Universitatea Politehnica Bucureşti   Vrije Universiteit Amsterdam

Master Thesis in

Parallel and Distributed Computer Systems

Micro Adaptivity in a Vectorized Database System

Bogdan Răducanu

2206449 (VU)

Supervisor:
Marcin Żukowski
Actian BV

First reader:
Peter Boncz
Dept. of Computer Science
Vrije Universiteit
Centrum Wiskunde & Informatica

Second reader:
Henri Bal
Dept. of Computer Science
Vrije Universiteit

August 2012

Abstract

This thesis investigates the benefits of micro adaptivity in a high performance DBMS. A micro adaptive DBMS is able to tune itself to the context in which it is running, by having multiple implementations of performance critical sections and a mechanism that chooses the best (fastest) one during runtime. This is different than other adaptive database systems, such as those that continuously tune the execution plan at runtime because these do so at the coarser operator level. Our goal is to augment the Vectorwise system with micro adaptivity which should increase performance and also reduce performance variation due to data characteristics, hardware or system state. We first present the factors that cause changes in performance (e.g. different hardware features, data selectivity) and show the opportunities that they create. Then, we introduce the micro adaptivity sub-system and describe how it is implemented within Vectorwise. The thesis concludes with an analysis of results obtained on the TPC-H decision support benchmark.

Contents

Chapter 1

Introduction

Recent hardware advances (increasing parallelism, memory size) have led to the redesign of well established technologies, such as the Database Management System (DBMS). The expanding data volume that needs to be processed by these systems has made performance more important than ever.

The object of this research project is Vectorwise [Zuk09], an innovative high performance analytical database system developed at CWI and now a product of Actian. Its efficient vectorized execution engine gives us the opportunity to apply methods that previously did not appear in database systems. In this case, our goal was to enrich Vectorwise with *micro adaptivity*. This means the execution engine will tune itself at runtime by choosing between functionally equivalent performance-critical functions, which produce the same results but are implemented or compiled differently. The goal of this is to increase performance by taking advantage of different optimizations but also to make the performance more robust (i.e. less influenced by hardware variation, data characteristics or system state).

A number of cases, presented in Chapter 3, suggest that adaptivity will be beneficial. For these, Vectorwise already uses simple heuristics. However, these are often not optimal, because they rely on constants that were determined experimentally on one platform, so they might not be as good on other platforms. Therefore, a secondary goal of this project is to implement a generic framework that will maximize the benefit of these existing cases and possibly that of the new ones.

This thesis is structured as follows. Chapter 2 presents key aspects about the Vectorwise execution engine, relevant to this research. Chapter 3 contains case studies that form the argument for micro adaptivity as well as a base for our prototype. Chapter 4 describes our micro adaptivity implementation, showing how we generate, store and load flavors. Here, we also analyze the underlying optimization problem and our solution. Finally, in Chapter 5 we show concrete results obtained on the TPC-H analytical DBMS benchmark.

1.1 Motivation

The Vectorwise engine (discussed in more detail in Chapter 2) contains over 5000 specialized functions, called *primitives*, which perform the actual data processing when executing a query. There are primitives for arithmetic operations, comparison operations (equality, inequalities), aggregations, etc. A SQL query is translated to an *operator tree* and then data flows between operators (see also Section 2.3). Every operator (e.g. Select, Join, Project) calls some of these primitives to perform its task. For example, the query `SELECT l_orderkey FROM lineitem WHERE l_quantity > 40` involves a Select operator which will call a primitive that performs *greater than* comparisons on integer values. The amount of CPU cycles spent executing this query is shown in Table 1.1. The operators are initialized in the preprocessing stage and called in the execute stage. Almost all the time is spent in this stage (99.92%) and within this stage, almost all the time is spent in primitives (92.17%). Thus, **the performance of these primitives is critical to the overall performance**.

Table 1.1: Example of time spent in different query execution stages

stage	CPU cycles	%
preprocess	1679694	0.03%
execute	4321561972	99.92%
primitives	3983412990	92.17 %
postprocess	807654	0.01%
total	4324736514	

The performance of primitives is affected by many factors, including: hardware (e.g. processor speed, cache size, branch predictor, memory speed), compiler (e.g. optimizations, use of SIMD code), query context (e.g. selectivity), manual optimizations (e.g. loop unrolling, loop fusion, loop fission, branching removal), machine state (e.g. other threads thrashing the cache, or using memory bandwidth). As a result, **different implementations of the same primitive can behave differently, depending on the context**.

1.1.1 Example

A good example of context dependent performance are branching and non-branching implementations of selection primitives. The branching primitives use *if* statements to test a predicate while the non-branching primitives use logical operators and index arithmetic to completely remove any branching. The primitive in Listing 1.1 performs selection on a vector. It accepts as arguments a vector `col` of **`int`**`s` and its size `n`, a vector `res` where to store the result and a constant `val`. It produces a *selection vector* with the indices of the elements in the input vector which have a value strictly less than the constant value. The selection vector is then passed to other primitives (see also Section 2.5.2). The implementation in Listing 1.1 uses a branch to perform its work while the primitive shown in Listing 1.2 is branch free. These implementations are equivalent (i.e. they will produce the same result).

Modern processors attempt to predict the outcome of branches, with the aim of executing future instructions, in case there is not enough work at the moment, thus increasing efficiency. This prediction relies on heuristics, so it has its limits. It will do a good job when there is a simple pattern (e.g. the branch condition is always true or always false). In our case, this pattern depends on the data being processed. Consequently, **if we use branches, the performance of the primitive will depend on the data**.

Without branches, there is no need to rely on heuristics to predict which instruction is next, so performance will not depend on data and the processor is free to execute future instructions.

More details about the effects of branch prediction on query execution performance can be found in [Ros04] and [CG02].

```
size_t select_less_than(size_t n, int* res, int* col, int* val){
        size_t k = 0, i;
        int v = *val;
        for(i = 0; i < n; ++i){
                if(col[i] < v]) res[k++] = i;
        }
        return k;
}
```

Listing 1.1: A simple selection primitive with branching

```
size_t select_less_than(size_t n, int* res, int* col, int* val){
        size_t k = 0, i;
        int v = *val;
        for(i = 0; i < n; ++i){
                res[k] = i;
                k += (col[i] < v);
        }
        return k;
}
```

Listing 1.2: A simple selection primitive without branching

1.1.2 Need for adaptivity

The non-branching implementation always does the same number of arithmetic operations, while with branching, this number depends on the data. If the data is such that the branch is almost never taken, then the non-branching implementation will do a lot of unnecessary work, while the branching implementation will do exactly the required amount of work and benefit from branch prediction (since the pattern is simple). So, **which is the fastest implementation depends on the data**. This can be seen in Figure 1.1, where *selectivity* is the probability that the branch condition will be true. We can see that for very low and very high selectivities, branching is the fastest implementation while non-branching is the fastest in the other cases.

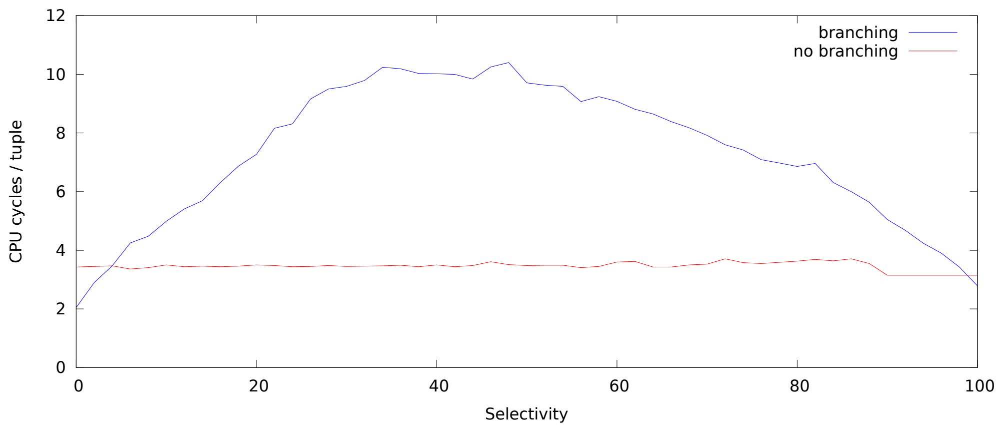

Figure 1.1: Performance of branching and no branching primitives on synthetic benchmark (from [Zuk09])

Figure 1.2 is a preview of a case study that will be presented in Section 3.4. It shows how the performance varies within the same query, due to the changes in data selectivity. In this example

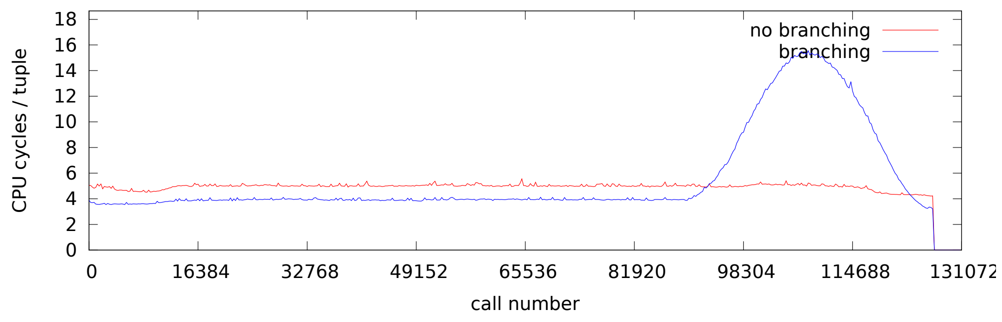

Figure 1.2: Performance of branching and no branching primitives on TPC-H Query 12

(which is from TPC-H Query 12), the primitive is called 126976 times and for most calls, the branching primitive is faster (by around 20%, from 5 cycles/tuple to 4 cycles/tuple). Towards the end of the query, the selectivity changes and no longer favors the branching implementation which starts to perform much worse than the non-branching one. This is because that selectivity makes it more difficult to predict the outcome of the branch and the processor incurs a penalty whenever it mispredicts (extra cycles are needed to flush out the wrong instructions from the pipeline).

1.1.3 Summary

This section showed an example of how one can have two equivalent primitive implementations that perform differently depending on the selectivity, demonstrating data dependent performance. We also showed that during execution the context may change, so different primitive implementations may perform better at different stages of the execution of a single query. There are many other possibilities worth exploring: e.g. implementations with different loop optimizations, implementations targeting different hardware features, etc.. We can even have some implementations that perform poorly, except in just a few cases, where they are better than the standard implementation.

The Vectorwise execution model that concentrates computational effort of a query in the vectorized primitives thus gives us a unique opportunity to further increase the performance of DBMS-es by using a technique that is not usually used in database systems. This project aims to make the execution engine *micro adaptive*. This means that the execution engine should be able to adapt to the current context by switching to the best performing primitive implementation. This technique is sometimes used in numerical applications, which also have computationally intensive parts that can be implemented in different ways. Other DBMS-es do not have this exact architecture, but micro adaptivity can still be useful. The requirement is to be able to identify computationally intensive kernels within the execution engine.

1.2 Related Work

There are quite a few projects that have successfully created adaptive systems, but there is yet no DBMS, that we know of, to use this paradigm. This is because, unlike Vectorwise, other DBMS-es do not have these distinguishable performance-critical vectorized primitives.

1.2.1 Adaptive scientific applications

FFTW [FJ98] is a library that computes the Discrete Fourier Transform and tunes itself to the hardware that it is running on to achieve better performance. Adaptivity in FFTW is achieved by using a *planner* which, prior to computing the actual transform, tests multiple execution plans and chooses the fastest for that machine. An execution plan is a decomposition of the problem into simpler sub-problems. The simplest sub-problems are solved by specialized code fragments called *codelets*. For example, there could be a codelet that is optimized for solving real transforms, one for complex transforms, codelets that use SIMD instructions, etc. They are generated automatically based on the problem size, but one can also hand write them (for example, to make use of some machine specific feature). In a way, these codelets are similar to Vectorwise primitives, which too are generated automatically for different specializations (e.g. there is a primitive for every arithmetic operation and every possible operand data type combination, primitives that make use of SIMD or loop optimizations). However, in the case of Vectorwise, the performance of primitives is also data dependent, so it will not be sufficient to only rely on a planner that chooses the best primitive flavors once. Our system will need to be continuously adapting, to handle changes in query types, selectivities, etc.

The ATLAS project [APD00] is another example of how adaptivity can achieve a good and portable performance, this time for linear algebra operations (e.g. matrix multiplications). ATLAS has an install phase where it first probes the hardware (e.g. determines the L1 cache size, availability of fused multiply add instructions). Based on these parameters, it then generates different implementations and benchmarks them to find the fastest. For example, it produces different matrix multiply implementations, varying the blocking factor or the loop unroll factor.

Adaptivity can be pushed even further, to support, not only things such as different cache sizes, but also completely different architectures (e.g. NUMA, GPU). [DMV$^+$08] presents a system for computing stencil operations that is able to generate different implementations targeting various architecture specific features (e.g. NUMA, DMA).

The previous three projects attempt to achieve *performance portability* (i.e. get roughly the same performance no matter on which hardware they run). A slightly different approach is presented in [LGP04]. Here, the aim is to have an adaptive sorting library. Again, the performance of this library will depend on the hardware (cache size, number of registers, etc.) but, this work shows that data (i.e. array to be sorted) also has an affect on performance. So, based on some statistics about the data, the library chooses the best sorting algorithm. This is similar to what we intend to have, although, in the case of a DBMS it is not possible to compute statistics on all the data in advance (because query plans and data values vary between queries). Computing statistics on the vector that is about to be processed is possible, but since most of the Vectorwise primitives are simple loops, this would create too much overhead. A more lightweight approach is needed.

1.2.2 Adaptive Query Processing

Adaptivity in DBMS-es is often implemented in the execution plan level, e.g. by modifying the plan at runtime or changing the order in which data flows between operators. As noted in [DIR07], adaptive query processing (AQP) attempts to overcome the difficulties encountered by DBMS-es that use the traditional optimize than execute approach. The optimize phase relies on having estimates about the cardinality, selectivity, etc. and in modern workloads these might be unreliable or even impossible to produce (e.g. streaming queries, remote data sources). [IC91] shows that the error for these statistics increases exponentially with the number of joins. This would mean that for complex queries, such as those processed by analytical systems, the chosen execution plan could be far from optimal. Additionally, when executing long running queries, the context parameters (data characteristics, system state) may change, so a static plan approach would lead to poor performance.

The Symmetric Hash Join operator is one way to introduce adaptivity in query processing. Traditional Join operators have a build phase in which they create a hash table with keys from the smaller table and a probe phase in which tuples from the other table are looked up in the hash table. Deciding which is the smaller table is done in the query optimization phase and changing this in the execution phase is costly. Furthermore, the standard Join operator has to wait for the build phase to complete before it can produce results. The Symmetric Hash Join (SHJ) operator solves both problems by building hash tables on both tables. By doing this, the operator can produce results immediately. Also, the SHJ operator naturally handles the case when tuples arrive in an interleaved order (alternating between the two tables), which might happen in wide area environments. This operator introduces some adaptivity on its own, but it is also suited to be part of a broader adaptive system, which continuously changes the execution plan, because it supports changing the join order.

The Eddy operator, described in [AH00] achieves adaptivity by changing the order in which tuples are processed by operators (tuple routing). Every operator receives input tuples from an Eddy and sends output tuples back to the same Eddy, which will then send them to other operators. When a tuple is processed by all operators, the Eddy will send it to the output. Eddies build statistics about each operator and use them to decide where to route tuples. Since every tuple passes through Eddies, they can build fine grained statistics for each operator.

The MJoin operator is a generalization of the binary Symmetric Hash Join to multiple inputs. Similar to the Eddy operator, it routes tuples from one hash table to another.

The adaptive query processing methods presented in this section can be complemented by micro adaptivity since this operates on a lower level and is not affected by changes in the execution plan.

1.3 Research questions

There is already strong indication that Vectorwise will benefit from adaptivity. For example, some primitives are faster when Vectorwise is built with a certain compiler (see Section 3.5). So, in the first phase of the project we will find out **what are the factors that affect primitive performance (Q1)**. Here, we are interested in optimizations which result in improvements in some contexts, but not in all [1] . Practically, this involves evaluating many primitive implementations (e.g. branching, non-branching, SIMD, non-SIMD, different loop unrolls). Additionally, we want to **determine the context in which one implementation is better than another (Q2)**. Knowing what triggers this change in performance will help us **design a selection method (Q3)**, which will choose between implementations, at runtime. The selection method will also keep a state (e.g. performance history for each primitive), so it can make better decisions. What this state consists of and how often a selection occurs depends on the findings for Q2. For example, the selection method can be called between queries (a query possibly involving millions of primitive calls) or within a single query. In the end, we will emphasize **the benefits of adaptivity in a high performance DBMS** and also present **the key requirements for a DBMS to have any benefit from adaptivity**.

[1] Of course, if there is some optimization that is always improving the performance, there is no need for adaptivity. But even what seem to be obvious improvements, like using SIMD, might be detrimental in some cases.

Chapter 2

Vectorwise engine

Vectorwise is a high performance RDBMS (relational DBMS) for analytical workloads, i.e. workloads that involve complex read operations on large data-sets. In this domain, it is currently one of the fastest systems, according to benchmarks such as TPC-H. Its performance comes from innovations in data storage, where it uses efficient compression algorithms to increase data throughput, as well as in execution which is *vectorized* in order to benefit more from modern hardware features. This chapter describes how a query is executed within Vectorwise.

2.1 Relational database model

The relational model[Cod69] was introduced in 1969 and is still used today. In this model, data is viewed as a mathematical relation: given n sets S_1, S_2, ..., S_n, a relation R is a subset of $S_1 \times S_2 \times ... \times S_n$. For example, if we want to keep information about some products, we can use the sets $prod\_id = \{1, 2, ...\}, in\_stock = \{yes, no\}, color = \{blue, black, orange\}$. These relations are typically known as tables, the sets as columns and elements of relations as *rows* or *tuples*. A row usually describes some entity (e.g. a product) for which the columns can be thought as attributes. Sometimes, one or more attributes can be combined to uniquely identify the entity, in which case this combination is called a *key*.

There are a few operations with relations that form the basis of this model and are implemented in any RDBMS. A **projection** of a table is obtained by removing some columns, e.g. keeping only the `prod_id` column. A **selection** is obtained by removing some of the rows, e.g. keeping only the `products` that have the `blue` attribute. A **join** operation combines data from multiple tables based on some condition. For example, we might want to combine data from the `products` table with data from the `sales` table. We combine the rows that have the same `prod_id` attribute. In a RDBMS, these operations are implemented in *operators* and there can be more than one possible implementation for a given operation.

