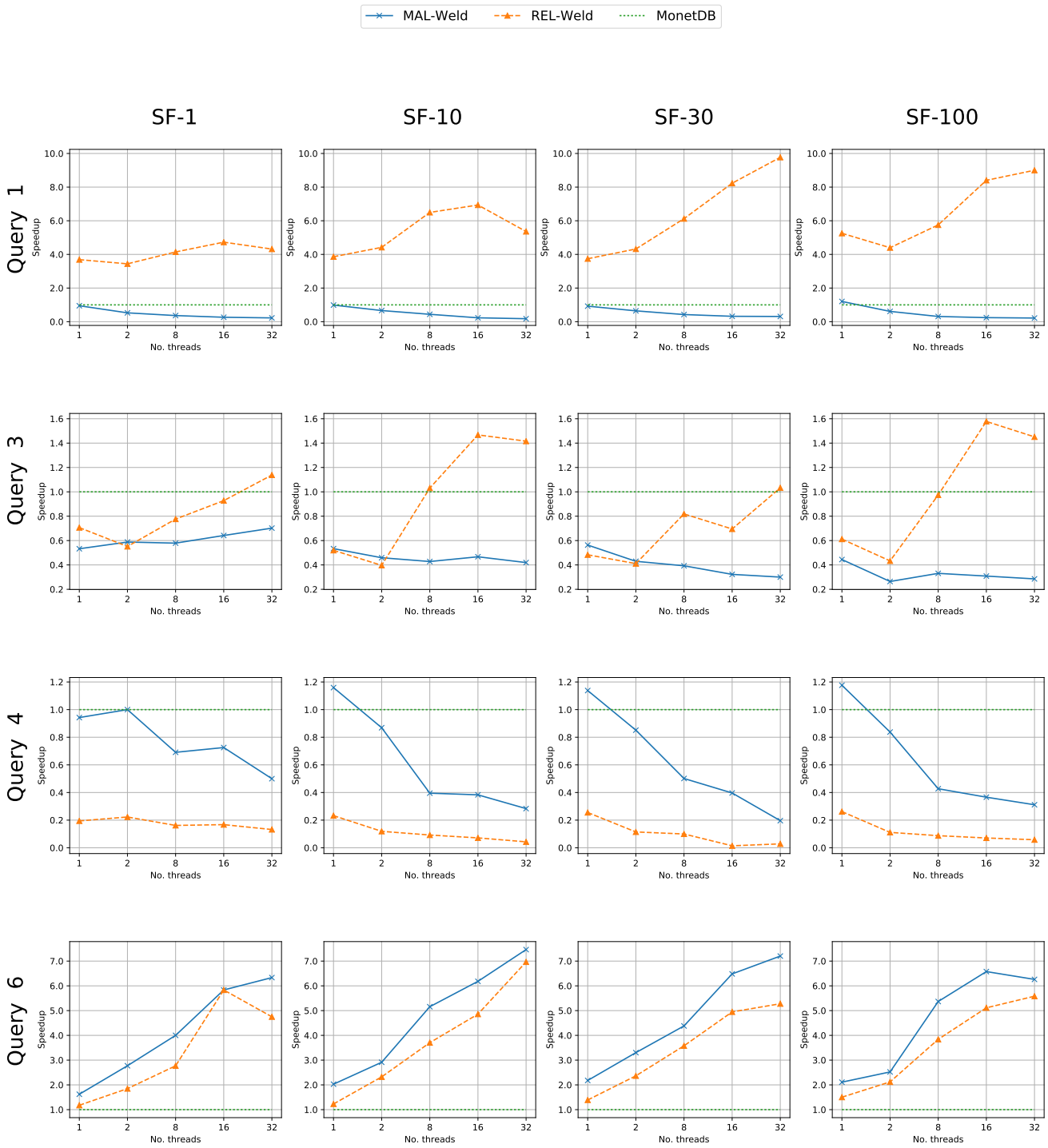


Figure 5.2: Speedup of the REL-Weld and MAL-Weld implementations over MonetDB. More means better.

### TPC-H Query 4

The main operations in Query 4 are a semi-join and an aggregation. In REL-Weld the semi-join is implemented using the better dictionary (*dictmerger*) and the generated Weld code perfectly captures the data flow imposed by the produce-consume model. It is therefore disappointing to see that REL-Weld's performance is at best 5 times worse than MonetDB's.

MAL-Weld shows a small speedup over MonetDB when running on a single thread, but the performance degrades abruptly when increasing the level of parallelism. We attribute this behavior to the fact that the semi-join is still performed by MonetDB, and while we can enable parallelism for the rest of the Weld program, we cannot do so for a single MAL instruction. Therefore, in MAL-Weld the join operation will always execute sequentially while in MonetDB it will be parallelized.

### TPC-H Query 6

Both of our solutions perform better than MonetDB. Query 6 consists of a Selection followed by an Aggregation of the whole filtered data, without grouping. The Weld code generated by the two approaches is almost identical, the only difference lies in how data is traversed. In MAL-Weld we iterate over one of the columns and then use the iteration index *i* to lookup values in the other columns, whereas in REL-Weld we iterate over the columns at the same time using the *zip* function. The *if* statements used for filtering, which make up the rest of the program, are applied in the same order. It is therefore surprising that MAL-Weld is about 20% faster than REL-Weld.

In Table 5.1 we show how for Query 6 REL-Weld generates about 2 times as many instruction references and 4 times data references as MAL-Weld. In a small experiment we replaced the 5 variables from the two Weld programs (the variables used to filter the data in Query 6) with literals and both programs generated about 50 mil IR and around 7.5 mil DR - which is what MAL-Weld generates by default. This shows that in the REL-Weld translation the 5 variables are needlessly reread multiple times and that a small change in the Weld program caused the LLVM compiler to miss an optimization opportunity.

## 5.2 Multicore performance

Figure 5.3 shows the speedup gained by the three programs when we increase the number of threads. The speedup is determined by the formula speedup $= \frac{T_1}{T_p}$, where the reference value $T_1$ is the evaluation time for the query running with only one thread, and $T_p$ is the evaluation time for the query running on $p$ threads. $T_1$ and $T_p$ are computed on the same program flavor for each curve in the plot. This experiment is an indicator of how the system scales on a machine with multiple execution units.

The one clear observation is that, the performance of Query 4 degrades when increasing the scale factor, while for Query 1, 3 and 6 REL-Weld and respectively both REL-Weld and MAL-Weld beat MonetDB. The best speedup is achieved for Query 1 and 6 which are more computationally intensive as the aggregations make up for most of the tasks.

## 5.3 Performance issues

As we saw in the previous chapter, there is a huge discrepancy between the performance gains in Query 1 and 6 and the slowdown in Query 3 and 4. In order to investigate the underlying issue we take a look at one of Weld's core data structures: the dictionary. The *dictmerger* builder is Weld's implementation of a hash map which we use to handle the *Aggregation* operator. To compare it against MonetDB, we