2.2 Query optimization

To run a SQL statement, a DBMS first has to build an *execution plan* for it. This is done by the *query optimizer* component, which looks at all the possible plans for a query and chooses the one it considers to be the most efficient. This decision is made by computing an estimated cost for each plan. For example, different plans can be built by changing the order in which relations are joined. Another way to generate different plans is to use different implementations of the same operator, e.g. a join can be a HashJoin, MergeJoin, etc.

2.3 Operators

Operators are the basic components of an execution plan. They implement the logic of the relational operations. Vectorwise, like many DBMS-es, executes queries in a pipelined fashion, similar to the Volcano model described in [Gra94] (see also Figure 2.2). In this model, operators act as iterators and expose the `open()`, `close()` and `next()` methods. `open()` and `close()` are used for initialization and cleanup of an operator, while `next()` is called to process data.

An execution plan is a tree of operators (Figure 2.2). During query execution, data flows from the bottom of the tree to the top. An operator processes data pulled from its children. The leaves of the tree, which have no children, are operators which act as data sources. They either generate data or read it from storage.

On the lowest level of the operator tree there is usually a **Scan** operator, which reads raw data from database storage and acts like a source of data for the other operators. It accepts as arguments: the name of the table and a set of columns. The **Select** operator is used to filter out tuples that do not match some criteria and the **Project** operator is used to select a subset of columns, by name, or create new columns with arithmetic expressions. The **Join** operators are used to combine tables and there are a number of algorithms that can be used to implement them (e.g. MergeJoin, HashJoin).

2.4 Vectorized execution

Vectorwise proposes a compromise between the tuple at a time model and the column at a time model. The former has the advantage that intermediate result materialization is not needed, which would otherwise require a lot of memory (which is the case for column at a time). The penalty for this is increased interpretation overhead, inefficient cache usage, incompatibility with superscalar execution, and others [Zuk09]. So, instead of processing one tuple at a time, Vectorwise processes arrays of tuples, called **vectors**. The vector size should be small enough so that there is a high chance that the data will fit in the cache, but it should not be too small because then the advantages of vectorized execution are lost. Currently, the default vector size is 1024.

With vectorized execution there is more useful computation in each pipeline stage, amortizing the overhead of passing data between operators. Furthermore, there is a good spatial locality of data, which allows more efficient caching. Vectorized execution also makes better use of hardware features such as instruction level parallelism, SIMD instructions and memory prefetching.

2.5 Primitives

In Vectorwise, operators call primitives to do the actual processing of data. There are primitives for arithmetic operations, relational operations, hash table lookups, bloom filter lookups, string operations, aggregates, etc.

Each primitive is identified by a *signature*. For example, the `select_int_lt_int_col_int_val` primitive signature contains the type of the primitive (select), the return type (int, i.e. signed int), the relational operation (lt, i.e. less than) and the two argument types: a column with signed int values (col) and a vector with constants of type signed int (val). A **map** primitive performs some computation and produces an output of the same size as the input, while a **select** primitive applies a condition and returns a subset of the input tuples.

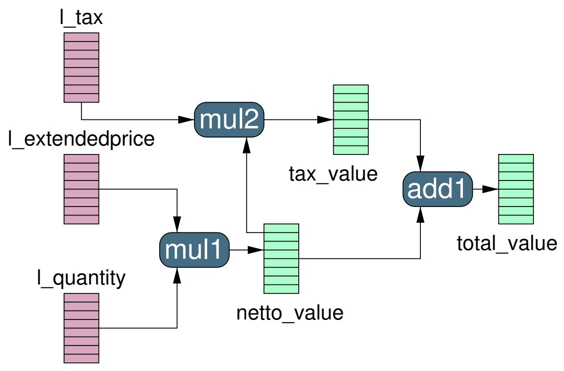

Figure 2.1: Query plan with primitives (from [Zuk09])

Listing 2.1 shows a **map** primitive which multiplies two integer columns. Here, `n` is the size of the vector, `res` is the results vector, `col1` and `col2` are vectors with attributes from the two columns.

Modern compilers will generate SIMD code for this loop, which, depending on the data type (the primitive shown here multiplies words, but there are similar primitives for shorts and longs), can give a great speedup (see also Section 3.8 and Figure 3.15).

```
size_t map_int_mul_int_col_int_col(size_t n, int* res, int* col1, int* col2) {
        size_t i;
        for(i = 0; i < n; ++i){
                res[i] = col1[i] * col2[i];
        }
}
```

Listing 2.1: Example of integer multiplication primitive

2.5.1 Primitive instances

We use the term primitive instance to distinguish calls to the same primitive in different contexts. For example, a **map** primitive can be called from a Project operator to multiply 2 pairs of columns, such as for the query shown in Listing 2.2. The plan for this query is shown in Figure 2.1, where **mul1** and **mul2** are two instances of a multiplication primitive.

```
SELECT  l_quantity * l_extendedprice AS netto_value,
                netto_value * l_tax AS tax_value,
                netto_value + tax_value AS total_value
FROM    lineitem
```

Listing 2.2: Example query that uses 2 primitive instances for a multiplication primitive

It is important to make this distinction because different primitive instances process different streams of data so their performance characteristics can be different.

2.5.2 Selection vectors

```
size_t map_int_mul_int_col_int_col(size_t n, int* res, int* col1, int* col2, int* sel
    ) {
        size_t i, j;
        if(sel) {
                for(j = 0; j < n; ++j){
                        i = sel[j];
                        res[i] = col1[i] * col2[i];
                }
        } else {
                for(i = 0; i < n; ++i){
                        res[i] = col1[i] * col2[i];
                }
        }
}
```

Listing 2.3: Multiplication primitive with selection vector

Many primitives accept an optional *selection vector* which defines the subset of input tuples that needs to be processed (see also Figure 3.11). Listing 2.3 shows such a primitive, which accepts the argument `sel` as the selection vector. Selection primitives generate these vectors which are then passed to other primitives, so that they avoid doing the unnecessary work of copying all column vectors after a selection operator, to eliminate the tuples that did not pass. However, sometimes it is beneficial to ignore this argument, as shown in Section 3.8.

2.5.3 Primitive flavors

We also introduce the term *primitive flavors* to refer to equivalent implementations of the same primitive in a micro adaptive system. The idea is to have a collection of flavors for each primitive, so the system can test out different implementations at run time and decide empirically which to use. Note that during query processing on a large table with a billion tuples, a **map** primitive used for some computation on all tuples is called about 1 million times inside the same query (for a vector size of 1000). Thus, there is an opportunity for the system to experiment with multiple flavors and decide on which is best. Section 4.1.1 describes how we can easily generate flavors using the Mx macro language and compiler flags.

2.6 Example

Figure 2.2 shows a simple query along with the operators and primitives involved in the execution of this query. In this example, the **Scan** operator retrieves data from storage and passes the attribute vectors to **Select** (i.e. id = [1, 2, 3, 4], name = ["Jan", "Anna", "Elise", "Hanna"], age = [...], income = [...]). **Select** would then invoke the **select_int_lt _int_col_int_val**[1] primitive function with the *income* values as the input vector. This primitive will apply the condition on the input vector and produce another vector, which will be passed to the **Project** operator to filter the tuples. **Project** uses the **map_int_mul_int_col_int_val**[2] primitive to compute an attribute vector corresponding to the expression $income * 1.2$ and then produces the final result.

[1]selects tuples from a signed int column which are less than some signed int constants
[2]multiplies values from a signed int column with a vector of signed int constants

SQL Query
select name, age, income, income * 1.2 as raise **from** people **where** income < 5000

people			
id	name	age	income
1	Jan	21	4000
2	Anna	23	5200
3	Elise	28	5800
4	Hanna	26	4800

results				
id	name	age	income	raise
1	Jan	21	4000	4800
4	Hanna	26	4800	5760

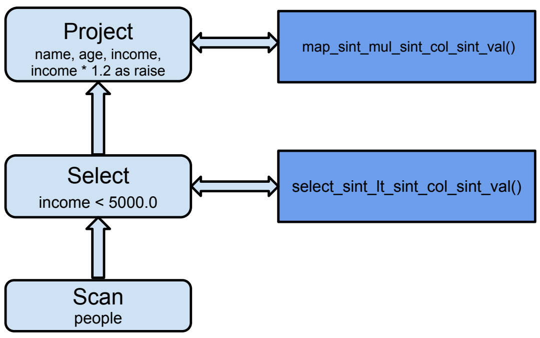
Project
name, age, income, income * 1.2 as raise
map_sint_mul_sint_col_sint_val()
Select
income < 5000.0
select_sint_lt_sint_col_sint_val()
Scan
people

Figure 2.2: Example of Vectorwise execution of a SQL query. Light blue objects represent operators while dark blue objects represent primitives.

2.7 Hash join

A Join operation combines tuples from multiple relations based on some condition. An Equi-Join is a Join operation for which the condition is expressed only with equalities between attributes. One algorithm to implement the Equi-Join is the HashJoin.

A HashJoin operation is performed in 2 phases. In the *build* phase, a hash table is constructed with all the tuples from the smaller relation (to minimize memory usage), indexed on the attribute that is part of the Join condition, called the key. The second phase is the *probe* phase, in which the tuples from the other relation are checked against the hash table.

2.7.1 Bloom filter

Hash table lookups are expensive because of the random memory access pattern that causes many cache misses. Because of this, a bloom filter (BF) is used to discard some tuples before they are checked against the hash table.

The bloom filter is a compact, randomized data structure used to perform set membership tests. It can produce false positives, with a known probability, so hash table lookups will always be needed to produce the final result. However, it cannot produce false negatives, i.e. if the check routine returns false, then it is certain that the key does not exist.

A bloom filter consists of a *bit map* and a set of K hash functions. The bit map has constant size, determined by the bloom filter *width*. The width is the number of bits per key. To store a set of n keys, we can use a bit map of size m. $\frac{m}{n}$ is the width of the bloom filter.

The false positive rate (FPR) decreases when increasing the width. Every false positive will be checked against the hash table, so decreasing the FPR will increase the performance. On the other hand, increasing the width will be detrimental to performance because of more data and TLB cache misses.

Each hash function maps a key to a position in the bit map. Inserts are done by setting the K bits obtained by hashing the key with the different hash functions. The FPR also decreases when increasing the number of hash functions. Of course, with more hash functions, we will have more computations and more memory operations.

One of the big advantages of bloom filters is that they allow for this trade off between a low FPR and a low membership check cost. By default, Vectorwise uses a width of 5 bits and 1 or 2 hash functions. For a small number of keys (below some threshold), Vectorwise uses only 1 hash function. The checks against this BF are considerably faster than those for the 2 hash function BF, which is used for most relations. As an optimization, the 2 hash functions are designed to map a key to 2 positions inside the same 64-bits word, which allows for more efficient memory loads and guarantees that the positions are also in the same cache line.

2.8 Summary

In this chapter we briefly described the Vectorwise execution model. We introduced terms that will be used throughout the thesis, such as primitive, primitive instances, primitive flavors, operators, vectors, etc. This chapter also described the bloom filter data structure which greatly speeds up hash joins.

Chapter 3

Micro Adaptivity Opportunities

We define *micro adaptivity* as the property of a DBMS to continuously choose between primitive flavors at runtime with the goal of minimizing the overall query execution time, choosing the most promising flavor based on real time statistics, such as the average number of clock cycles per processed tuple. This type of adaptivity, implemented in the query *execution* engine, is different than methods such as those presented in [DIR07], [AH00] which work by tuning the execution plan. Both types of adaptivity have the same goal: enable the DBMS to adapt to different platforms and data streams. The other adaptivity methods attempt to achieve this at a higher level by tuning the plan during execution while micro adaptivity does it at a lowest level by swapping primitive flavors.

This section compares different primitive flavors and shows that their performance can vary for reasons that are hard to predict, so an empirical micro adaptivity framework has the potential to reap the benefits of some, possibly obscure, optimizations that might not work in most common scenarios.

3.1 Levels

Looking at how primitive flavor performances changes we can identify different levels of micro adaptivity in a DBMS.

Platform adaptivity is when a flavor of a primitive is always the best on a certain platform. In this case our system should quickly discover this flavor and switch to it. We have observed this kind of adaptivity with compiler-flavors, presented in Section 3.5.

Instance adaptivity occurs when a flavor is best for a primitive instance. The reason why this could happen is the difference in data streams that are processed by the instances. For example, the streams can have different selectivities.

Call adaptivity is the most challenging case and a big reason why micro adaptivity is needed. In this scenario, the best flavor can change even for the same primitive instance. These flavors are influenced by data stream characteristics, and possibly machine state (which also change within the same instance).

Name	Architecture	RAM Size	LLC[1] Size	Branch Misprediction Penalty
Machine 1	Nehalem	48GB	12MB	17 cycles
Machine 2	Core2	8GB	4MB	15 cycles
Machine 3	AMD K8	62GB	1MB	12 cycles
Machine 4	Sandy Bridge	16GB	8MB	15 cycles

Table 3.1: Test machine details

3.2 Experimental setup

For the experiments discussed in the next sections we used micro benchmarks and TPC-H benchmarks on 4 machines with characteristics shown in Table 3.1. The reason for choosing these machines was to have different architectures. Machines 1 and 3 are used for most TPC-H experiments because of their bigger memory size.

3.3 Performance graphs

During execution of a query some primitives are called many times. It would be nice to be able to generate a graph of their performance history for that query. Unfortunately, there can be millions of calls, so it is not ideal to record a value for every one of them. Instead, we build an approximated performance history (APH) with only a few values. In this thesis we used 512 values. We found these graphs very useful when trying to understand the different types of adaptivity.

Each APH value represents the average of a number (power of 2) of consecutive primitive call performances. We call these values bins because they hold information from multiple calls. Figure 3.1 shows how an APH is built for 4 bins. Initially, the bins are empty and their size is 1, i.e. they can only hold information about 1 primitive call. The performance of the first primitive call is added to the first bin, the second to the second bin and so on until we reach the 5th call for which there is no empty bin. At this point, the array of bins is shrunk and the bin size is increased to 2. To shrink, values from 2 consecutive bins are added together in one bin. This leaves half of the bins completely full and half of the bins completely empty. The next calls will fill the second half of the bins at which point another shrinking occurs which will increase the size of the bins to 4.

For example, if there are 10K calls (like in Figure 3.2), each value in the approximated graph is the average of 32 actual values. The APH for this primitive is shown in Figure 3.3 and uses 312 bins.

3.4 Control vs. data dependency

In Section 1.1.1 we saw how the same primitive can be implemented with branches and without branches. The branching implementation has a *control dependency* because some instructions are executed depending on the outcome of previous instructions (in this case, the conditional branch is the previous instruction). The non-branching implementation has a *data dependency*, i.e. the outcome depends on data.

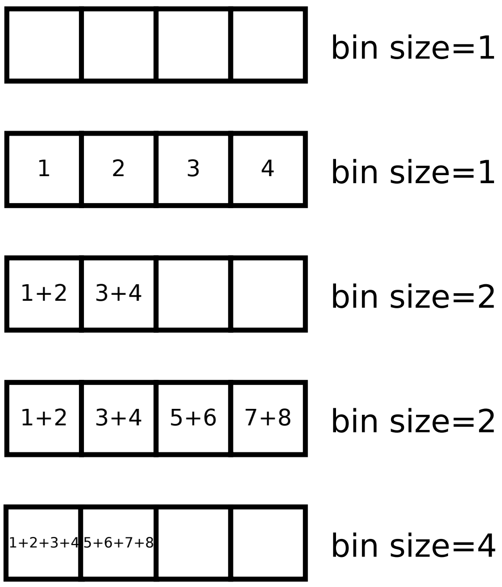

Figure 3.1: Approximated Performance Graph build process for the 8 primitive calls

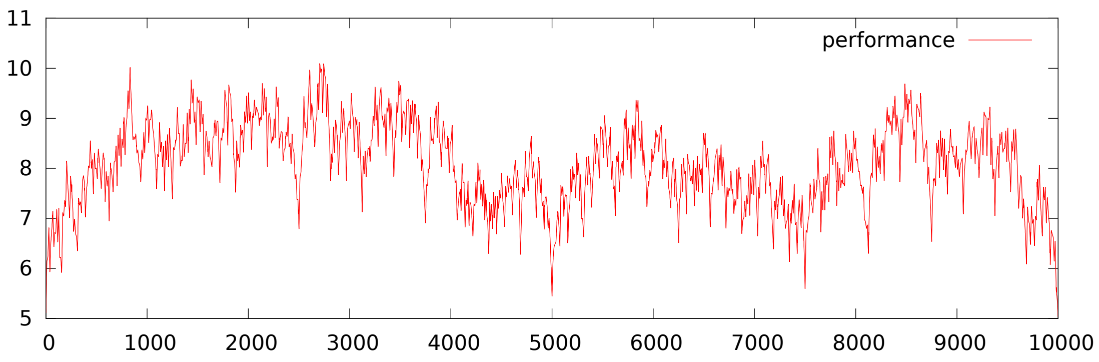

Figure 3.2: Original primitive call performance graph

3.4.1 Branch prediction

Current CPUs use pipelined execution to increase instruction throughput. There can be many instructions in the pipeline at a time, in different stages (e.g. one instruction could be in the fetch stage while another is in the execute stage, where fetching means loading an instruction from memory and executing means performing the actual work, like doing an arithmetic operation).

To fill the pipeline, processors speculatively execute future instructions (those which do not depend on the result of earlier instructions). A major obstacle for this model is *conditional branching*. The processor has to guess which way a branch goes in order to add future instructions to the pipeline. If the guess is correct, everything continues normally. But, if the guess is incorrect, then the wrong instructions have to be flushed from the pipeline, causing a delay. The unit responsible for these guesses is called the *branch predictor* and it uses the history of a branch to predict its outcome.