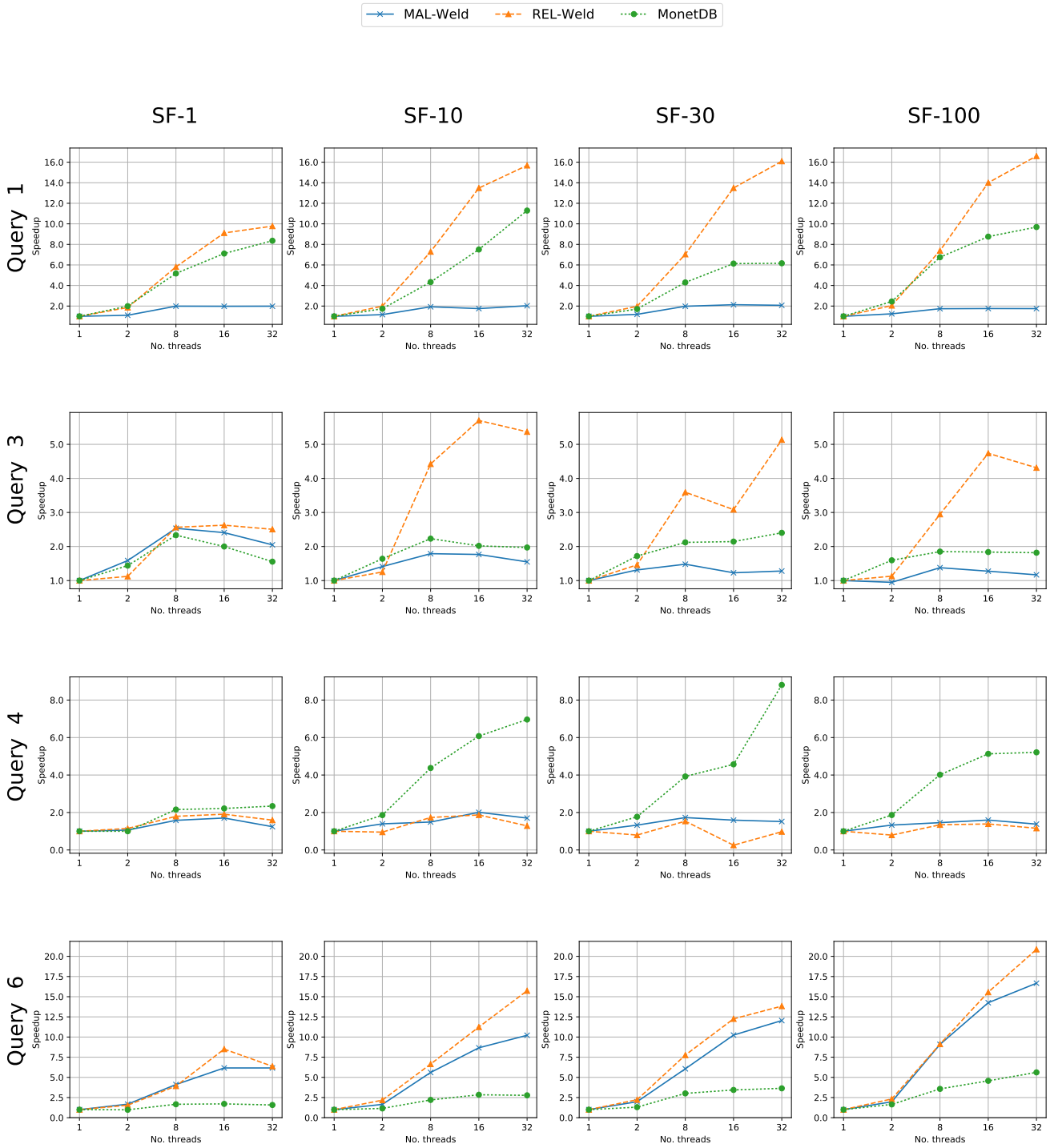


Figure 5.3: Multicore speedup of REL-Weld, MAL-Weld and MonetDB. More means better.

issue a simple query that counts the number of unique keys in column: `SELECT x, COUNT(x)FROM table GROUP BY x`. We varied the number of uniformly distributed unique keys from 1 to 10 million and we plot the results in figure 5.4. In this experiment we also include the *vecmerger* builder which acts as a dictionary with a fixed and known number of keys. We use the *vecmerger* for the aggregations in the MAL-Weld translation as the groupings, and implicitly the total number of groups, have been previously determined.

We can observe that for a small number of keys Weld's dictionary scales well and running under 32 threads the performance is equal to MonetDB's. This is exactly the case of Query 1 and 6, regardless of scale factor there are always 4 and respectively 1 groupings. On the other hand, for Query 3 and 4 we have more expensive Join operations which require the creation of dictionaries with multiple keys. As Figure 5.4 shows, when dealing with 10 million keys Weld's dictionaries can be almost up to 8 times slower than MonetDB's.

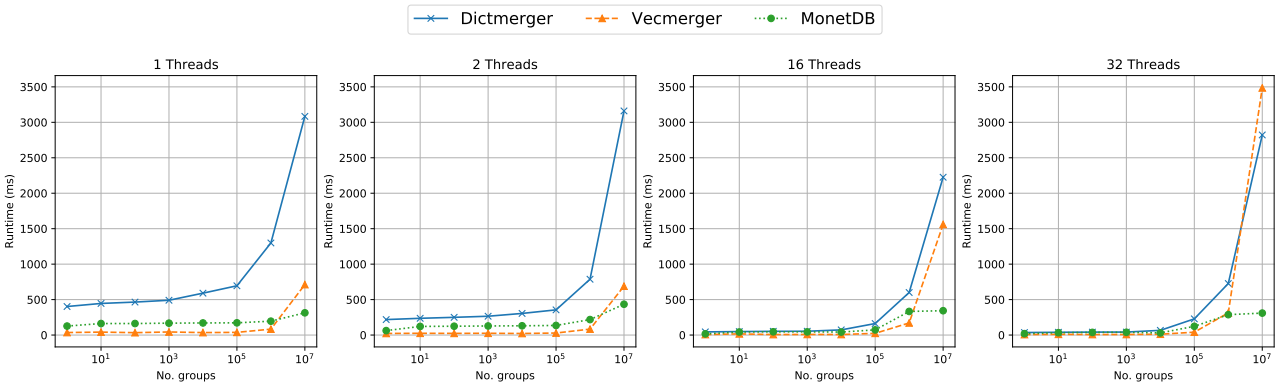


Figure 5.4: GROUP BY performance comparison between Weld and MonetDB.