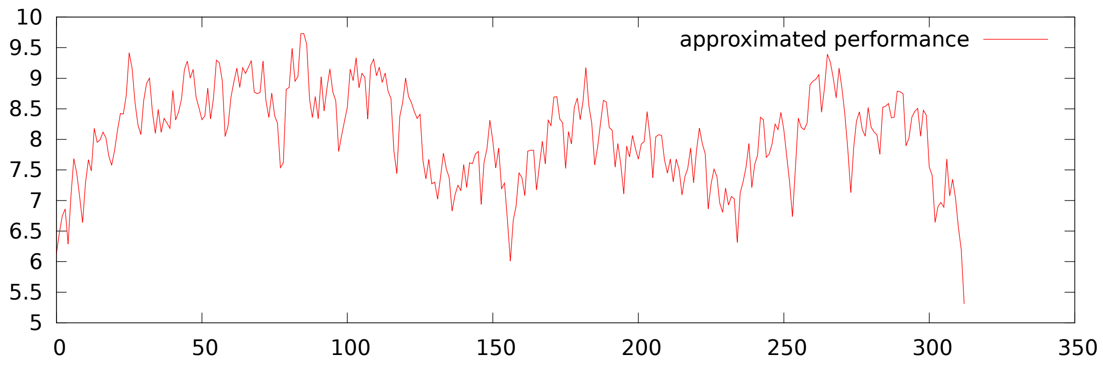

Figure 3.3: Approximated primitive call performance graph

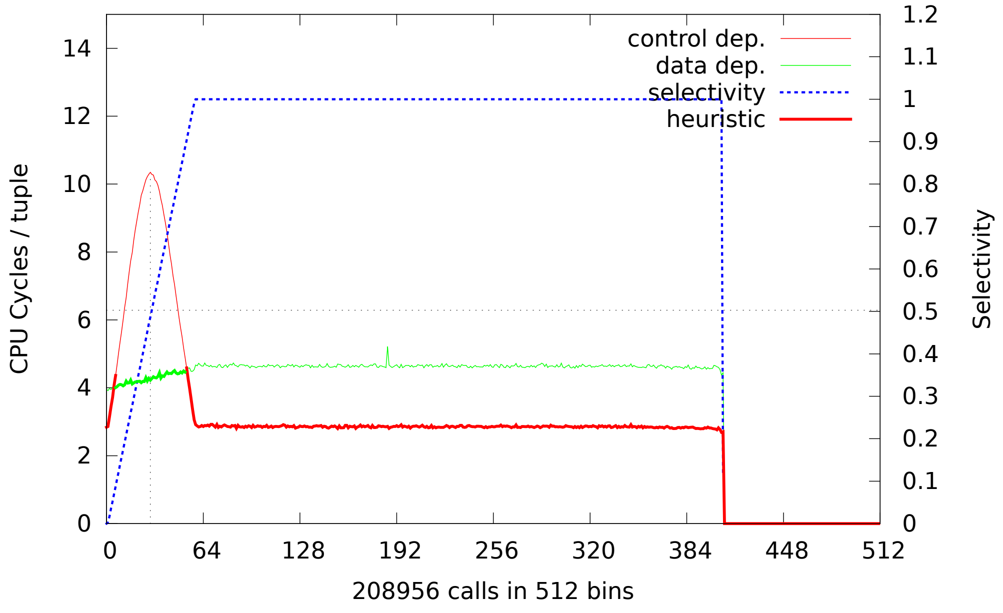

Figure 3.4: Selection primitive with and without branches on Machine 3, TPC-H Query 7

3.4.2 Selectivity

Data selectivity influences the effectiveness of the branch predictor. If, due to low(high) selectivity, a branch is almost never(always) taken, then the predictor will easily notice this and the processor will execute future instructions. However, a selectivity of 50%, for example, makes it hard to predict the branch outcome. This is because the predictor tries to find a repeating pattern in the branch outcome and such a pattern might not exist in the data that is processed. If this happens, there will be a lot of mispredictions which cause extra delays. [Fog12] contains more information about branch predictors in different architectures.

By removing the branch, we stop relying on the branch predictor, but we can potentially introduce more instructions, which might hurt performance.

Figure 3.4 and Figure 3.5 show performance graphs for the primitive *select_int_ge_int_col_-int_val* called for TPC-H Query 7. This primitive selects tuples with an attribute value greater than or equal to some constant. The data dependency flavor has a constant performance while the control dependency flavor has an irregularity in the beginning of the query. The dotted blue line shows how the data selectivity varies for this primitive. The selectivity increases from

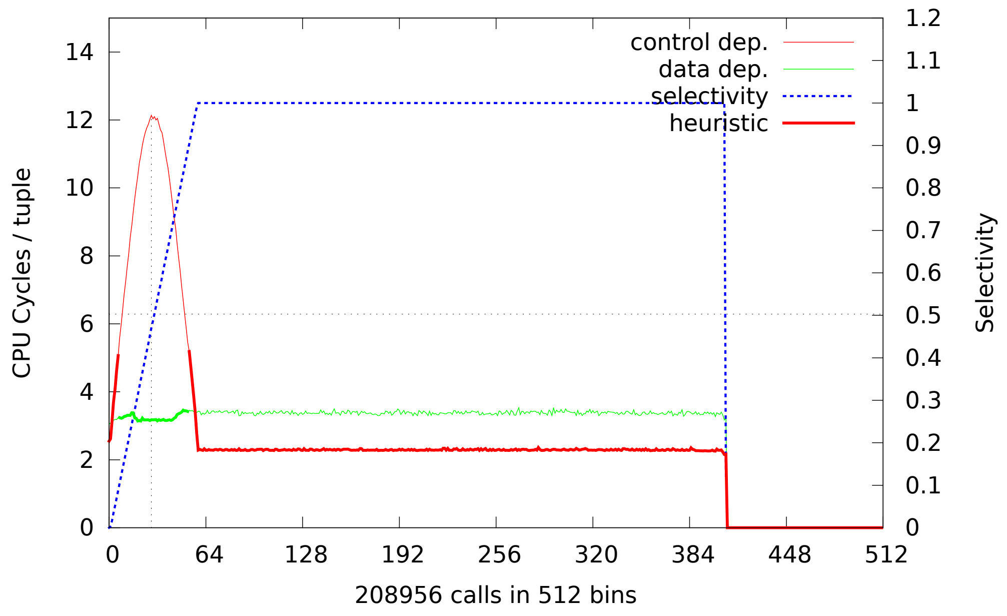

Figure 3.5: Selection primitive with and without branches on Machine 1, TPC-H Query 7

0.0 to 1.0 in the first 30000 calls. For these calls, the control dependency flavor performs worse than the data dependency flavor. The performance degrades as the selectivity increases from 0.0 to 0.5, where it achieves the worst performance. After this, it starts to improve and when the selectivity reaches 1.0 the control dependency flavor becomes the fastest and maintains a constant number of CPU cycles / tuple until the end of the query.

3.4.3 Heuristic

Vectorwise uses a configurable threshold for enabling data dependency. By default, if the selectivity of the previous primitive call is between 0.1 and 0.9 then data dependency is enabled for the next call. This is illustrated in Figure 3.4 and Figure 3.5 with thick lines. The heuristic works well in this case because the selectivity goes from 0.0 to 1.0 quickly. So, even if the heuristic is not correct for some selectivities, the impact on performance is negligible. However, we can see from Figure 3.5 that the heuristic is not perfect. On this machine, it would have been better to use *data dependency* for some selectivities greater than 0.9. The two machines used in these tests have significantly different branch misprediction penalties. Machine 1 (Nehalem architecture) has a penalty of around 17 cycles while Machine 3 (AMD K8) has a penalty of only 12 cycles. Thus, if the heuristic causes extra mispredictions, the effect will be amplified on Machine 1.

3.4.4 Summary

This section presented a case where two flavors have different performance due to changes in data characteristics. This is related to a hardware feature called branch prediction which does not work so well on some data patterns. The current heuristic used by Vectorwise uses static thresholds and seems to work well on the TPC-H benchmark. However, close analysis shows that this heuristic can be sub-optimal due to the hard-coded parameters, so replacing it with micro adaptivity should make performance more robust (good performance every time).

3.5 Compilers

Compilers use various heuristics to optimize the code. Vectorwise is compiled on one machine and those binaries are distributed to customers, which could be running them on any hardware. So, it is not easy to predict which compiler or flags are optimal. Optimizations made by a compiler could work well on some platforms but not on others.

To investigate this, we built 3 versions of the Vectorwise engine using icc, gcc and clang compilers. We ran the TPC-H benchmark for each of them and recorded profile data. We use the term compiler-flavor to refer to flavors obtained by changing the compilation process (i.e. using different compilers and/or different optimization flags). The compilation flags used for generating these flavors are those used by the standard Vectorwise build process. The optimization level is O3 for all compilers and SIMD code generation is enabled. At this stage we did not investigate the effects of compilation flags (except for loop unrolling which is presented in Section 3.7), but this could be part of future research.

3.5.1 Platform adaptivity

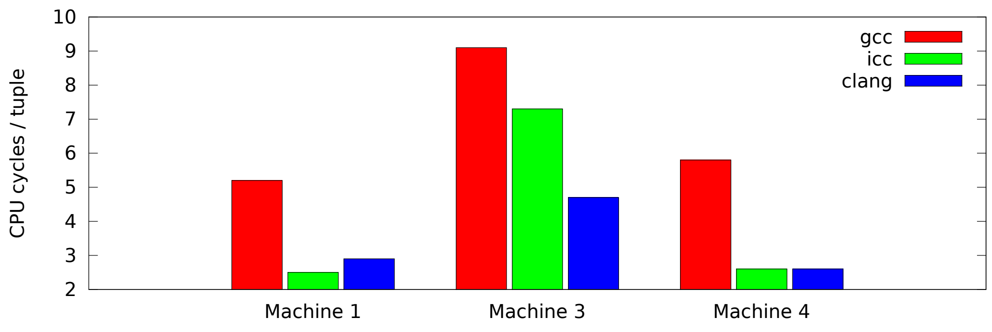

Figure 3.6: Platform adaptivity example with merge join primitive called in Query 2. The fastest flavor depends on the machine.

In Figure 3.6 we plotted the average clock cycles spent per tuple for a primitive that performs a merge join, called when executing TPC-H Query 2. Here, the gcc flavor is the slowest on all 3 machines. However, the fastest flavor is either clang or icc, depending on the machine. On Machine 3, it is clang while on Machine 2 it is icc. The reason for this might be the fact that Machine 2 has an Intel CPU while Machine 3 has an AMD CPU. The icc compiler is tuned for Intel hardware, so it should not be surprising if it produces faster flavors[1].

3.5.2 Instance adaptivity

The `select_==_sint_col_sint_val` primitive is called while executing TPC-H Queries 2 and 21. Figure 3.7 shows how the 3 compiler flavors perform in these two instances. For Query 2, the gcc flavor is the fastest, followed by icc and then clang while for Query 21 clang is the fastest, followed by icc and then gcc. So, even on the same machine, it is unknown which flavor is the best.

[1]In fact, icc is infamous for producing binaries that are slow on AMD CPUs [Fog09].

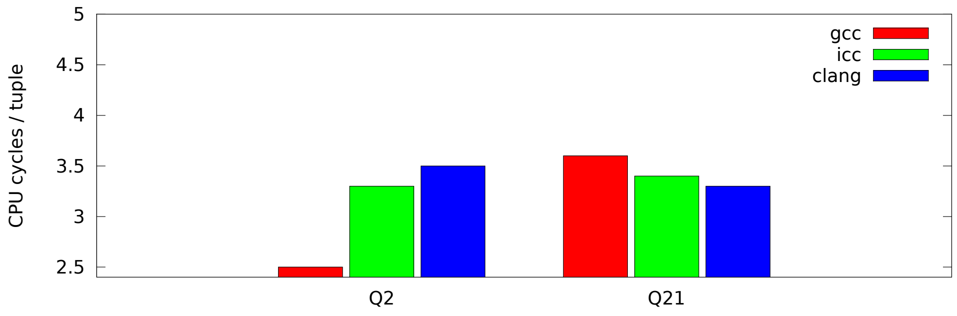

Figure 3.7: Instance adaptivity example with merge join primitive. On the same platform (Machine 3), the fastest flavor depends on the primitive instance

3.5.3 Call adaptivity

An even more interesting case is when the input data processed by a primitive instance changes in such a way that it triggers different performance changes for each flavor. An example of this (Figure 3.8) is an instance of the `select_==_sint_col_sint_val` primitive, called in Query 21. clang is the best flavor until there is some change in the input data and gcc becomes better.

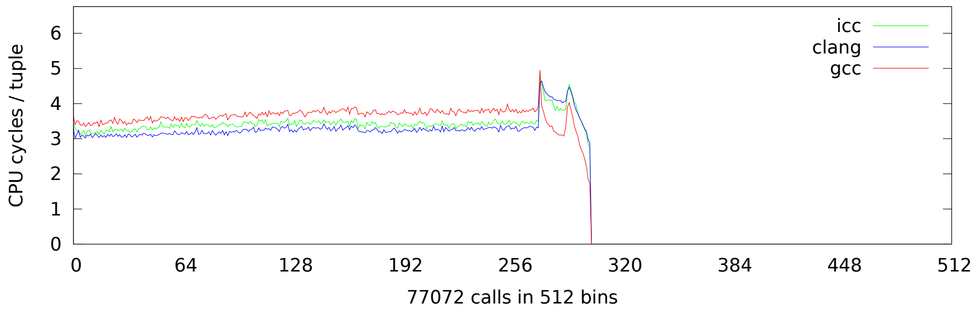

Figure 3.8: Call adaptivity example with selection primitive called in TPC-H Query 21. The performance ranking of the flavors changes at some point in the query.

3.5.4 Summary

This section presented one opportunity to improve performance by simply building multiple versions of the database system with different compilers. While we might be able to determine a priori which flavor is best for a given hardware platform, to fully exploit this we need to have an adaptive system that also works when performance is influenced by the data that is processed.

3.6 Loop fission

In this section we look at another optimization that can influence performance. The loop fission optimization is done by splitting a loop into multiple loops over the same range. The goal is to improve locality of reference. The code in Listing 3.1 does not use the cache as efficiently as the equivalent code in Listing 3.2. In the first example, both a and b arrays should be prefetched into the cache at the same time to maximize performance, while in the second example only one array has to be prefetched. And, as we will see in this section, another reason for loop fission is to remove data dependencies that might stand in the way of generating parallel memory loads.

```
for(i = 0; i < n; ++i){
        s += a[i] * 2 + b[i] * 3
}
```

Listing 3.1: Example of loop without fission

```
for(i = 0; i < n; ++i){
        s += a[i] * 2;
}
for(i = 0; i < n; ++i){
        s += b[i] * 3;
}
```

Listing 3.2: Example of loop with fission

3.6.1 In bloom filter

The bloom filter (see Section 2.7) supports membership tests through a collection of check primitives, one for each key type. A simplified version of the check primitive for 1 bit bloom filters is shown below.

```
size_t sel_bitfiltercheck(size_t n, size_t* res, uchr* bitmap, ulng* keys) {
    ulng ret = 0;
    size_t i;
    for (i = 0; i < n; ++i) {
        slng hv = bf_hash(keys[i]);
        res[ret] = i;
        ret += bf_get(bitmap, hv);
    }
}
```

Listing 3.3: Bloom filter check primitive

Each key is first hashed and the hash value is then mapped to a bit position in the bloom filter bit map. If that bit is set, the check is positive and the index of this key is added to the result. Normally, there would be a branch in the loop, $if(bf\_get(bitmap, hv))$, but, as in many other places, Vectorwise replaces it with data dependent code.

This simple function can be made significantly faster in some cases by breaking the loop body in two. The first loop will query the bloom filter and store the results in a temporary array. The second loop will fill the result vector using the data from the temporary array.

```
size_t sel_bitfiltercheck2pass(size_t n, size_t* res, uchr* bitmap, ulng* keys) {
    ulng ret = 0;
    size_t i;
    for (i = 0; i < n; ++i) {
        slng hv = bf_hash(keys[i]);
        tmp[i] = bf_get(bitmap, hv);
    }
    for (i = 0; i < n; ++i) {
        res[ret] = i;
        ret += tmp[i];
    }
}
```

Listing 3.4: Bloom filter check primitive with fission

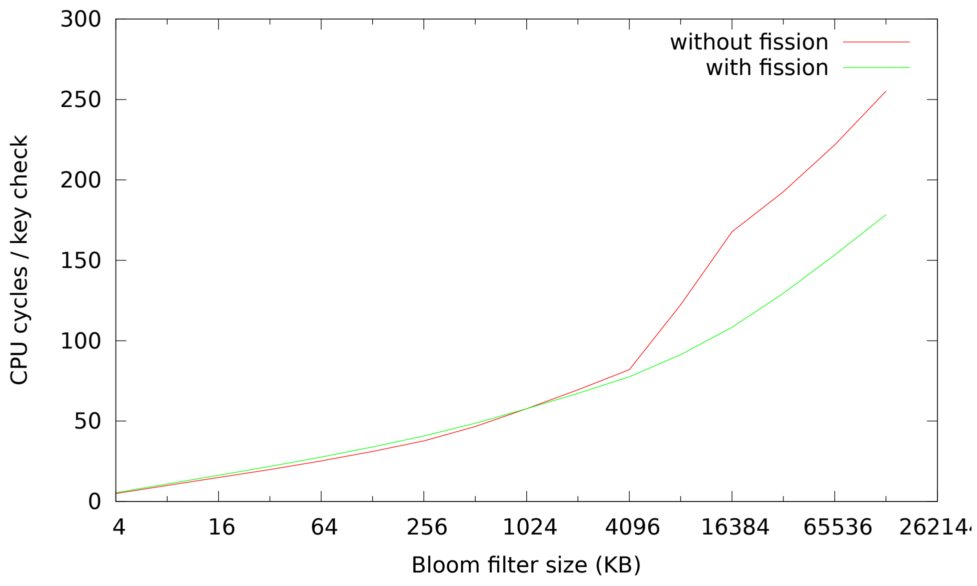

Figure 3.9: Bloom filter check with/without fission on Machine 1

Figure 3.9 shows the performance comparison between the two versions in terms of clock cycles / key processed. For this experiment we varied the number of build keys from 2^{12} to 2^{27}, which required bloom filters with sizes from 4KB to 131072KB.

We can see that for large bloom filters the fission code performs much better, sometimes 50% faster. We can also see that for small bloom filters, the fission code can actually be slower, sometimes by 10%. Bloom filter checks are often one of the most time consuming operations in a query with hash joins, so even small improvements here will be noticeable on the overall query time.

To understand why the fission version is faster, let us look at the memory operations done by the non fission code. *bf_get loads* from a random memory address and it will be the slowest of all the memory operations due to many cache misses. $keys[i]$ and $res[ret]$ are accessed sequentially, so the cache hit rate will be high for them. The real culprit here is the *ret* variable, which causes a dependency between the $res[ret]$ *store* and the *load* in *bf_get*. The *store* instruction cannot start until the *loads* from all the previous iterations are complete. In contrast, for the first loop of the fissioned code, the store/load pairs are independent and can be executed in parallel. *ret* still causes a dependency in the second loop, but now the load operation is much faster, since the temporary array is small and fits in the cache.

In conclusion, the fission code is faster because the dependency introduced by *ret* is now between memory operations on cache resident arrays. This also suggests that there is no benefit from fission when the bloom filter bit map is so small that it fits in the cache. Figure 3.9 supports this theory. We see that fission is actually a little slower for bloom filters smaller than 4096KB. For these small bloom filters, fission is slower, because it executes more instructions than the normal version, without gaining anything.

Figure 3.10 shows the speedup obtained by loop fission on different machines. For example, we can see that on Machine 4 there is speedup later than on Machine 3.