The Weld dictionary is implemented in C++ and has multi-threading support. Each thread has a thread local dictionary and a local buffer to which entries, that no longer fit in the local dictionary, are appended. By default, the local dictionaries can hold up to 100 MB of data and the local buffers are drained to a global dictionary every 128 items.

Part of the differences in performance can be attributed to the fact that in Weld the dictionary is backed by a linear hashtable whereas MonetDB uses a bucket chained hashtable. Moreover, the Weld dictionary is generic, it maps bytes to bytes. Given the fact that Weld specializes in JIT compilation, a dictionary that takes advantage of the data type could be generated. For every key-value pair that is inserted into the dictionary there are two expensive calls to *pthread_getspecific* just to get the threadID and then there is a backward call to LLVM to actually perform a merging operation on the key. This shows that the dictionary does not integrate well with the rest of the generated LLVM code and that a significant overhead is generated through potentially avoidable function calls.

Weld allows users to pass hints to builders through annotations. In the case of the *dictmerger* we can specify the initial capacity of the hashtables. This is useful, for example, when constructing the hashtable used for probing in a join operation, when the materialized column has unique values, and we thus know the total number of keys. In Figure 5.5 we plot the runtimes of the same query as in the previous experiment: `SELECT x, COUNT(x)FROM table GROUP BY x` while we vary the initial dictionary capacity. In this experiment we used 32 threads and the data consisted of 1 million uniformly distributed keys, which can fit in a thread local dictionary. The vertical lines show the default initial dictionary size and the final size of the global dictionary. When setting the initial size to the final dictionary size we actually observe a considerable slowdown. This happens because of the way Weld splits the work among threads, for our data distribution a thread will not encounter all the keys and will thus have a smaller local dictionary. The size hint affects both the local and the global dictionaries and therefore it causes the local dictionaries to be unnecessarily large in our experiment, which leads to a slowdown in execution

time. We did not observe any runtime improvements for any of the possible initial sizes and we therefore conclude that resizing the hashtable does not add significant overhead to the Weld dictionary.

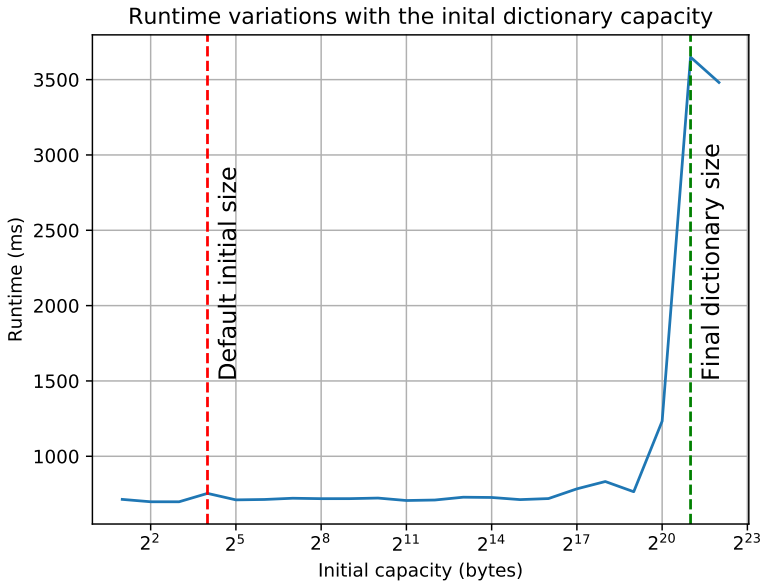


Figure 5.5: GROUP BY runtime variation with the initial dictionary capacity.

## 5.4 Code quality and maintainability

In order to better understand the performance differences between our Weld integrations and MonetDB it useful to look at the executed code in more detail. We profiled the execution of the three versions of MonetDB with *callgrind* and we limited the data collection through *callgrind*'s client requests API to only include the actual query execution. In table 5.1 we recorded the number of executed instructions (Ir), memory reads (Dr), level 1 cache misses (D1mr), last level cache misses (DLmr), number of branches executed (Bc) and the number of missed branch predictions (Bcm). We ran the experiment with one thread on scale factor 1.

We can see that MAL-Weld recorded worse stats than MonetDB for Query 1 and 3: there are a higher number of executed instructions and more memory references, but performs better than MonetDB for Query 4 and 6, where for the latter there are by a factor of 10 fewer memory accesses. For the REL-Weld translation we observe the same pattern as in the previous experiments: good results for Query 1 and 6 but otherwise poor performance for Query 3 and 4. Although in theory the produce-consume model should produce a data flow where tuples kept in the CPU cache as long as possible, we can see from the Query 4's recorded stats that this is not the case for Weld's data structures. The main drive behind the Weld translation was to reduce the intermediate materialization and improve the memory bandwidth, but we have instead increased the memory references by up to 5 times.

### 5.4.1 Compilation time

JIT compilation comes with the cost of the code compilation on the fly. Although JIT compilation can produce a more optimized code, the incurred overhead can actually nullify the benefits. We have recorded the compilation times for the four queries in table 5.2, and as we can see, they are in the order of hundreds of milliseconds. To put the compilation time into perspective, MonetDB can execute each

Table 5.1: Number of instructions, data references, cache locality and branching

| | | Ir (mil) | Dr (mil) | D1mr (mil) | DLmr (mil) | Bc (mil) | Bcm (mil) |
|---|---|---|---|---|---|---|---|
| Q1 | MonetDB | 6,000 | 1,379 | 27 | 27 | 1,227 | 7.63 |
| | MAL-Weld | 6,991 | 2,097 | 27 | 26 | 645 | 6.05 |
| | REL-Weld | 2,125 | 752 | 3.59 | 3.57 | 136 | 0.11 |
| Q3 | MonetDB | 440 | 157 | 5.24 | 2.68 | 77 | 1.94 |
| | MAL-Weld | 795 | 280 | 9.77 | 6.37 | 91 | 3.39 |
| | REL-Weld | 715 | 248 | 6.58 | 3.57 | 76 | 5.34 |
| Q4 | MonetDB | 325 | 114 | 2.65 | 2.04 | 69 | 3.01 |
| | MAL-Weld | 275 | 93 | 2.67 | 1.66 | 44 | 3.30 |
| | REL-Weld | 1,391 | 453 | 5.01 | 3.18 | 145 | 9.52 |
| Q6 | MonetDB | 190 | 86 | 1.25 | 1.16 | 32 | 1.59 |
| | MAL-Weld | 50 | 7.42 | 0.96 | 0.94 | 18 | 2.09 |
| | REL-Weld | 114 | 36 | 1.99 | 1.97 | 15 | 3.09 |

Table 5.2: Weld code compilation time

| | Q1 (ms) | Q3 (ms) | Q4 (ms) | Q6 (ms) |
|---|---|---|---|---|
| MAL-Weld | 977 | 810 | 405 | 127 |
| REL-Weld | 240 | 327 | 227 | 109 |

of the four queries on scale factor 3 and 32 threads faster than Weld can compile the code for them. Of course, for long running queries the compilation time becomes negligible and the JITing technique can be successfully applied.

To restore the responsiveness of the databases for short running queries a query cache can be implemented. The Weld programs that we are generating are parametrized, the code reflects the data flow and the input columns and the literals are passed as runtime arguments to the program. Therefore we have implemented a simple query cache where a Weld program is linked to a Weld module (the compiled and dynamically linked library) which is available to be rerun.

### 5.4.2 Maintainability

A MAL program in MonetDB has a distinctive data flow that results from the proposed BAT algebraic operators. The MAL-Weld translation has the advantage that the Weld integration can operate seamlessly within the already existing pipeline. By allowing the execution of both Weld implemented operators and GDK backed ones we can choose the best from two worlds and combine them efficiently. For example, to handle joins in MAL-Weld we rely on GDK join functions which we can call through and UDF in Weld, or we can generate two Weld programs that are connected by a join MAL instruction. As each MAL instruction usually performs a simple operation on several columns, the equivalent Weld code is also quite simple and concise. In our implementation, we have the Weld code fragment corresponding to the translated operation in a single string with variable names placeholders which we replace with *sprintf*. In other words, the Weld IR pieces are both easy to write and understand. The downside is that in GDK there is an extensive number of implementations for an also large number of BAT algebraic operators. A MAL instruction can have several variations where the number of arguments can vary, the instruction might or might not operate on a candidate list, and the candidate list might or might not be dense. Therefore achieving full compatibility through the MAL-Weld translation is not trivial.