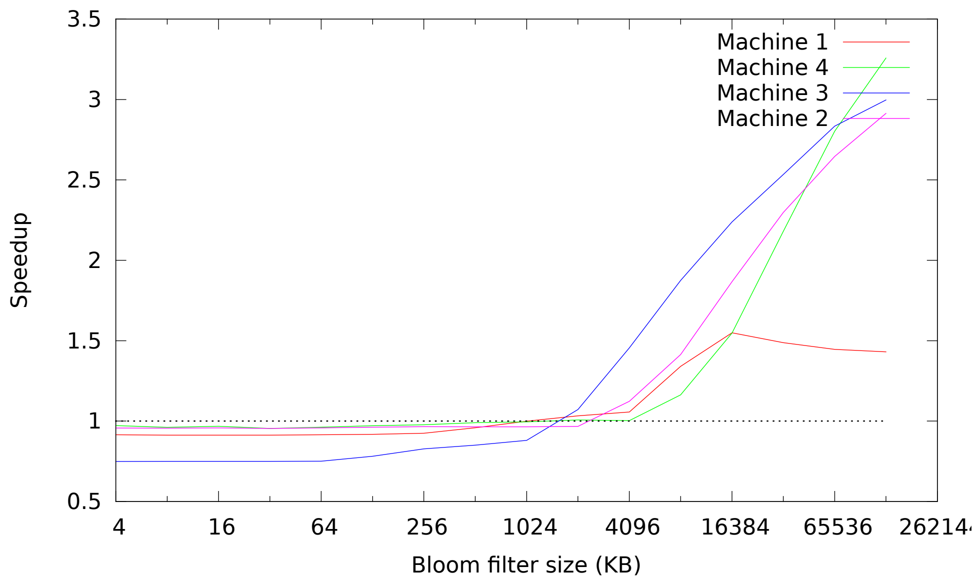

Figure 3.10: Bloom filter check with/without fission speedup on different machines

3.6.2 Summary

This section analyzed loop fission inside the bloom filter check primitive. In micro benchmarks we observed that the efficiency of this optimization depends on the size of the bloom filter. We also saw that the threshold size after which it becomes efficient depends slightly on the platform.

3.7 Loop unrolling

Many Vectorwise primitives consist of loops with only a few instructions. Consider the following pseudo-assembly code for the integer multiplication primitive in Listing 2.1.

```
loop:
        LOAD col1[i]
        LOAD col2[i]
        MULTIPLY res[i], col1[i], col2[i]
        STORE res[i]
        INCREMENT i
        COMPARE i, n
        JUMP loop // jump if i < n
```

Listing 3.5: Multiplication primitive without loop unrolling

There are 4 instructions that are needed for computing 1 result element (2 loads, 1 multiplication and 1 store) and there are 3 extra instructions for loop control (increment, compare and jump). In total, it takes 7 instructions to compute 1 result element.

Loop unrolling [HP06] is a technique for reducing the loop control overhead. It can be applied manually, by transforming the code, or automatically by an optimizing compiler. The idea is to replicate the loop body a number of times (called unroll factor). The code below shows how we unrolled the previous loop by a factor of 2.

```
loop:
        LOAD col1[i]
        LOAD col2[i]
        MULTIPLY res[i], col1[i], col2[i]
        STORE res[i]
        INCREMENT i

        LOAD col1[i]
        LOAD col2[i]
        MULTIPLY res[i], col1[i], col2[i]
        STORE res[i]
        INCREMENT i

        COMPARE i, n
        JUMP loop // jump if i < n
```

Listing 3.6: Multiplication primitive with loop unrolling of 2

The new loop needs 6 instructions per result element, compared to the 7 of the original loop. If we unroll by a factor of 8, the loop will need an average of 5.25 instructions per result element. This, of course, is a big advantage, but it is not the only one. By unrolling, we also reduce the number of branches, so the processor can now use out of order execution more efficiently, by combining instructions from different loop iterations.

One disadvantage of loop unrolling is that it increases the code size. Thus, an unrolled loop will cause more instruction cache misses. Code size increases with unroll factor so, there will be a point after which unrolling no longer pays off. But even a small unroll factor can be detrimental to performance because it might prevent the compiler from generating SIMD code.

3.7.1 Side effects

Many Vectorwise primitives are manually unrolled with a factor of 8. To check if this can have side effects, we tested the primitive from Listing 2.1 with and without manual unrolling as well as with and without compiler auto unrolling. The code was compiled with gcc using the default Vectorwise flags, which include **ftree-vectorize** which enables auto vectorization of loops as well as **funroll-loops** which enables auto loop unrolling. Section 3.7.1 shows the average cycles per tuple processed by the primitive.

For this experiment, the best performance is obtained when manual unrolling is disabled but compiler unrolling is enabled. In this case, gcc unrolls the loop and also generates SIMD code. This makes it achieve close to 1 cycle per tuple. The next best performance is obtained by disabling both compiler and manual unrolling, but leaving vectorization. In this case the compiler only generates SIMD code, which is around 15% slower without unrolling. If manual unrolling is used, gcc no longer generates SIMD code for the loop. In this case an average of 1.73 cycles is used for a tuple. So, for this primitive manual unrolling hurts performance by almost 70%.

Table 3.2: Cycles per tuple with/without compiler and manual loop unrolling on Machine 1

Compiler / Manual	On		Off	
	SIMD	no SIMD	SIMD	no SIMD
On	1.73	1.73	1.73	1.73
Off	1.03	1.74	1.18	2.59

Running the same binary on Machine 3 produces slightly different results (Table 3.3). The manually unrolled version still has constant performance, since the compiler generates the same code for it in every case. However, now the manually unrolled version is the fastest. The second best performance is obtained when auto unrolling is on but vectorization is off. On this machine it seems SIMD code is considerably slower than unrolled code.

Table 3.3: Cycles per tuple with/without compiler and manual loop unrolling on Machine 3

Compiler / Manual	On		Off	
	SIMD	no SIMD	SIMD	no SIMD
On	2.02	2.02	2.02	2.02
Off	3.61	2.15	3.55	4.03

3.7.2 Summary

This section showed another case when a single implementation is not optimal in all cases. We observed that on one machine manual loop unrolling hurts performance while on another machine it improves it.

3.8 Full computation

Many primitives in Vectorwise accept a *selection* vector argument (e.g. Listing 2.3, Section 2.5.2). It contains the indices of the tuples that need to be processed by the primitive. For example, a selection primitive creates the selection vector shown in Figure 3.11 which will then be passed to a multiplication primitive that does some computation on the selected tuples. Only the required tuples are processed (marked here with white background) and the result vector will have undefined values in the positions not in the selection vector (grey background).

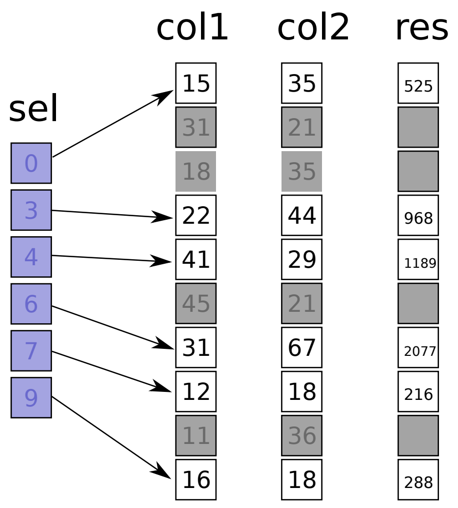

Figure 3.11: Selective computation

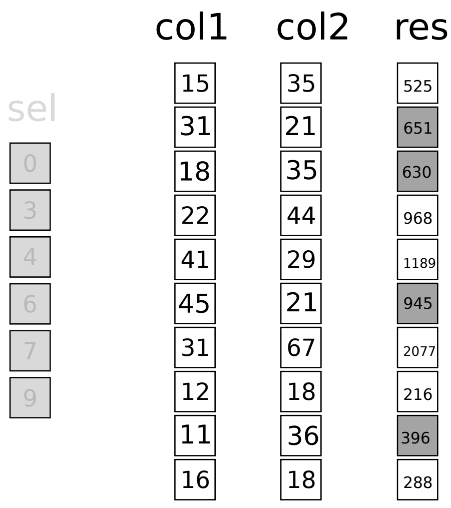

Figure 3.12: Full computation

For some operations it is possible to ignore the selection vector and still produce correct results. The multiplication primitive in Figure 3.12 processes all tuples, even though some of the results are not needed. This turns out to open possibilities for performance improvements.

For the experiments presented in this section, we used a vector size of 1024 and varied the selection vector size from 64 to 1024. The selectivity axis, expressed in percentages, represents the ratio $\frac{selection\ vector\ size}{vector\ size}$. The selection vector is filled with unique random positions (between 0 and 1023) and then the primitive is called with and without full computation.

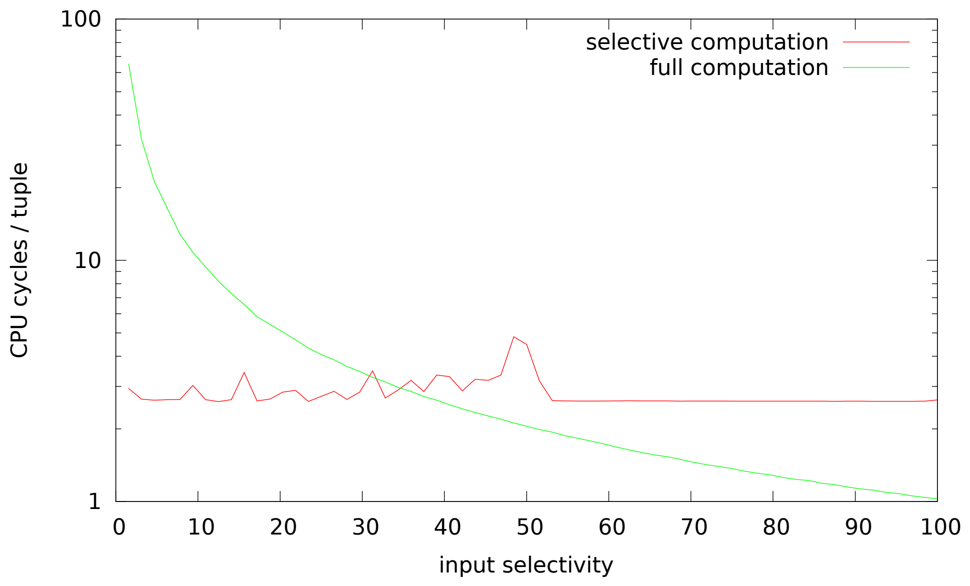

Figure 3.13: Micro benchmark showing full computation benefit with SIMD

3.8.1 With SIMD

Without the selection vector, the compiler can generate SIMD code which has lower cost per operation and might be faster overall, despite doing more work. Figure 3.13 shows the impact of full computation in a micro benchmark that uses the *map_int_mul_int_col_int_col* map primitive with SIMD code generation enabled.

Full computation always does the same amount of work (it processes the whole vector) while selective computation will only do as much work as needed, according to the selection vector. Thus, it is expected that when the selection vector is small, full computation will not be efficient. But, there is a point after which it does become efficient. In Figure 3.13 this happens when the selectivity goes above 30%.

3.8.2 Without SIMD

Not all primitives can be SIMDized (e.g. those that compute division, trigonometric functions). Furthermore, some operations need too many CPU cycles which makes it impossible to amortize the overhead of full computation. We can see this in Figure 3.14. It shows the speedup of primitives which compute 4 byte integer multiplication (SIMD disabled), 4 byte integer division and 8 byte integer division. The integer multiplication primitive, for which we disabled SIMD code generation (with **-fno-tree-vectorize** compiler flag) still gains some benefit from full computation. This happens for higher selectivities than with SIMD and is likely because full computation does not use the selection vector so there is less pressure on the cache and fewer instructions per tuple. However, for the other two primitives, there is little benefit, at selectivities near 1.0.

3.8.3 Adaptivity

The selectivity values for which full computation is faster are not the same on all test machines. Figure 3.15 shows that the threshold after which full computation performs better ($speedup > 1$)

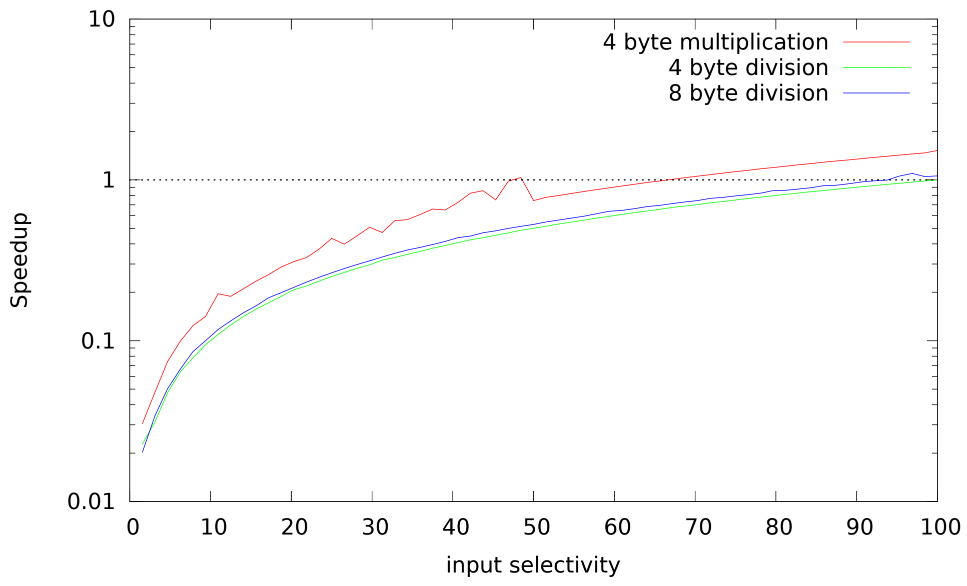

Figure 3.14: Micro benchmark showing full computation benefit without SIMD

is different on each machine. Calculating this threshold is difficult because it could depend on a number of factors, such as SIMD hardware performance or memory speed.

The benefit from full computation will also depend on the data type size, as the SIMD registers have fixed sizes. This can be seen clearly in Figure 3.16. For 2 byte multiplication, there is speedup as soon as the selectivity passes 10%, while for 8 byte multiplication there is no benefit at all.

3.8.4 Heuristic

Vectorwise currently does full computation, inside primitives which can be SIMDized, when the condition $n*16 > vector\_size*width$ is true. **n** is the size of the selection vector, $vector\_size$ is the size of the input vector (1024, in these experiments) and $width$ is width of the input data types. The heuristic is based on the observation that SIMD has higher benefit for smaller data types. Since this expression evaluates to the same number on all platforms, the current heuristic might be completely inappropriate for some machines. The vertical line in Figure 3.15 marks the point after which full computation is enabled by the heuristic. As it can be noticed, this happens much too early for some machines, causing a slowdown.

3.8.5 Summary

This section presented an optimization called full computation which is beneficial for some primitives (especially those that can be SIMDized). The benefit can be substantial but the point when this optimization should be enabled depends on the platform. Because of this, the current mechanism is sometimes sub-optimal and a more generic method is needed.

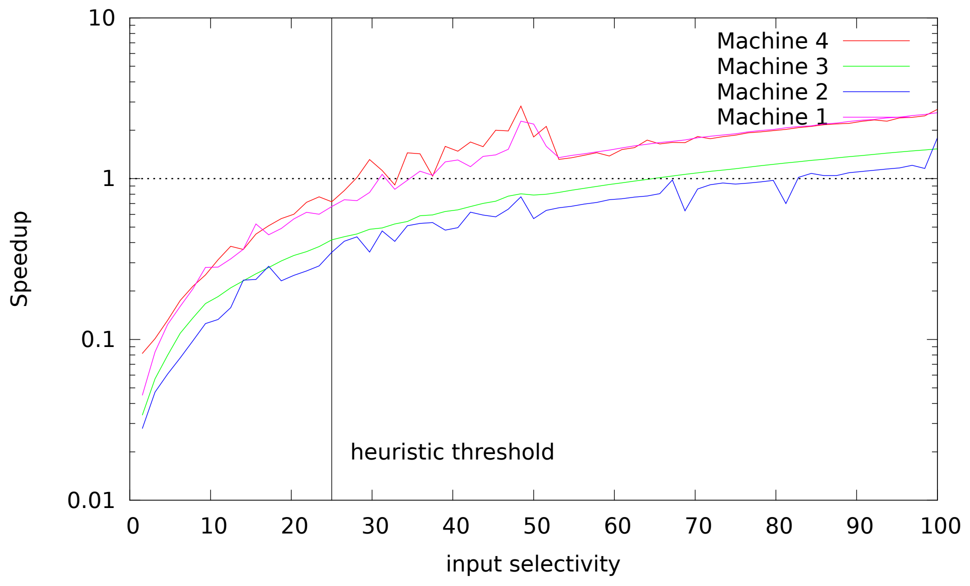

Figure 3.15: Micro benchmark showing full computation speedup on different machines.

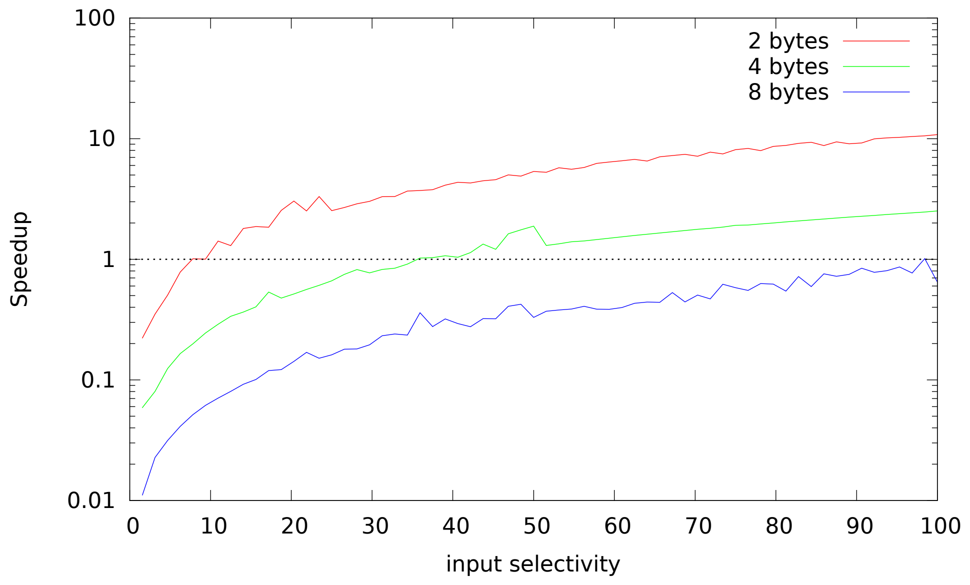

Figure 3.16: Full computation with different data types

Chapter 4

Micro Adaptivity System

We designed and implemented micro adaptivity system within the Vectorwise query execution engine. A description of the system architecture with details about the flavor library management is given in Section 4.1. Section 4.2 presents the machine learning problem related to micro adaptivity along with our proposed solution. We conclude the chapter with an evaluation of the proposed solution on data gathered from the TPC-H benchmark.