In the REL-Weld translation we implemented the produce-consume model where each algebraic operator can generate a part of the pipeline in both of its produce and consume phases. An unfortunate early

design choice was to actually emit pieces of Weld code at each step and append them to a buffer shared between the operators. The consequence of this approach is that it is almost impossible for a new reader to understand what code is generated by an operator. Furthermore, developing the translation was quite error prone as the Weld fragments were scattered throughout the code, for example a pair of matching parentheses might be separated by a hundred lines of C code. On the other hand, as the data flow is dictated by the relational algebra, the generated Weld code in its final form is surprisingly easy to understand and to have its correctness verified. The Weld translation itself is also concise as we only had to deal with a small number of algebraic operators, and even though there might be several physical operator implementations to choose from, the number of variations still falls short of the large number of MAL instructions.

Overall, we believe that the REL-Weld translation is easier to maintain and develop. An alternative to producing bits of Weld code on the fly would be to construct an intermediate representation of the Weld program, which at end could be transformed into the string form of the program which is expected by the current Weld API.

# Chapter 6

# Related Work

The tuple-at-a-time Volcano [8] iterator model was popular during the time when the query execution performance was bounded by the IO capabilities of the system. However, as hardware evolved and the data access time was reduced, the interpretation overhead became non-negligible. Padmanabhan et al. propose in [13] a block oriented processing model for row-based databases, where blocks of tuples are passed from one operator to another. In his way, modern CPU features such as pipelining, prefetching and larger memories are better utilized.

Building on the idea of processing data tuples in blocks, a new query engine called X100 [6] (which later spun off into a new project called VectorWise) was developed for MonetDB. X100 combines the columnar data layout of MonetDB with the incremental materialization of the Volcano style pipeline. X100's performance derives from moving small chunks of columns (vectors) through the relational operator tree while the vector size is chosen such that the data slices fit into the CPU cache. X100 benefits from low interpretation overhead by maintaining the vector in the fast CPU cache and performing several different operations over a data subset. To improve the data processing time even further and to close the gap between the newly obtained data processing bandwidth and the I/O bandwidth, a new storage layer named ColumnBM [20] was developed for X100. ColumnBM relies on light compression and cooperative scanning to speedup the data scanning process.

An execution model based on query interpretation will always incur overhead. Some execution engines, such as MonetDB's for example, also suffer from unnecessary data materialization during the interpretation process. Just-in-time (JIT) compilation emerges as a possible solution for the two problems. The idea behind JIT execution is to generate and compile code on the fly for incoming queries. This is based on the assumption that a compiled program will always be faster/more efficient than its interpreted version. The code generation and the compilation does incur its own overhead, but in the case of DBMSs this cost can be amortized by the large scale of the data which the query will be executed on.

System R [7] was an early implementation of a relational database created with the purpose of demonstrating that such a system is feasible in a production environment. In System R a SQL query is decomposed into small machine language code fragments (assembly snippets) that are then put together and compiled into a single executable program. The authors observed an improved execution time for short and repetitive transactional queries, but also acknowledged that for ad-hoc it is less trivial to compensate for the compilation time.

Sompolski et al. show in [17] that indeed JIT compilation can improve the performance of the X100 engine when used in combination with the already existing vectorized execution. The research shows that there is no simple rule for deciding between the vector-at-a-time and the tuple-at-a-time model introduced by JIT compilation, but rather a dynamic analysis is required to determine which operator performs best in which situation.

The traditional algebraic operator model is focused on the operators themselves, rather than on the data, thus drawing a clear boundary between the execution phases of the query. Neumann [12] proposes a new evaluation technique called *producer-consumer* model which aims to remove the boundaries between the operators. The resulting execution model is again tuple-at-a-time, but unlike in the Volcano model, where data is pulled upwards in the query plan, data is now pushed from one operator to another and the execution becomes data centric. Conceptually, once an operator produces its data in the *produce* phase, it pushes the data to the next operator by invoking its *consume* procedure. This execution model has been integrated into HyPer [9], an in-memory database system, and its query performance has become mostly CPU bound. The first approach of JIT compilation in HyPer was to generate C++ code. The C++ code files were easy to produce and could seamlessly interact with the rest of the code and data structures from the database system. However the compilation times were too high (several seconds) and the performance proved to be suboptimal. HyPer now moved to generating LLVM code for the incoming queries and the compilation time is surprisingly small, in the order for tens of milliseconds for certain TPC-H queries. The complex parts of the system are still written in C++ and pre-complied, but the code paths that would take most of the execution time are JITed. As a result, the overall query response time, which also includes the compilation time, can beat MonetDB's by even a factor of ten.

The data centric model has been successfully implemented in other query engines such as SparkSQL [3], LegoBase [10] and DBLAB [16]. LegoBase and DBLAB are written in Scala and emit specialized C code, with the mention that DBLAB uses multiple intermediate Domain Specific Languages (DSL) and compiler passes in order to produce optimized code.

Another JIT code compilation technique that involves code templates is implemented in HIQUE [11]. The authors name their approach a *holistic query evaluation*, as during the code generation process the system looks both at the evaluated query and the underlying hardware. HIQUE parses the SQL query into an intermediate representation and then uses a catalog of templates to dynamically generate code for the operators in the query plan. The system generates a new C file which is compiled and dynamically loaded into the main process.Benchmarks of the system on the TPC-H with a scale factor of 1 shows that the performance is comparable to MonetDB's, but as the system uses the *gcc* compiler, the compilation time can go up to hundreds of milliseconds with the -O3 optimization level.

DBToaster [1] is a query compilation framework that specializes in generating query evaluators for maintaining long standing aggregation views under a high number of updates per second. Traditionally data increments are expressed and evaluated as queries, whereas DBToaster recursively considers multiple deltas for a given query and generates code that optimally applies them. The authors show that at each recursion the generated code becomes simpler, by generating C++ code they can avoid the interpretation overhead and then allow the compiler to further optimize the code through function inlining. DBToaster was shown to run faster than other responsive analytics systems by multiple orders of magnitude .

Impala [4] is a massively parallel SQL engine that runs on top of the Hadoop [1] environment. Impala takes a more localized approach to JIT compilation and focuses on optimizing function calls inside for-loops that get executed for every data record that is processed. The execution model is vectorized (as in X100) and the and the operators are individually optimized by substituting data-manipulation methods with JIT compiled snippets. A significant part of the interpretation overhead is generated by virtual functions, which are regularly used in the code to ease the development. However, the exact function implementation can be determined at runtime and be replaced in the function call through code generation. Other optimizations include partially evaluating expression trees, loop unrolling and reducing the branching factor. Impala uses LLVM to instrument JIT compilation and has achieved a speedup of up to 5 over some TPC-H queries.

In [18] Tahboub et al. argue that the current query compilation techniques are quite complex and difficult to develop. The authors propose a simplified technique based on the concept known as Futamura

[1] http://hadoop.apache.org/

projection, which links interpreters to compilers. The idea behind the Futamura projection is that a compiler can be seen a specialization of an interpreter. In this context, specialisation refers to adapting the execution of a generic function for a particular use-case, based on a parameter. Another step forward in this direction takes us to programmatic specialization which allows a function to self specialize for any input. Using this insight the authors show that they can obtain similar results to the produce-consume model in a more intuitive way, denoted data centric evaluation with callbacks. The *produce* and *consume* interface is refactored into a single method that takes as argument a *callback*. Each operator generates its result and then applies the callback function on every tuple. The authors implemented this new approach in the LB2 [18] query engine and they show that its performance is comparable to other state of the art JIT compiled query engines such as HyPer [9].

A project that shares some design goals with Weld is Voodoo [15]. While Weld can, at the moment, only generate code for the CPU through LLVM IR, Voodoo specializes in producing portable OpenCL code which can run both on CPUs and GPUs. The data model behind Voodoo is called Structured Vectors, which map to arrays of structures in C and are addressable on most hardware platforms. The intermediate vector algebra in Voodoo consists of several operators which allow parallelism at runtime without the use of hardware specific abstractions. [15] shows how Voodoo can be integrated into MonetDB, replacing the existing query engine, and how the resulting performance is comparable to HyPer.

# Chapter 7

# Conclusion

## 7.1 Contributions

In this thesis we presented two approaches of integrating JIT compilation into MonetDB through Weld. The first Weld integration is at MAL level, where individual MAL instructions are translated into fragments of Weld code which can operate on the same input and produce an identical result. We showed how Weld's program optimizers can combine the code fragments associated with Selection and Projection operations through loop fusion, but fails to improve the data flow and reduce the materializations encountered in Aggregations and Joins - which results in an overall poor query execution performance.

In the second approach we generate a data-centric Weld program from the relational algebra using the produce-consume execution model. We have shown that Weld's data structures are expressive and have allowed us to perfectly implement the conceptual operator pipelines. Benchmarking the REL-Weld translation showed that Weld can outperform MonetDB on several simpler TPC-H queries, but it is otherwise slower on queries involving Joins.