4.1 Architecture

This section describes our approach to generating, storing and loading primitive flavors. We also show how the micro adaptivity subsystem is integrated into the Vectorwise query execution architecture.

4.1.1 Generating flavors

Vectorwise primitives are written in a macro language called Mx [KSvdBB96], which was developed for MonetDB [Bon02]. Each primitive has a template which is expanded to C code before the actual compilation. Listing 4.2 shows the template used to generate the code for `map` primitives that apply some operation on two columns. In a source file that contains primitive implementations, the macro `impl_op_binary` will be called with 4 parameters that are used to expand `@1, @2, @3, @4` and `@5`. In this particular example, `@1` represents the name of the operation which will be included in the primitive signature but also the name of a macro/-function that performs the actual computation. Listing 4.1 shows such a macro, `int_add`. `@2` and `@3` represent the types of the two operands and `@4` is the type of the result. Invoking the macro as `impl_op_binary(int_add, int, int, long)` will generate code for a primitive that adds two **`int`** vectors and produces a vector of **`long`**s. The `@impl` macro is used to reshape the code. In this example, it generates code that checks if the selection vector argument is `null` or not and calls the corresponding macros (`implnosel` or `implsel`) that produce the final code. The actual `@impl`, used in Vectorwise, also contains code for the full computation heuristic and for loop unrolling. This is omitted here.

```
@:def(int_add,(r,x,y), r=((x)+(y)))@
@= implnosel
{
        size_t i=0;
        for(i=0; i<n; i++) { @1; }
}
@= implsel
{
        size_t i=0,j=0;
        for(j=0; j<n; j++) { i = sel[j]; @1; }
}
@= impl
{
        if(sel){
                @:implsel(@1)@
        } else {
                @:implnosel(@1)@
        }
}
```

Listing 4.1: Mx macros used by all primitive templates

```
@=impl_op_binary
        size_t map_@1_@2_col_@3_col(size_t n, @4* res, @2* col1, @3* col2, size_t*
            sel)
        {
                @:impl(@1(res[i], col1[i],col2[i]))@
        }
```

Listing 4.2: Generic binary operation primitive written in Mx. This `impl_op_binary` macro can be used to generate primitives that perform some operation on two columns

Using templates it is easy to write and maintain code for a large collection of primitives but it also gives us a quick way to generate flavors. For example, in Listing 4.3 we slightly modified the `@impl` macro to generate full computation flavors for all primitives (see Section 3.8). If the option `FULL_COMPUTATION_FLAVOR` is active, then we set the size of the input vector to `sel[n-1]+1`, where `sel[n-1]` is the index (0-based) of the last tuple to be processed. Now, the primitive will process all tuples from 0 to `sel[n-1]`. Additionally, we set `sel` to `null` so the branch without selection vector is taken.

```
@= impl
{
        #ifdef FULL_COMPUTATION_FLAVOR
                if(sel)
                {
                        n = sel[n-1] + 1;
                        sel = NULL;
                }
        #endif
        if(sel){
                @:implsel(@1)@
        } else {
                @:implnosel(@1)@
        }
}
```

Listing 4.3: Modified `@impl` macro used to generate full computation flavors

4.1.2 Flavor libraries

Vectorwise maintains a dictionary with all the primitives (indexed by the signatures). A SQL query is translated to an operator tree and each operator calls one or more primitives to do the actual work. The primitives are referenced by their signature and a hash table is used to quickly find the function pointer of a primitive given its signature. We extended this mechanism to work with primitive flavors.

A *flavor library* is a collection of flavors for different primitives. In any library, there is at most one flavor for each Vectorwise primitive. For example, there is a flavor library called for full computation which contains all the Vectorwise primitives, but with full computation always enabled. This library, together with the full computation off library can be used to have a simple micro adaptive system that exploits the benefits of full computation, where they exist.

Each flavor library is stored as a dynamically linked library. To build the library, it is possible to use the standard Vectorwise build process. At the end, we simply link together the object files that contain primitive implementations, to create a dynamically linked library.

To activate a certain flavor we use the compiler `define` flag.
For example, passing `-DFULL_COMPUTATION_FLAVOR` to the compiler will activate the full computation flavor (see Listing 4.3). Additionally, the flavor library exposes a function, `flavor_name()`, which simply returns the name of the flavor. This is currently used for gathering profiling data, so we can find out which flavor is active at every primitive call. The value returned by this function is also set using the `define` flag (e.g. `-DFLAVOR=full_computation_on`).

4.1.3 Loading flavor libraries

Flavor libraries are loaded with the POSIX `dlopen` function during the Vectorwise kernel initialization, before any query is received. The same symbol (e.g. a function) can exist in more than one library, so, normally, it is uncertain how the symbol is resolved. Luckily, on Linux, `dlopen` can be called with the `RTLD_DEEPBIND` argument. With this, the symbols referenced by a flavor library are resolved by first looking inside the same library, whereas without the argument, symbols are resolved using previously loaded libraries. This argument is not standardized, so for other platforms, like Windows, we might need to find a different approach.

During initialization, Vectorwise will register all flavors found in the "flavors" directory. Registering a flavor means adding it to the global primitives dictionary, the same way primitives are currently registered. This mechanism is sufficient for implementing the two functions `get_random_flavor()` and `get_best_flavor()` needed for the adaptive optimizer, discussed in Section 4.2.

4.1.4 Adaptivity loop

The standard Vectorwise query execution process starts with the **parse** stage where an operator tree is constructed for the given query. Execution (pull-based) is done by repeatedly calling the `next()` method of the top-most operator. Each operator will first pull vectors of **tuples** from its children (by calling their `next()` methods) and then process them using the necessary primitives. To make the execution micro adaptive, we added an extra stage (Figure 4.1) which occurs after every primitive call. After each call, **performance** data is sent to the micro adaptive module which uses it to decide which flavor is the best one for that primitive and **switch** to it.

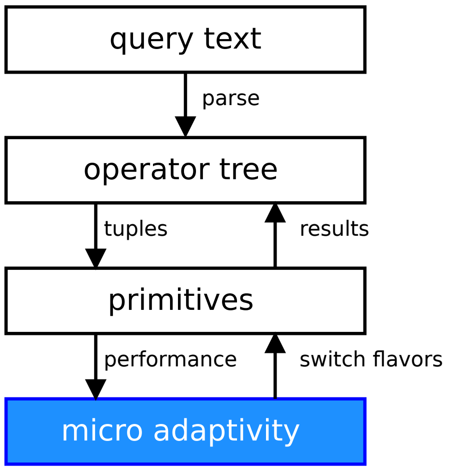

Figure 4.1: Adaptive Vectorwise query execution flow

4.1.5 Summary

This section presented practical aspects regarding the implementation of a micro adaptive system within Vectorwise. We showed how flavor libraries can be quickly generated using the Mx macro language. These are stored as dynamically linked libraries and loaded at runtime. Within the execution engine, micro adaptivity acts as a feedback loop, gathering information from primitives and changing their state based on this.

4.2 The optimization problem

The key component of our micro adaptivity system is what we call the **adaptive optimizer**. Its role is to select the best performing flavor for a given primitive instance. We saw that performance can be data and platform dependent, so supervised learning strategies (e.g. artificial neural networks) which rely on an off-line training phase are not a good fit for our optimizer because we will not be able to build a training data set that is representative for every possible workload. Optimization strategies such as genetic algorithms, hill climbing, simulated annealing, etc. might work well on platform adaptivity and instance adaptivity cases, where they can converge to the optimal flavor. However, in the case of instance adaptivity, when there could be different optimal flavors in different sections of the query, these algorithms will get stuck in one of them and fail to find the others. Therefore, we seek an algorithm with real-time learning, i.e. one that is able to adjust to sudden context changes.

Consider that for a given primitive there is a number of flavors available in the system. Before each expression evaluation, the optimizer is called to select the most promising of these flavors. We can say that each flavor brings a reward proportional to its performance. **These rewards are not constant and the reward of a flavor is unknown until the system actually calls that flavor and records its performance**. So, the most promising flavor, chosen by the optimizer, has to be the flavor that will lead to the maximum total reward. After each choice, the optimizer updates his knowledge about the flavors, so it is able to make better decisions in the future. This type of problem is called a *multi armed bandit problem* [Rob52].

Expected regret

Assume that the rewards of the K flavors of a primitive follow some probability distributions $R_1, R_2, ..., R_K$ and let $\mu_1, \mu_2, ..., \mu_K$ be the expected values of these distributions and $\mu^* = max_k\{\mu_k\}$ the maximum expected value. During the execution of a query, the system will make T flavor choices. At the end, we can compute the *total expected regret*, as $R_T = T * \mu^* - \sum_{t=1}^{T} \mu_{j(t)}$, where $j(t)$ is the index of the flavor that was actually chosen by the system at iteration t. The regret tells us how good our selection strategy is, i.e. the smaller the regret the better.

We can also express the regret as $R_T = T * \mu^* - \sum_{j=1}^{K} \mu_j \mathbb{E}[T(j)]$, where $T(j)$ is a random variable for the number of times that flavor j was chosen and $\mathbb{E}[T(j)]$ is the expected value of this number. For certain reward distributions, it can be proven that the regret grows at least logarithmically with the number of iterations, i.e. $R_T = \Omega(lnT)$ [LR85] .

Exploitation vs. exploration

After a number of iterations, the system will have some knowledge about the rewards of each flavor. Based on this, it can determine the best one to choose. But there is a hidden danger here. If the system keeps choosing the same flavor, it will build more knowledge about it and the knowledge about the other flavors will become stale. In the meantime contextual parameters that determine the performance of a primitive (e.g. selectivity, cache/memory traffic) may change, so another flavor could become the best one and our system will fail to switch to it. To overcome this, the optimizer should *sometimes* choose a flavor that is not optimal based on the knowledge so far. Often, this means choosing a random flavor. Of course, this cannot be done too often, because it is probable that the chosen flavor is indeed not optimal, so it will hurt performance. Using the knowledge gathered so far to choose the most promising flavor is called *exploitation* while choosing a random flavor to try and find new opportunities is called *exploration*. Thus, the key to the problem is to figure out how much exploration and how much exploitation to do.

Stationary case

Each flavor can be viewed as a random process with one random variable which represents the call performance. A random process is called *stationary* if its probability distribution does not change in time. Statistical properties such as mean or variance, if they exist, are constant in such a system. For this case, there are known algorithms that solve the MAB problem optimally [ACBF02], i.e. the regret increases logarithmically with the number of games.

Unfortunately, in our case, we cannot be certain that the flavors are stationary processes since the performance can be influenced by data, for example. This means that the theory behind these algorithms no longer applies so they might perform poorly in practice. Because of this, we chose to base our approach on one of the simpler algorithms, ε-greedy, which was easier to alter so it performs better in the non-stationary case.

4.3 The Solution

The multi armed bandit problem has applications in many different domains (e.g. clinical trials, routing, online advertising) so it has been the subject of extensive research which produced a number of solutions. One family of simple and yet efficient solutions is called the $\varepsilon - greedy$ strategy [Wat89].

4.3.1 ε-greedy strategy

With the ε-greedy approach, a random flavor is chosen (exploration) with probability ε and the flavor with the best estimated reward (exploitation) is chosen with probability $1-\varepsilon$. This decision is made at every primitive call. For each flavor, the algorithm maintains the performance mean and uses it to choose the best flavor in the exploitation phase. The mean is updated after each call.

The efficiency of this method depends on the ε parameter (chosen by the user). If it is small (less exploration, more exploitation) there is less time wasted testing sub-optimal flavors, but it also means that it will take more time to find the optimal flavor.

A variant of this approach, the $\varepsilon - first$ strategy, is to first do the exploration completely and then the exploitation. If it is known that there will be T primitive calls, then the first εT calls will be to random flavors. After the exploration phase comes a pure exploitation phases which just picks the best flavor based on the computed means. This would work well in cases where the flavors keep their performance ranking for the entire query. So, the knowledge built in the beginning will be accurate until the end of the query. So, it could work well for platform adaptivity cases but it will not work so well for instance or call adaptivity. The performance ranking can change at any moment, even towards the end of the query, because of changes in data characteristics, system state, etc. Additionally, the performance of the first few primitive calls might be much worse than the rest because some objects are not in the cache yet (e.g. the data vectors).

The ε-greedy strategy is not optimal. Since ε is constant, the regret will increase linearly with the number of iterations. With the ε-decreasing strategy, ε is decreased after every iteration and [ACBF02] shows that this strategy can achieve optimal regret when ε decreases at a rate of $1/n$, n being the number of calls so far.

4.3.2 SoftMax strategy

SoftMax is a probability based approach which does not have clear exploration/exploitation phases. Instead, at every call, the algorithm chooses flavor k with probability p_k. The probabilities are calculated based on the performance means. Flavors with higher means will have higher probabilities. $p_k = e^{\frac{\mu_k}{t}} * \frac{1}{\sum_{i=1}^{n} e^{\frac{\mu_i}{t}}}$, where μ_k is the current mean of flavor k and t is an algorithm parameter called the temperature.

4.4 vw-greedy

To solve the MAB problem for our optimizer we designed the `vw-greedy` algorithm (Listing 4.4), based on the ε-greedy strategy, but with the following differences:

1. exploration and exploitation alternate in a deterministic pattern, instead of a random pattern
2. to choose the best flavor, we look at *recent* information about performance, instead of keeping an overall average of performance

The main goal of these changes is to improve the efficiency for the non-stationary case. The standard ε-greedy method computes the performance mean for each flavor using all calls since the beginning of the query. In the stationary case, this mean will eventually converge because the true mean is constant. But in our case it does not make sense to compute the overall mean. Instead, we compute the mean of recent calls only. This lets the algorithm handle sudden changes in performance, but makes it more vulnerable to noise.

Using a deterministic pattern of exploration/exploitation phases makes it easier to compute the mean of recent calls. Otherwise, the algorithm would need to keep an array with performance for recent calls to compute an accurate mean or use an approximating algorithm. It also helps when analyzing the results of the algorithm.

Suppose that in a query there are `QUERY_LENGTH` calls to a primitive. The `vw-greedy` algorithm performs exploration every `EXPLORE_PERIOD` calls. Exploration means choosing a random flavor, ignoring any performance information we have gathered so far. This random flavor is then used for the next `EXPLORE_LENGTH` primitive calls. The regret caused by exploration will grow linearly with the number of calls by an amount proportional to the ratio $\frac{EXPLORE\_LENGTH}{EXPLORE\_PERIOD}$. Additionally, every `EXPLOIT_PERIOD` primitive calls, the algorithm chooses the best flavor and uses this for the next `EXPLOIT_PERIOD` calls. This pattern is shown in Figure 4.2.

4.4.1 Relation to ε

`EXPLORE_PERIOD` and `EXPLORE_LENGTH` are related to the ε parameter used by ε-greedy and ε-first strategies. ε controls the number of explorations: in ε-greedy and ε-first there will be roughly $\varepsilon * QUERY\_LENGTH$ exploration calls. Our algorithm will do roughly $\frac{1}{EXPLORE\_PERIOD} * QUERY\_LENGTH$ explorations, so we can consider this algorithm to have $\varepsilon = \frac{1}{EXPLORE\_PERIOD}$.

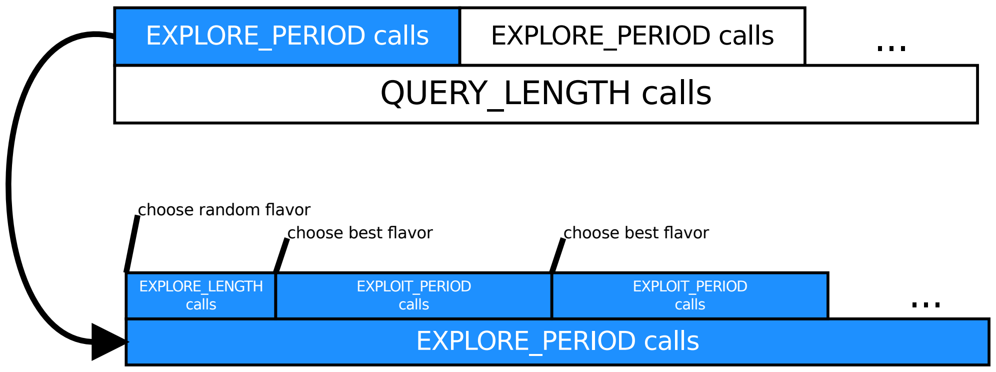

Figure 4.2: `vw-greedy` algorithm pattern

4.4.2 Learning process

For determining the best flavor, we compute the average number of tuples processed per CPU clock cycle. The best flavor is then the flavor for which this value is the maximum. This average is computed only for the calls in the same phase (either exploration or exploitation). So, it is an estimate of the performance of *recent* calls only. During a given phase, the chosen flavor is called a number of times (either `EXPLOIT_LENGTH` or `EXPLOIT_PERIOD`) and we sum the number of tuples and the number of CPU cycles for all calls. At the end of the phase, we assign this performance to the flavor. After this, a decision is made using the new information about this flavor together with old information for the other flavors.

4.4.3 Algorithm

```
function vw-greedy(primitive, last_call_perf) {
        // this determines the exploration/exploitation pattern
        adp_idx = primitive.adp_idx
        // get information about the currently active flavor
        cflavor = primitive.current_flavor
        // get the performance of the currently active flavor
        cperformance = primitive.performance
        new_flavor = cflavor

        // at the end of each phase we update statistics
        if(adp_idx % EXPLOIT_PERIOD == EXPLORE_LENGTH){
                set_perf(cflavor, cperformance)
                reset_perf(primitive)
        }
        if(adp_idx == 0) {
                // start a new exploration phase
                new_flavor = get_random_flavor()
        } else if(adp_idx % EXPLOIT_PERIOD == EXPLORE_LENGTH) {
                // end of an exploration or exploitation phase
                new_flavor = get_best_flavor()
        } else {
                // within a phase
                update_perf(primitive.performance, last_call_perf)
        }
        // switch to another flavor
        primitive.current_flavor = new_flavor
        // update algorithm state
        primitive.adp_idx = (adp_idx + 1) % EXPLORE_PERIOD
}
```

Listing 4.4: `vw-greedy` algorithm

The `vw-greedy` function, shown in Listing 4.4, is called after every primitive call to update the knowledge about the current flavor and to select a new flavor if it is the case. The exploration/exploitation pattern is given by the `adp_idx` variable. This is incremented after every primitive call. Alternating phases is done in the following way.

1. if `adp_idx == k * EXPLORE_PERIOD`, for some k: start a new exploration phase
2. if `adp_idx == k * EXPLOIT_PERIOD + EXPLORE_LENGTH`, for some k: end the current exploration or exploitation phase, update the knowledge for the current flavor and start a new exploitation phase
3. in any other case it means we are currently inside a phase: sum the statistics of the last call

4.4.4 Simulations

This section discusses two simulations that show how the algorithm works. For these examples we used `EXPLORE_PERIOD` = 1024, `EXPLOIT_PERIOD` = 256 and `EXPLORE_LENGTH` = 32. The performances of the three flavors used here were generated randomly but they do not obey any probability distribution.

In the first experiment we simulated a primitive for which the performance ranking of its 3 flavors remains the same throughout the entire query execution.

In Figure 4.3, the orange curve is the performance trace of the `vw-greedy` algorithm. It almost completely covers the red curve, which shows the performance of `flavor 1`, the fastest flavor for this experiment. So, for this simple test, the algorithm does use the best flavor most of the time. The small black vertical lines at the bottom of the chart mark the start of an exploration phase. The distance between them is therefore equal to EXPLORE_PERIOD. Because of the exploration phases, there are a lot of spikes in the performance. We can also see that at the start of the query, the algorithm is unlucky and first chooses the worst flavor, then the second worst and only after 2 exploration phases it finds the best flavor.

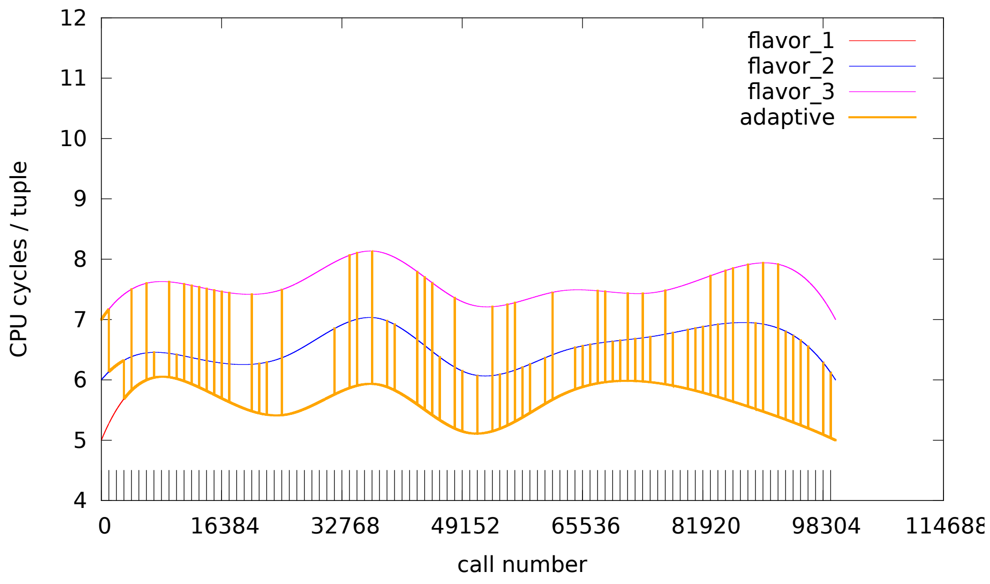

Figure 4.3: `vw-greedy` algorithm simulation with constant performance ranking

For the second experiment, the algorithm is simulated with 3 flavors for which one of them is always the best, except for a portion of the query, where another flavor becomes the best. In Figure 4.4 we see that usually `flavor_1` is the fastest, except near the middle of the query, where `flavor_3` becomes the best, for a short while. The algorithm reacts to this switch as well as the switch back.

4.4.5 Evaluation

The experiments in this section were performed on profile data gathered from a TPC-H SF-100 benchmark on Machine 1. The data contains over 300 primitive instances. The number of calls to these primitives ranged from $16K$ to $32K$. For these preliminary evaluation we used 3 flavors (gcc, icc and clang).

To decide how good an algorithm is, we compare its performance with the optimal performance. Optimal performance is achieved by choosing the best flavor for every primitive call. The optimal performance for a query is the sum of the optimal performances of its primitive instances. The score for a query is the ratio between the performance obtained by the algorithm and the optimal performance of that query, i.e. the lower the score, the better.

Parameters

First, we calibrated the `vw-greedy` parameters by trying different combinations of parameter values. The values {64, 128, 256, 512, 1024, 2048} were used for EXPLORE_PERIOD while the other parameters were powers of 2, so that EXPLORE_PERIOD > EXPLOIT_PERIOD > EXPLORE_LENGTH. Table 4.1 summarizes the results. For each query, the table shows the best

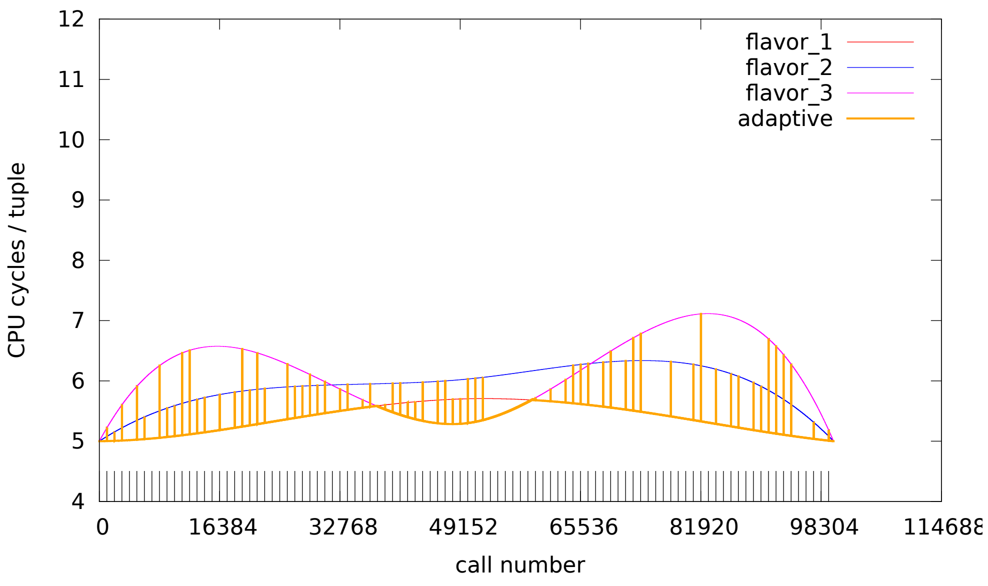

Figure 4.4: `vw-greedy` algorithm simulation with one performance ranking switch

and worst parameter combinations. There is no clear winning value for `EXPLORE_PERIOD`, while for the other two parameters smaller values seem to lead to better scores. The worst combination for any query always has a high value for `EXPLOIT_PERIOD`. Among the best scores, the worst (1.034) is obtained for Query 20. This score means that, for this query, the algorithm is within 4% of the optimal performance.

For the entire benchmark, the best overall score, 1.014 was obtained by the combination $(1024, 8, 2)$ while the worst overall score, 1.07 was for $(128, 64, 1)$.

Algorithm comparison

To see how our algorithm compares with the others, we evaluated 3 instances of each algorithm on this data set. The results are summarized in Table 4.2. The parameters for `vw-greedy` are those that obtained the best 3 scores in the calibration experiment. For the others, we chose parameters similar to the ones evaluated in [VM05]. For the ε-greedy and ε-first algorithms we also chose one parameter that we thought is equivalent to the `vw-greedy` parameters (as described in Section 4.4.1, e.g. for `EXPLORE_PERIOD` of 1024 we chose $\varepsilon = 0.001$).

The best overall performance (Table 4.2) is obtained by an ε-first algorithm, although the scores for all algorithms, with the exception of SoftMax, are very similar (as it also concluded in [VM05]). SoftMax is not so efficient on this benchmark probably because there is often not a big difference between flavor performances, thus the flavor selection probabilities will also be similar (see Section 4.3.2). A bigger temperature parameter would make it less likely that a sub-optimal flavor is chosen, even if the performance difference is small. This is because $p_k = (e^t)^\mu$, so increasing t amplifies the difference between flavors. However, increasing the temperature will lead to less exploration, which will hurt in other cases, e.g. when there are many flavors.

Although the ε-first strategy has the best overall score on this benchmark, its average score per primitive is slightly worse than that of `vw-greedy`. In fact, `vw-greedy` has the best average score, which means its efficiency is more stable and if we choose a random primitive, we expect `vw-greedy` to perform better than the other algorithms. But, seeing ε-first perform

Table 4.1: vw-greedy best parameter combinations for each TPC-H query

Query	Best			Worst		
	Parameters	ε	Score	Parameters	ε	Score
Q01	(512,16,4)	0.002	1.001	(128,64,1)	0.008	1.018
Q02	(2048,8,4)	0.0004	1.010	(64,32,1)	0.0004	1.088
Q03	(64,8,2)	0.016	1.010	(1024,512,1)	0.001	1.058
Q04	(2048,32,2)	0.0004	1.001	(64,32,1)	0.016	1.080
Q05	(256,16,2)	0.004	1.004	(128,64,1)	0.008	1.043
Q06	(512,8,1)	0.002	1.011	(512,256,1)	0.002	1.075
Q07	(512,4,1)	0.002	1.008	(2048,1024,1)	0.0004	1.051
Q08	(128,8,2)	0.008	1.010	(128,64,1)	0.008	1.072
Q09	(2048,32,2)	0.0004	1.001	(64,32,1)	0.016	1.042
Q10	(512,4,1)	0.002	1.006	(64,32,1)	0.016	1.036
Q11	(128,8,2)	0.008	1.010	(1024,512,1)	0.001	1.066
Q12	(2048,4,1)	0.0004	1.005	(2048,128,16)	0.004	1.044
Q13	(256,16,2)	0.004	1.001	(128,64,1)	0.008	1.026
Q14	(256,8,2)	0.004	1.007	(2048,1024,2)	0.0004	1.079
Q15	(256,4,1)	0.004	1.006	(2048,1024,1)	0.0004	1.101
Q16	(2048,16,2)	0.0004	1.001	(2048,1024,1)	0.0004	1.081
Q17	(512,16,2)	0.002	1.020	(2048,1024,1)	0.0004	1.214
Q18	(1024,8,2)	0.001	1.000	(1024,512,1)	0.001	1.070
Q19	(1024,8,2)	0.001	1.015	(128,64,1)	0.008	1.066
Q20	(128,8,2)	0.008	1.013	(2048,1024,1)	0.0004	1.111
Q21	(512,4,1)	0.002	1.009	(64,32,1)	0.016	1.134
Q22	(256,8,2)	0.004	1.001	(2048,512,1)	0.0004	1.024

so well also says something about the flavors we tested. If a strategy that only explores in the beginning obtains such a good score, it means that often the performance ranking remains the same (perhaps one flavor degrades, which will be noticed by ε-first, but rarely will a flavor improve and become better than the others). Based on this, we added a initial exploration phase to our algorithm. In the first `EXPLORE_PERIOD` calls we randomly choose flavors. After this phase the algorithm continues with the standard strategy of alternating exploration with exploitation.

4.4.6 Summary

This section described the `vw-greedy` algorithm which was implemented within Vectorwise to solve the optimization problem needed for the micro adaptive system. It follows the ε-greedy approach, slowly building up knowledge about the performance of the flavors using exploration phases which need a fraction of the primitive calls. We evaluated the efficiency of this algorithm on data from a TPC-H benchmark and compared it with other algorithms. Results showed that `vw-greedy` might not obtain the best overall efficiency but it is more stable.

Table 4.2: Overall performance of the 5 MAB algorithms with different parameters

Algorithm	Score	Avg.
eps-first(0.001)	1.012	1.016
vw-greedy(1024,8,2)	1.015	1.011
eps-greedy(0.001)	1.015	1.015
vw-greedy(2048,8,1)	1.015	1.015
eps-decreasing(1.0)	1.015	1.016
eps-decreasing(0.1)	1.015	1.016
eps-greedy(0.05)	1.017	1.015
eps-first(0.1)	1.017	1.023
vw-greedy(2048,8,2)	1.018	1.013
eps-greedy(0.1)	1.018	1.021
eps-first(0.05)	1.020	1.019
eps-decreasing(5.0)	1.022	1.015
SoftMax(0.001)	1.073	1.115
SoftMax(0.05)	1.073	1.118
SoftMax(0.1)	1.081	1.122

Table 4.3: Best MAB algorithm for every TPC-H query

Query	Best	
	Algorithm	Score
Q01	vw-greedy(1024,8,2)	1.001
Q02	vw-greedy(1024,8,2)	1.011
Q03	eps-decreasing(0.1)	1.015
Q04	vw-greedy(1024,8,2)	1.001
Q05	eps-decreasing(0.1)	1.002
Q06	vw-greedy(1024,8,2)	1.012
Q07	vw-greedy(1024,8,2)	1.009
Q08	eps-decreasing(0.1)	1.007
Q09	eps-decreasing(1.0)	1.000
Q10	eps-decreasing(1.0)	1.004
Q11	eps-decreasing(0.1)	1.010
Q12	vw-greedy(2048,8,2)	1.005
Q13	vw-greedy(2048,8,2)	1.002
Q14	vw-greedy(1024,8,2)	1.011
Q15	vw-greedy(1024,8,2)	1.012
Q16	eps-decreasing(5.0)	1.001
Q17	vw-greedy(2048,8,2)	1.030
Q18	eps-decreasing(0.1)	1.000
Q19	eps-first(0.001)	1.009
Q20	eps-greedy(0.05)	1.019
Q21	eps-greedy(0.001)	1.006
Q22	vw-greedy(1024,8,2)	1.001

Chapter 5

TPC-H Experiments

In this chapter we discuss experiments with the TPC-H benchmark. In the first section we go back to the case studies of Chapter 3 and show examples of queries where they occur. We also compare the performance of the micro adaptive system with that of the standard system. The next two sections describe how we measured the variation of flavor performance on the same platform and across platforms. In the end, we also report the improvement in overall query times obtained by micro adaptivity on the TPC-H SF-100.

5.1 Flavors

This section revisits the cases presented in Chapter 3 and shows examples of where they occur in the TPC-H queries. Once more, we try to explain what happens in each case and also show that our algorithm is able to exploit these micro adaptivity opportunities and that the heuristics currently used by Vectorwise are not perfect. In every graph, the thick orange line shows the performance obtained with micro adaptivity while the dotted black line shows the performance of the standard Vectorwise build.

5.1.1 Compilers

Static case

As an introductory example, let us look at how the adaptive system works for a string comparison primitive, `select_==_str_col_str_val` called by a Select operator during execution of Query 3 on Machine 3.

The graph in Figure 5.1 shows the performance of the 4 compiler-flavors of this primitive, along with the performance of the micro adaptive system. We can notice that the 4 flavors have different costs per tuple, but these costs are constant for the entire execution. The gcc flavor is the fastest while the clang flavor is the slowest. The micro adaptive systems picks the fastest flavor for almost all the primitive calls.

The `vw-greedy` initial exploration phase, which lasts `EXPLORE_PERIOD` calls, is noticeable in Figure 5.1. The unstable performance in the beginning of the query is due to this phase, when `vw-greedy` performs a random switch every `EXPLORE_LENGTH` calls. The low performance peaks are caused by the normal exploration phases, that occur every `EXPLORE_PERIOD` calls for the entire duration of the query.

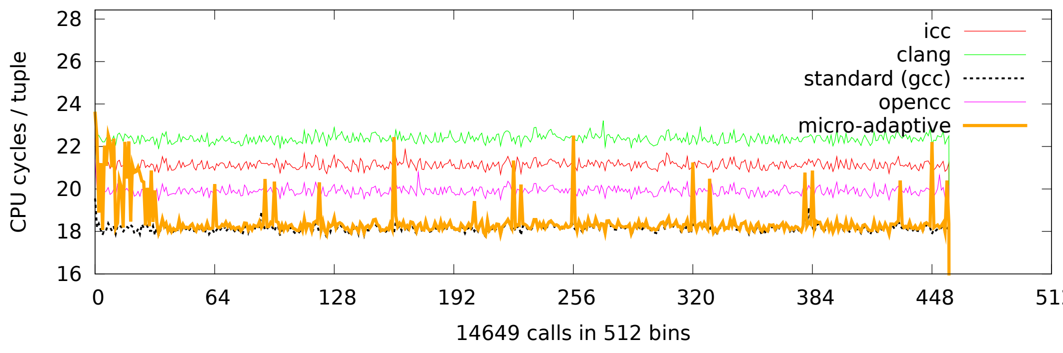

Figure 5.1: Micro adaptive execution with 4 `select_==_str_col_str_val` compiler-flavors in TPC-H Query 3 on Machine 3

Dynamic case

Figure 5.2 and Figure 5.3 illustrate the instance adaptivity case first presented in Section 3.5. For Query 2, gcc is the fastest flavor and the micro adaptive system is able to detect and exploit this. For the primitive instance called when executing Query 21 gcc is no longer the best choice. For the majority of primitive calls, clang is faster than the rest. However, towards the end of the query, gcc becomes the fastest again.

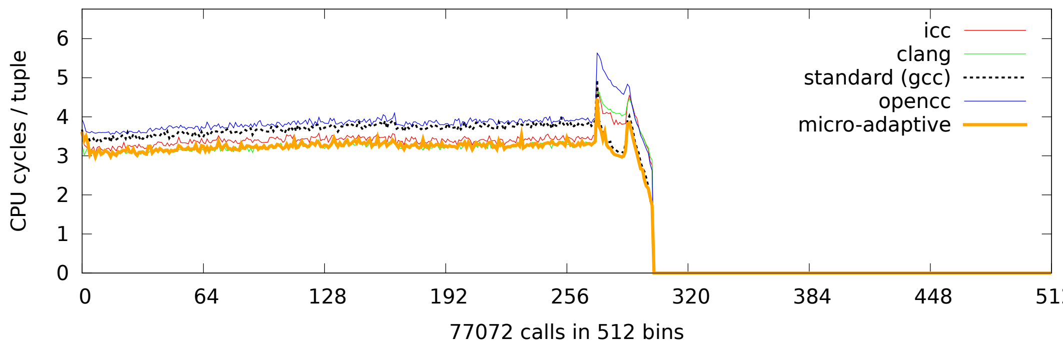

Figure 5.2: Micro adaptive execution with 4 `select_==_sint_col_sint_val compiler -flavors` in TPC-H Query 21. An example of call adaptivity.

5.1.2 Data dependency

In this experiment we investigate the data dependency and control dependency flavors of the same selection primitive as in section Section 3.4, `select_int_ge_int_col_int_val`, but this time called in TPC-H Query 6. Again, we can see that the current heuristic does a good job except for a very small fraction of the calls for which it incorrectly chooses the control dependency flavor (Figure 5.4). The micro adaptive algorithm is able to exploit even this very small opportunity. Of course, the benefit of this on the overall query time is marginal, but this example proves that the heuristic is not perfect. It is not unlikely that some query will exhibit precisely this selectivity that beats the heuristic.