## 7.2 Answers to research questions

Our research questions were:

- Can the query compilation into native machine code reduce the amount of intermediate materialization by removing the boundaries between operators and fusing them together?

Both Weld translations are able to reduce the amount of materialization. Weld's loop fusion optimization passes and the effects of the data flow resulting from the produce-consume model are best seen in TPC-H Query 6. Both MAL-Weld and REL-Weld generate a single for-loop for the query evaluation, whereas MonetDB executes 7 for-loops.

- Is Weld able to perform loop fusion given the data flow generated by the MAL program? If not, can we add new optimization rules to Weld?

Weld was not able to optimize the data flow that results from MonetDB's aggregation operations. As we showed in Section 4.1.3, this loop fusion opportunity would require tracking functional dependencies in order to eliminate the groupIDs from the aggregation process and resort to the usage of a single *dictmerger*.

- Can we express all relevant operations using Weld's IR and parallel builders?

Weld's for-loops and generic data structures have been sufficient for our use-case and have allowed us to express the data flows and the main operations encountered in a query plan. Weld also allows the usage of UDFs which were useful when dealing with more complicated transformations such as extracting a year from a date or performing string manipulations. However, in our work we focused on translating TPC-H queries where the aggregation operations only consisted of *sum* (*avg* was expressed as *sum* as well), *min* and *max*. When we started the project, Weld only supported the *sum* and *prod* operations so we implemented ourselves *min* and *max*. Moreover, Weld currently only allows a single merging operation per builder (or per aggregation), which was luckily the case in the TPC-H queries and we have, therefore, not run into this limitation.

- Is the data-centric compilation approach in REL-Weld superior to MAL-Weld?

Yes, in the REL-Weld translation we no longer rely on Weld to combine the operators, but we instead generate the Weld code such that the tuples are processed by several operators pipelines uninterrupted by materialization. As Weld was not able to detect all the optimization opportunities in MAL-Weld, the REL-Weld approach yields better performance.

## 7.3 Future work

We believe that REL-Weld is the best approach to integrate Weld into MonetDB given the benchmarking results and implementation experience. We therefore discuss the future work from REL-Weld's perspective. One of our goals in the Weld translation was to fully support the 22 queries from the TPC-H benchmark. Unfortunately we did not succeed in running all the queries, so further steps are needed in order to ensure better query support:

**(FW1)** All the necessary data manipulation functions are already implemented in MonetDB and can be made easily accessible to the Weld runtime through UDFs. Some of the functions, such as those that process the string *like* operator, are only chosen at runtime inside GDK. More work is needed on the relational algebra parsing so that the correct UDF is chosen before running the Weld program.

**(FW2)** The relational algebra itself can be tuned for the produce-consume model and the Weld library. Certain operators are designed with the MAL interface in mind, for example the GroupBy operator can contain projections or the result of an Aggregation can have a different type from the aggregated data.

Beside the translation issues we described above, the performance of the Weld runtime also has to be improved in order for it to compete with MonetDB's:

**(FW3)** We argue that the main cause of slowdown in our Weld translation are the dictionaries. The dictionaries are used in some of the most common and significant operations such as Aggregations and Joins.

**(FW4)** Weld can apply a merging operation (e.g. *sum*) on a struct by applying the operator on every member for the struct. An improvement in this direction would be the support of different operations for each struct member, possibly even a user defined function.

**(FW5)** Weld is still in its development phase, therefore more work is required towards a final product. During our work with Weld we have encountered several bugs and unimplemented features for which we opened issue tickets on Weld's Github page [1].

[1] https://github.com/weld-project/weld/issues?q=is%3Aissue+author%3Amihai-varga+

# Bibliography


[1] Y. Ahmad and C. Koch. Dbtoaster: A sql compiler for high-performance delta processing in main-memory databases. *Proceedings of the VLDB Endowment*, 2(2):1566–1569, 2009.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[4] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[5] P. Boncz. Monet; a next-generation dbms kernel for query-intensive applications. 01 2002.

[6] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[7] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[8] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[9] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[10] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.

[11] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010.

[12] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[13] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society.

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

[15] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14):1707–1718, 2016.

[16] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1907–1922. ACM, 2016.

[17] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

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

[19] M. Zukowski, P. A. Boncz, et al. Vectorwise: Beyond column stores. 2012.

[20] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. Monetdb/x100-a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.