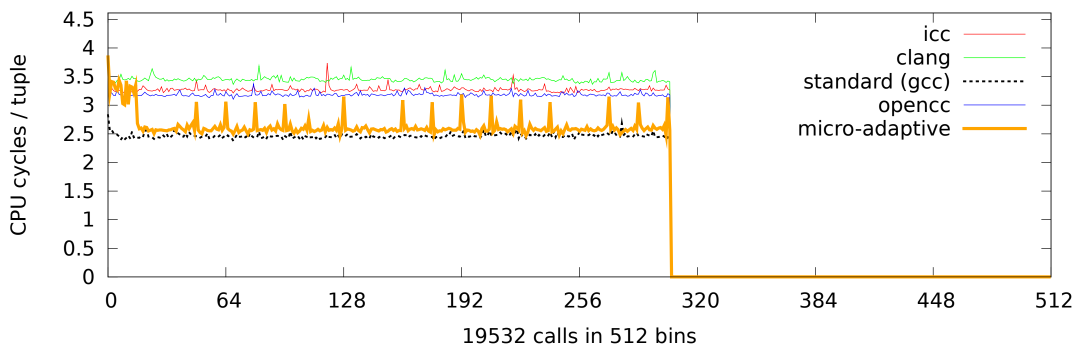

Figure 5.3: Micro adaptive execution with 4 `select_==_sint_col_sint_val compiler -flavors` in TPC-H Query 2

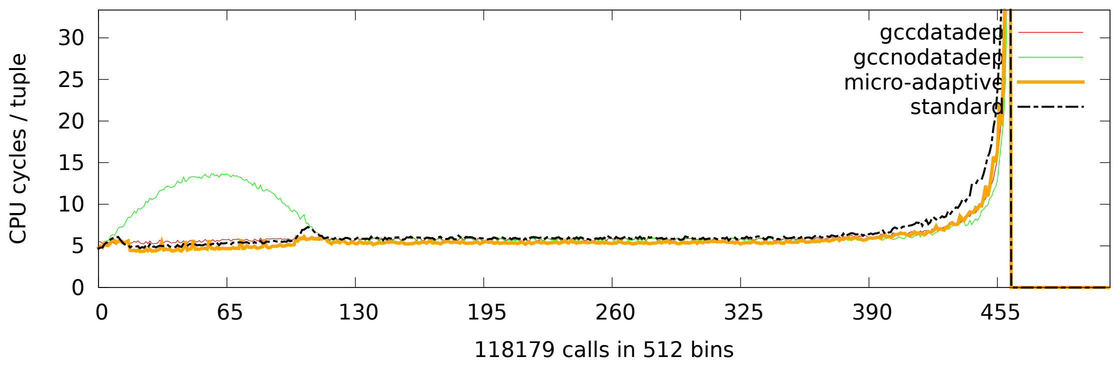

Figure 5.4: Micro adaptive execution with control dependency and data dependency flavors of a selection primitive called in TPC-H Query 6

5.1.3 Full computation

We saw in Section 3.8 that full computation is very effective when the primitive can be SIMDized and the data type is small. Figure 5.5 shows the performance of a short integer multiplication primitive called in Query 1. The full computation flavor of this primitive (shown in blue) is almost always faster in this test and the micro adaptive system is able to detect and exploit this. However, towards the end of the query there is a sudden performance degradation. The full computation flavor becomes slower. Fortunately, at this point, the micro adaptive system switches to the other flavor. The reason for the performance degradation is a change in input data characteristics. The dotted blue line represents the number of tuples processed by every primitive call. For most of the query, this primitive processes 1024 tuples (which is the default vector size). The performance drops near the end because the number of tuples also drops and, as noted in Section 3.8, the effectiveness of full computation decreases with the input vector size.

In Figure 5.6 we see the reverse of the scenario from Figure 5.5. Here, full computation is almost never an improvement, except for a section in the middle of the query (around bin number 192). Again, we see the correlation with input vector size. Full computation becomes better as the vector size increases. In the middle of the query, full computation is slightly better than the standard primitive.

Finally, Figure 5.7 shows a case when full computation is never effective. This is due to the constant small input vector size (around 40).

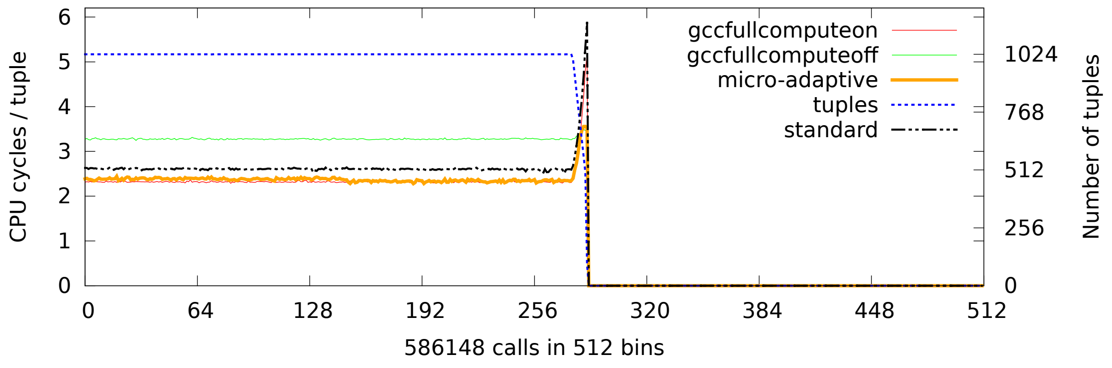

Figure 5.5: Full computation benefit for short integer multiplication primitive called in TPC-H Query 1

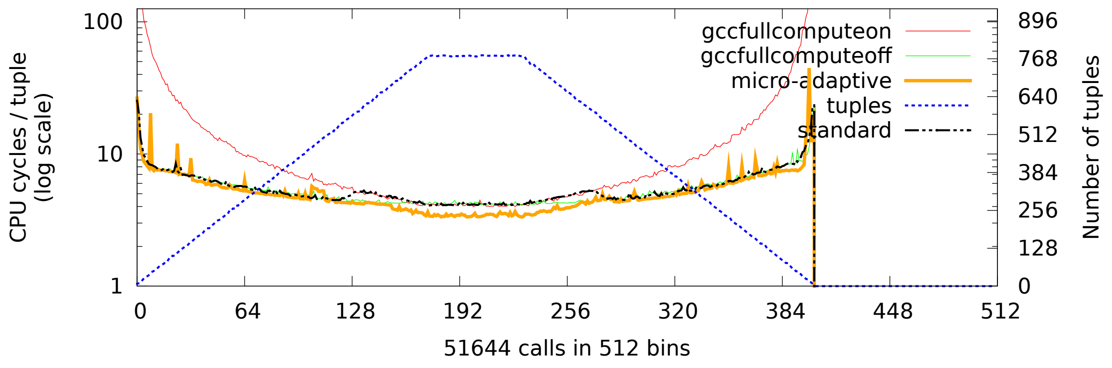

Figure 5.6: Full computation call adaptivity. Full computation is bad for most of the query except for a section in the middle. TPC-H Query 15, long integer column multiplication primitive

5.1.4 Loop fission

The micro benchmarks in Section 3.6 proved that loop fission is not always beneficial, at least not when applied to the Vectorwise bloom filter check primitive. Furthermore, the benefit depends on the size of the bloom filter. In this section we extend that analysis with two examples from the TPC-H benchmark.

For one bloom filter check primitive instance, in Query 7, loop fission is always better. Figure 5.8 and Figure 5.9 shows the performance of this primitive instance on Machine 1 and 3. Both the standard Vectorwise and the micro adaptive Vectorwise favor the fission flavor. In this static scenario, standard Vectorwise will have slightly better performance than micro adaptive Vectorwise. The former makes a decision to use fission after the bloom filter is built and the whole probe phase is executed with fission while the latter periodically switches to the non fission flavor, during the probe phase, to check if it somehow became faster. So far, we saw that fission efficiency depends on the bloom filter size (see Section 3.6), which does not change during the probe phase, so perhaps micro adaptivity is not useful for these two flavors. However, we can see in Figure 5.8 and Figure 5.9 that the fission flavor does have some dependence on data -

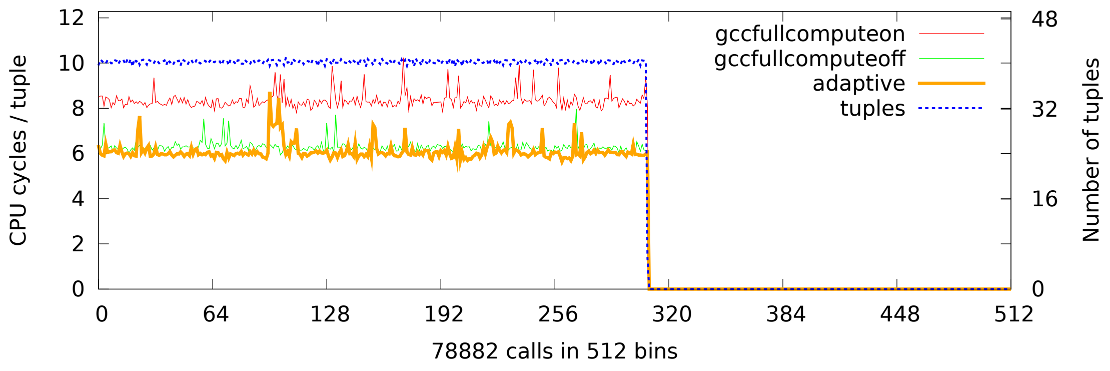

Figure 5.7: Full computation is ineffective with small vectors. TPC-H Query 11, long integer column multiplication primitive

the performance of this flavor in the beginning and in the end of the query, on both machines, is slightly better. Therefore, we should not rule out call adaptivity for these flavors. Furthermore, if Vectorwise is ever extended with other adaptive query processing methods, like the MJoin operator (see Section 1.2.2), then the size of the bloom filter might not be constant during the probe phase.

In Figure 5.9 we see that on Machine 3 the non fission flavor is almost 3 times slower than the fission flavor, so if the current heuristic is ever inaccurate, the penalty will be great, especially considering that bloom filter checks are often one of the most time consuming operations in a query. Figure 5.10 and Figure 5.11 shows such an example. This happens in the same TPC-H Query but in another hash join operator instance. Here, the bloom filter contains around 80K keys (which use approx. 700KB of memory) and the default heuristic threshold to activate fission is 100K keys. In the previous example, the bloom filter contained over 1M keys.

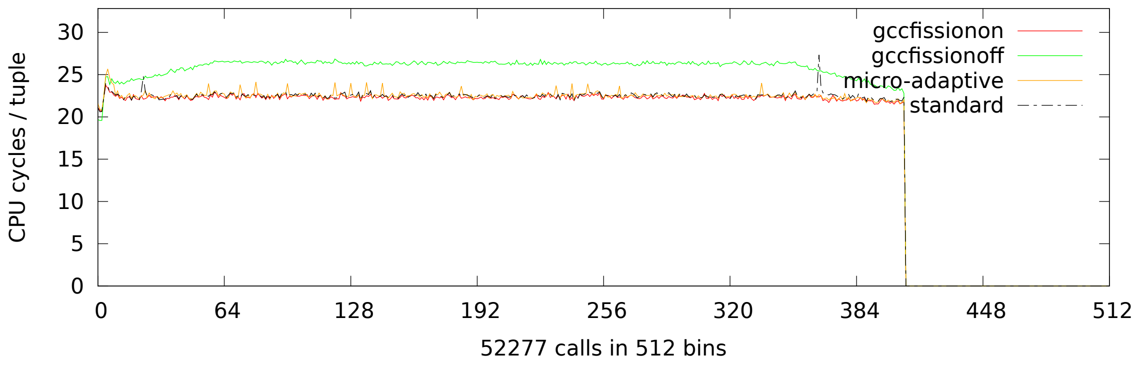

Figure 5.8: Micro adaptive execution with fission and non fission flavors of a bloom filter check primitive called in TPC-H Query 7 on Machine 1

5.1.5 Loop unrolling

These examples are from TPC-H Query 1. Figure 5.12 and Figure 5.13 show the same primitive instance on different machines (1 and 3). This primitive performs long integer addition between a column and a constant. Many Vectorwise primitives are unrolled with a factor of 8 and we see that this can be good on a machine (Figure 5.13) but bad on another machine (Figure 5.12). However, it does not mean that unrolling is always bad (or always good) on a certain machine.

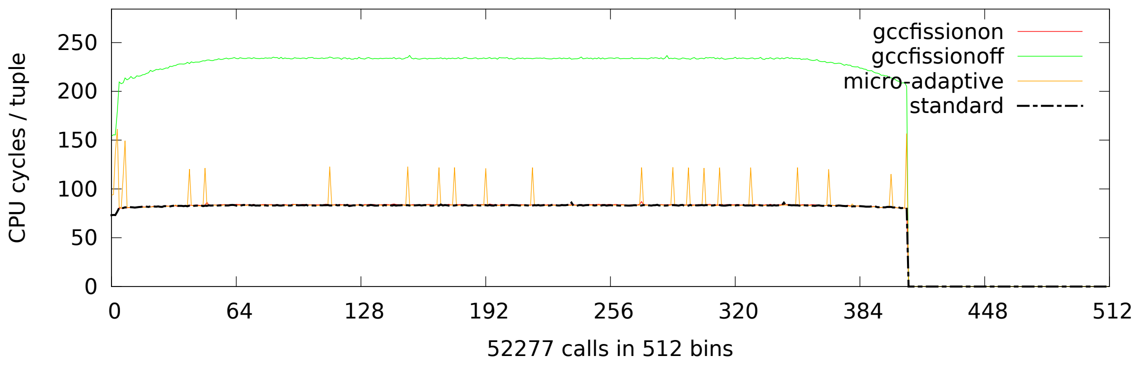

Figure 5.9: Micro adaptive execution with fission and non fission flavors of a bloom filter check primitive called in TPC-H Query 7 on Machine 1

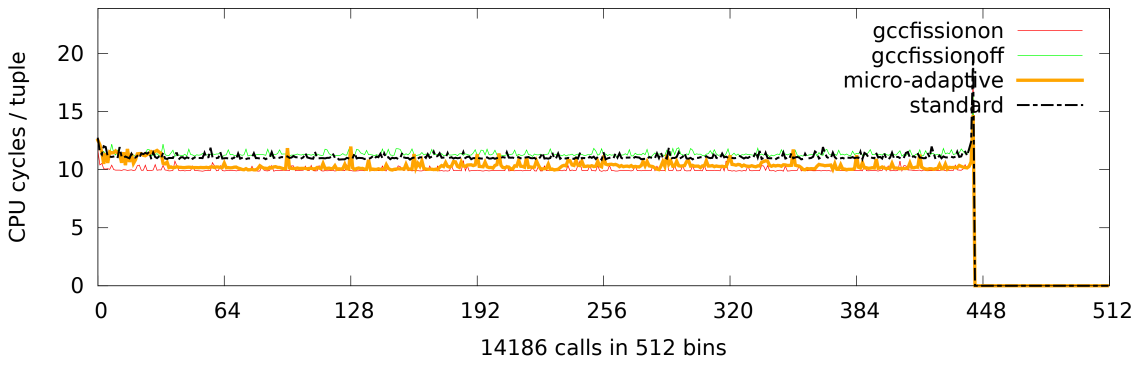

Figure 5.10: Micro adaptive execution with fission and non fission flavors of a bloom filter check primitive called in TPC-H Query 7 on Machine 1. A case when the current heuristic is inaccurate

Figure 5.14 shows that unrolling by 8 is beneficial on Machine 1 for a long integer column multiplication primitive.

5.2 Flavor variation on the same platform

In this experiment we ran a TPC-H SF-100 benchmark for each of the 32 flavors on Machine 1, single threaded. For every primitive instance we calculated the relative standard deviation of the performances of its flavors. The purpose of this experiment is to get a measure of how different, in terms of performance, the various flavors are on a given machine. If they are not so different, i.e. the performance deviation is small, then it would mean that there is less need for micro adaptivity.

Table 5.1 shows the deviation for the most time consuming primitives of each query. For each primitive we also identified the slowest and fastest flavors and their times in CPU clock cycles. We can see that for some queries the deviation is small (3% to 7%) but for more than half of the queries the deviation is bigger than 10%. For example, for Query 6, the deviation is 68% and the fastest flavor is more than two times as fast as the slowest flavor. This clearly shows that there is a great potential for micro adaptivity on this machine.

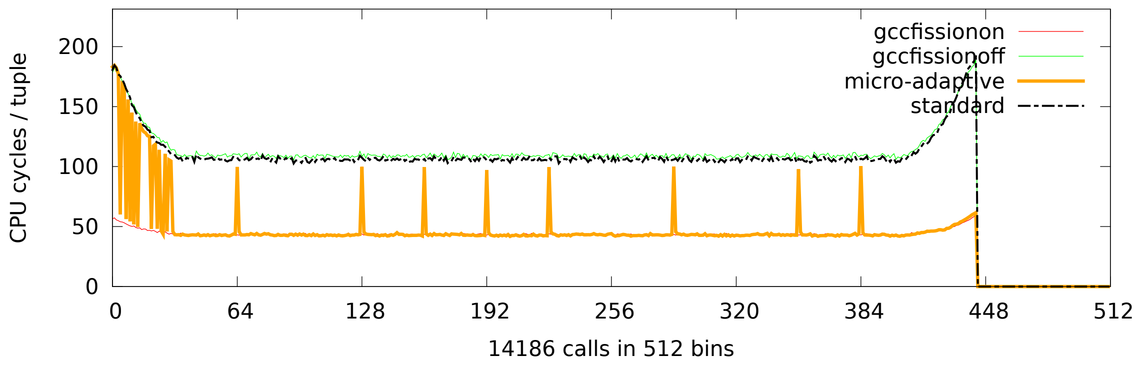

Figure 5.11: Micro adaptive execution with fission and non fission flavors of a bloom filter check primitive called in TPC-H Query 7 on Machine 3. A case when the current heuristic is inaccurate

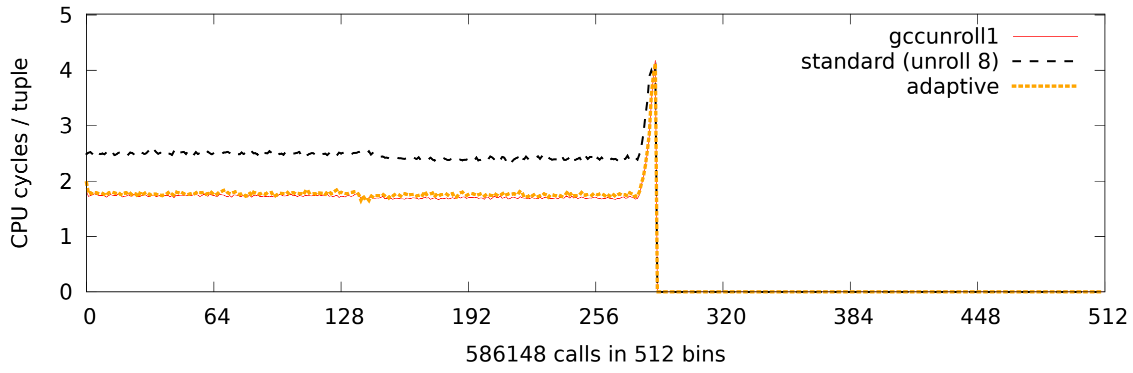

Figure 5.12: Micro adaptive execution with/without loop unrolling for a long integer addition primitive called in TPC-H Query 1 on Machine 1. Loop unrolling by a factor of 8 leads to poor performance.

5.3 Flavor variation on different platforms

Suppose that for a certain workload on a certain machine we know the best flavor for every primitive instance. We could then configure Vectorwise to use only the best flavors. So, how will this static configuration perform on different machines? The data in Table 5.2 was obtained by running Vectorwise on all 4 machines using the best flavors for Machine 1. The column named "Static" represents the slowdown due to the flavor configuration. The slowdown is the ratio between the performance of the static configuration execution and the optimal configuration execution. The static configuration is the one obtained on Machine 1 while the optimal configuration is specific to each machine. Looking at the values of this slowdown on Machines 2, 3 and 4 we can conclude that there is indeed a significant variation in the flavor performances on these machines. The static configuration approach does not use the entire potential of this flavor collection. However, it does seem to perform better than the standard Vectorwise. The column marked "Std." shows the slowdown of the standard Vectorwise while the column marked "Adp." contains the slowdowns for the micro adaptive system. In many cases micro adaptive Vectorwise has the lowest slowdown, but there are cases where it performs poorly.

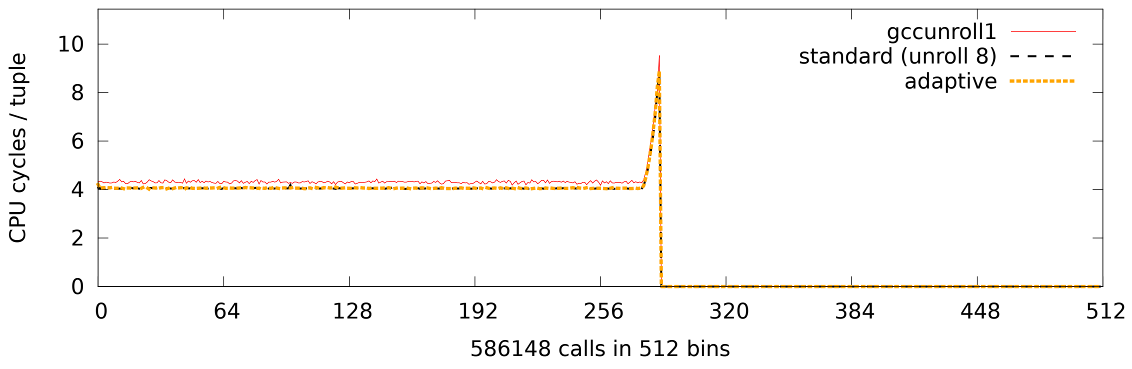

Figure 5.13: Micro adaptive execution with/without loop unrolling for a long integer addition primitive called in TPC-H Query 1 on Machine 3. Loop unrolling by a factor of 8 is slightly better than no loop unrolling.

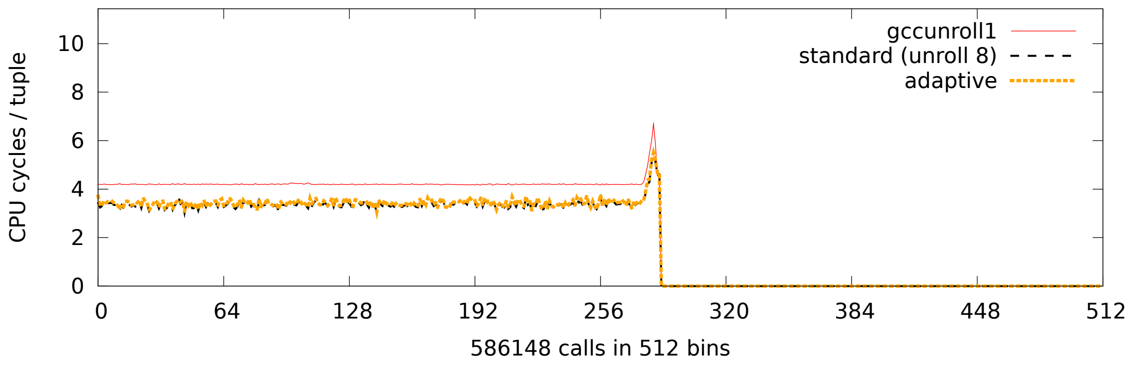

Figure 5.14: Micro adaptive execution with/without loop unrolling for a long integer column multiplication primitive called in TPC-H Query 1 on Machine 1. Loop unrolling by 8 is beneficial.

5.4 Evaluation

We conclude this chapter by showing the performance improvement obtained with micro adaptivity on two TPC-H SF-100 benchmarks, on Machine 1 and on Machine 3. In these experiments, we compared the standard Vectorwise, built with gcc, with the micro adaptive Vectorwise which features 32 flavors. We compared the overall times of each query and determined the speedup as the ratio between the standard time and the micro adaptive time. The results (Table 5.3) show that micro adaptivity is often an improvement, although not a great one. However, we did not expect to see substantial improvements in TPC-H overall query times, with these flavors.

Looking at the geometric mean of the query times, in Table 5.3, we can say that micro adaptivity improves the time of a TPC-H query by 5%. Consequently, micro adaptivity increases the power of the database engine - number of queries that can be executed per hour - with 30 on Machine 1 and 14 on Machine 3.

5.5 Summary

In this chapter we extended the analysis from Chapter 3 with concrete examples from the TPC-H benchmark. We also attempted to quantify the flavor performance variation and concluded that the variation is significant across platforms as well as on the same platform. In the end we showed the improvement obtained by micro adaptivity on the overall query times in the TPC-H benchmark.

Table 5.1: Flavor performance variation on the same machine. For each query the most time consuming primitive is shown. The numbers are CPU clock cycles. Deviation is the relative standard deviation

Query	Primitive	Slowest	Fastest	MicroAdaptive	Deviation
Q1	aggr_sum	gccunroll1 1355922312	openccunroll1 1169271584	1209802928	3%
Q2	map_mergejoin	openccfissionoff 72946036	iccfullcomputeon 37135944	54626468	26%
Q3	select_==_str	gccdatadep 85618092	clangunroll1 65465636	69749952	6%
Q4	map_fetch	iccfissionoff 273071728	clangnodatadep 190541204	194576312	13%
Q5	sel_bitfiltercheck	openccfullcomputeoff 330998564	iccdatadep 236935884	249322428	10%
Q6	select_<=	openccnodatadep 445397728	openccfissionoff 93634572	101971156	68%
Q7	map_mergejoin	openccnodatadep 529049272	iccfissionon 311473596	325718104	22%
Q8	map_mergejoin	openccfissionoff 644260776	iccfissionoff 365033352	385578560	20%
Q9	sel_bit2filtercheck	gccfissionoff 9176788956	clangfissionon 4998605484	6278899336	17%
Q10	gen_put	gccfullcomputeoff 1127053206	iccfissionoff 848556718	1017373346	9%
Q11	sel_bitfiltercheck	openccfissionon 211538568	icc 121954856	152534536	13%
Q12	select_!=_str	gccnodatadep 891499244	clangunroll1 728454376	746825608	5%
Q13	map_match	clang 11719979740	gccfullcomputeon 8409661034	8689464164	9%
Q14	gen_put	iccunroll1 626982434	iccdatadep 562460158	572536980	3%
Q15	aggr_sum	openccunroll1 444898244	clangfullcomputeon 253627764	261066304	17%
Q16	map_recheck	iccnodatadep 810380064	gccunroll1 612402064	658416136	10%
Q17	sel_bitfiltercheck	openccfissionon 1490432344	iccnodatadep 801953636	893472456	16%
Q18	sel_diff	clangfissionoff 1573604792	gccfullcomputeoff 1284350388	1314197656	6%
Q19	select_!=_str	clangfullcomputeoff 2118934056	clangunroll1 1678897088	1731130116	6%
Q20	sel_bit2filtercheck	openccunroll1 502904260	openccfissionoff 404689144	461457752	7%
Q21	sel_bit2filtercheck	clangfissionoff 1900119904	icc 1347695124	1457196956	10%
Q22	sel_bit2filtercheck	opencc 797822296	iccfullcomputeon 642834300	667607116	6%

Table 5.2: Flavor performance variation across platforms. The numbers represent slowdowns compared to the optimal flavor configuration

Query	Machine 1			Machine 2			Machine 3			Machine 4		
	Static	Std.	Adp.	Static	Std.	Adp.	Static	Std.	Adp.	Static	Std.	Adp.
Q01	1.00	1.16	1.03	1.12	1.26	1.22	1.13	1.22	1.06	1.05	1.27	1.05
Q02	1.00	1.27	1.13	1.16	1.16	1.11	1.19	1.21	1.11	1.28	1.20	1.13
Q03	1.00	1.24	1.10	1.10	1.17	1.07	1.17	1.16	1.08	1.15	1.20	1.09
Q04	1.00	1.21	1.02	1.08	1.34	1.04	1.03	1.17	1.04	1.07	1.22	1.04
Q05	1.00	1.14	1.11	1.08	1.31	1.02	1.15	1.16	1.05	1.05	1.11	1.06
Q06	1.00	1.49	1.03	1.03	1.38	1.06	1.04	1.13	1.07	1.07	1.12	1.18
Q07	1.00	1.26	1.06	1.12	1.21	1.13	1.20	1.31	1.10	1.09	1.23	1.09
Q08	1.00	1.15	1.10	1.06	1.09	1.08	1.21	1.20	1.08	1.09	1.13	1.09
Q09	1.00	1.05	1.04	1.08	1.04	1.15	1.15	1.15	1.03	1.17	1.04	1.04
Q10	1.00	1.16	1.08	1.08	1.13	1.06	1.12	1.13	1.06	1.41	1.35	1.07
Q11	1.00	1.08	1.05	1.10	1.06	1.05	1.21	1.22	1.09	1.06	1.09	1.05
Q12	1.00	1.16	1.02	1.13	1.27	1.04	1.06	1.07	1.02	1.06	1.10	1.02
Q13	1.00	1.05	1.01	1.12	1.07	1.04	1.12	1.20	1.01	1.16	1.14	1.08
Q14	1.00	1.05	1.03	1.04	1.04	1.03	1.14	1.19	1.01	1.09	1.05	1.09
Q15	1.00	1.14	1.03	1.07	1.09	1.04	1.09	1.18	1.05	1.03	1.10	1.03
Q16	1.00	1.08	1.05	1.10	1.05	1.03	1.04	1.14	1.02	1.06	1.06	1.08
Q17	1.00	1.08	1.04	1.03	1.13	1.04	1.27	1.25	1.04	1.03	1.06	1.08
Q18	1.00	1.21	1.02	1.02	1.24	1.07	1.07	1.27	1.02	1.09	1.27	1.08
Q19	1.00	1.13	1.03	1.04	1.16	1.10	1.07	1.09	1.05	1.06	1.09	1.09
Q20	1.00	1.12	1.04	1.07	1.10	1.07	1.15	1.15	1.08	1.05	1.10	1.07
Q21	1.00	1.14	1.04	1.09	1.13	1.06	1.14	1.16	1.03	1.16	1.11	1.05
Q22	1.00	1.09	1.05	1.10	1.09	1.08	1.17	1.14	1.04	1.11	1.11	1.08

Table 5.3: Micro adaptive Vectorwise vs. standard Vectorwise on TPC-H SF-100 benchmark, Machine 1 and Machine 3, single threaded.

Query	Machine 1			Machine 3		
	Micro adaptive time (s)	Standard time (s)	Speedup	Micro adaptive time (s)	Standard time (s)	Speedup
Q1	27.04	29.68	1.10	37.8	41.99	1.11
Q2	1.48	1.5	1.01	2.44	2.59	1.06
Q3	1.28	1.35	1.05	2.37	2.53	1.07
Q4	1.22	1.38	1.13	1.77	1.95	1.10
Q5	4.74	4.9	1.03	11.6	12.84	1.11
Q6	1.52	1.73	1.14	2.37	2.42	1.02
Q7	6.59	7.09	1.08	13.32	15.06	1.13
Q8	6.6	6.73	1.02	13.18	14.43	1.09
Q9	44.61	46.66	1.05	133.18	138.46	1.04
Q10	6.66	7.3	1.10	24.09	24.04	1.00
Q11	2.01	2.05	1.02	4.24	4.62	1.09
Q12	5.45	5.99	1.10	6.91	7.02	1.02
Q13	40.56	41.24	1.02	204.74	193.45	0.94
Q14	2.97	3.07	1.03	10.05	10.2	1.01
Q15	1.33	1.42	1.07	3.23	3.4	1.05
Q16	8.98	9.38	1.04	27.42	27.93	1.02
Q17	9.63	9.92	1.03	16.44	17.82	1.08
Q18	19.61	20.7	1.06	27.54	29.23	1.06
Q19	19.12	19.67	1.03	29.28	30.01	1.02
Q20	5.5	5.83	1.06	10.25	10.95	1.07
Q21	27.08	29.23	1.08	54.84	57.4	1.05
Q22	8.05	8.37	1.04	21.03	21.65	1.03
Geo. mean	6.309	6.673	1.05	13.266	13.968	1.05
Power	570.64	539.52		271.38	257.74	

Chapter 6

Conclusion and Future Work

This thesis introduced micro adaptivity, a new method to improve query execution performance and robustness. We identified and analyzed cases that justify the need for such a method. Addressing the scientific questions posed in Section 1.3, we showed how primitive performance is affected by implementation (Q1), by investigating different algorithms (e.g. data dependency, full computation), different optimizations (e.g. loop unrolling, fission) or different compilers. Additionally, we showed that performance is influenced by the execution context (Q2): hardware configuration, input data peculiarities like selectivity. This study formed the basis of our prototype implementation within the Vectorwise database engine. We modeled the underlying machine learning problem (Q3), which needs to be solved by this system, as a multi armed bandit problem and proposed a variation of the ε-greedy algorithm as a solution.

Preliminary tests showed that our micro adaptive system brings a performance improvement to the Vectorwise engine. Although on the TPC-H SF-100 benchmark the speedup is not exceptional, we believe that in other workloads the gains can be more significant.

6.1 Future work

6.1.1 New flavors

So far, we identified a few flavors (compiler flavors, data dependency, etc.) which already demonstrate the utility of micro adaptivity. In the future, more flavors should be discovered and the analysis tools created for this project make this process easier. Ways to automate the search for new flavors should also be sought.

6.1.2 Prefetching

Hash join operations are very costly on large data sets because of their random memory access patterns. For example, for many TPC-H queries the most time consuming primitives belong to join operations (bloom filter checks, hash table checks). Therefore, any improvement in these primitives would be noticeable in the overall query time. One way to optimize hash joins is by using memory prefetching [CAGM07]. The problem with this approach is that it is known to be unpredictable. Its success can depend on the platform, on the prefetch distance, on the system state, etc.. This is the type of situation where micro adaptivity can be very useful. We could add prefetching to hash joins without the risk of suffering a severe performance degradation on some machine. This would require an extension of the current micro adaptive framework.

Prefetching in one primitive could lead to benefits in another primitive, so the framework should be extended to support primitive groups. Flavors for the primitives inside a group are chosen so that the overall performance of the group is improved.

6.1.3 Bloom filters

One of the advantages of bloom filters is that they support a trade off between accuracy and performance, which can be exploited by micro adaptivity. The challenge is that usually a bloom filter has a build and a probe phase. Once the build phase is complete, the probe phase, which is the costly one, must use the same number of bits. One solution is to use multiple small bloom filters. The micro adaptive system will decide how many of them to probe. Probing just one would be faster but the system should also take into account the cost caused by false positives that reach the next stages of the hash join. This would also require the flavor groups extension.

6.1.4 Just in time compilation

One of the challenges of the Just-in-time (JIT) compilation system currently being developed for Vectorwise is determining when it is beneficial to use compilation. One of the conclusions of [Som] was that JIT is not always beneficial and it could even hurt performance sometimes. One solution is to try to create a cost model for JIT, but a simpler alternative is to use the optimizer from the micro adaptivity system.

6.1.5 Selection algorithm

For our prototype implementation of micro adaptivity we designed a simple algorithm similar to ε-greedy. However, there are other algorithms worth investigating, such as POKER, described in [VM05], which claims to be significantly more efficient than the algorithms we examined so far.

I would like to thank my supervisors, Marcin Żukowski and Peter Boncz, for giving me the opportunity to work on this project and be part of the Vectorwise team. I am truly grateful for their guidance and patience. It has been a pleasure to work here thanks to Gosia, Ala, Irish, Giel, Michał, Willem, Julek, Sandor, Michał jr., Christophe, David and the A-team - Adrian and Andrei. I also wish to thank Henri Bal for reviewing my thesis and helping me improve it.

I thank my parents and my brother Dragoş for their unrelenting support.

Bibliography

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

[APD00] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[Bon02] P. Boncz. Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications. 2002.

[CAGM07] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), August 2007.

[CG02] Jichuan Chang and Nikhil Gupta. More on conjunctive selection condition and branch prediction, 2002.

[Cod69] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[DIR07] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, January 2007.

[DMV$^+$08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing, Volume 3*, pages 1381+, 1998.

[Fog09] Agner Fog. Agner's cpu blog. Website, 2009. http://www.agner.org/optimize/blog/read.php?i=49.

[Fog12] Agner Fog. The microarchitecture of intel, amd and via cpus. Website, 2012. http://www.agner.org/optimize/microarchitecture.pdf.

[Gra94] G. Graefe. Volcano&#151 an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Rec.*, 20(2):268–277, April 1991.

[KSvdBB96] Martin L. Kersten, F. Schippers, Carel A. van den Berg, and Peter A. Boncz. *Mx documentation tool*, January 1996.

[LGP04] Xiaoming Li, Maria Jesus Garzaran, and David Padua. A dynamically tuned sorting library, 2004.

[LR85] T.L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[Rob52] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the AMS*, 58:527–535, 1952.

[Ros04] Kenneth A. Ross. Selection conditions in main memory. *ACM Trans. Database Syst.*, 29(1):132–161, March 2004.

[Som] J. Sompolski. *Just-in-time Compilation in Vectorized Query Execution*. PhD thesis.

[VM05] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *In European Conference on Machine Learning*, pages 437–448. Springer, 2005.

[Wat89] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989.

[Zuk09] M. Zukowski. *Balancing Vectorized Query Execution With Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, September 2009.