Micro Adaptivity in Vectorwise

Bogdan Răducanu
Actian
bogdan.raducanu@actian.com

Peter Boncz
CWI
p.boncz@cwi.nl

Marcin Żukowski[*]
Snowflake Computing
marcin.zukowski@snowflakecomputing.com

ABSTRACT

Performance of query processing functions in a DBMS can be affected by many factors, including the hardware platform, data distributions, predicate parameters, compilation method, algorithmic variations and the interactions between these. Given that there are often different function implementations possible, there is a latent performance diversity which represents both a threat to performance robustness if ignored (as is usual now) and an opportunity to increase the performance if one would be able to use the best performing implementation in each situation. *Micro Adaptivity*, proposed here, is a framework that keeps many alternative function implementations ("flavors") in a system. It uses a learning algorithm to choose the most promising flavor potentially at each function call, guided by the actual costs observed so far. We argue that Micro Adaptivity both increases performance robustness, and saves development time spent in finding and tuning heuristics and cost model thresholds in query optimization. In this paper, we (i) characterize a number of factors that cause performance diversity between primitive flavors, (ii) describe an ϵ-greedy learning algorithm that casts the flavor selection into a *multi-armed bandit* problem, and (iii) describe the software framework for Micro Adaptivity that we implemented in the Vectorwise system. We provide micro-benchmarks, and an overall evaluation on TPC-H, showing consistent improvements.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems - *Query processing*

Keywords: Adaptive; Self tuning; Query processing

1. INTRODUCTION

Adapting to unpredictable, uncertain and changing environments is one of the major challenges in database research. Adaptive query execution and query re-optimization have been valuable additions to database technology that aim to re-arrange the shape of a query plan in reaction to its execution so far, thereby enabling the system to correct mistakes in the cost model, or to e.g. adapt to changing selectivities, join hit ratios or tuple arrival rates. *Micro Adaptivity*, introduced here, is an orthogonal approach, taking adaptivity to the micro level by making the low-level functions of a query executor more performance-robust.

Micro Adaptivity was conceived inside the Vectorwise system, a high performance analytical RDBMS developed by Actian Corp. [18]. The raw execution power of Vectorwise stems from its *vectorized* query executor, in which each operator performs actions on vectors-at-a-time, rather than a single tuple-at-a-time; where each vector is an array of (e.g. 1000) tuples. This approach is a form of *block-oriented query execution* [11] in some existing database systems, and was shown to strongly improve performance [3].

Primitive Functions. In *vectorized execution* the basic computational actions for each relational algebra operator are implemented in *primitive functions*. Each such "primitive" works on input parameters that are each represented as a vector, i.e. an array of values, and similarly produces an array of output values. All relational operators such as Projection, Selection, Aggregation and Join, rely on such primitives to process data. The (non-duplicate eliminating) Projection operator is typically used to compute expressions as new columns, and each possible expression predicate combined with each possible input type typically maps to a separate primitive function (e.g. multiplication of doubles, addition of floats, subtraction of short integers, etc.). For Selection, the system contains primitives that for a boolean predicate compute a sub-vector containing the positions of qualifying tuples from the input vectors. For Aggregation, each aggregate function (sum, count, min, etc.) and possible input type leads to a primitive function that performs the work of updating an aggregate result. In addition, hash aggregation employs primitives for vectorized hash value computation, vectorized hash-table lookup and insertion – similar holds for the Hash-Join operator. In total, the Vectorwise system contains some 5000 different primitive functions.

stage:	preprocess	execute	primitives	postprocess
cycles	1679694	4321561972	3983412990	807654
%	0.03%	99.92%	92.17%	0.01%

Table 1: Example of time spent in different execution stages.

Consider a simple query that involves a Selection:

```
SELECT l_orderkey FROM lineitem WHERE l_quantity < 40
```

It calls a primitive that performs *smaller than* comparisons

[*]work performed while at Actian

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

on integer values. Table 1 shows a sample trace of this query. Almost all time is spent in the execute stage (99.92%) and within it, almost all time is spent in primitives (92.17%). Similar work division occurs in more complex query plans, including joins, aggregations etc. Thus, with primitives dominating the execution time, their speed is critical to overall system performance.

Given that the implementation of a vectorized primitive contains a loop over the input array(s), the advantage of the vectorized execution approach is that the overhead of calling the function is amortized over many tuples, thus strongly reduced compared with the tuple-at-a-time approach. Additionally, specific compiler techniques that revolve around the optimization of loop code are triggered, e.g. strength reduction, loop unrolling, loop fission, loop fusion but also the automatic generation of SIMD instructions. On the software engineering level, the effect of vectorization is a separation of control logic (the relational operators) and data processing logic (the primitives). As such, vectorized processing brings more than only performance advantages. One example is in performance monitoring and query plan profiling: vectorized primitives can be easily profiled, given the fact that significant CPU time is spent inside them, providing the ability to characterize all steps in a query plan to the level of CPU-cycles-per-tuple. This is less obviously done in tuple-at-a-time engines, where it would be too expensive to trigger performance measurement code around each functional call performing e.g. a single integer multiplication, because such performance monitoring code would likely be much more costly than the function being monitored. This ability to cheaply keep track of the performance of each call to a primitive function, enables our system to become adaptive to the observed cost of primitive calls.

Primitive performance. Primitive efficiency depends on the algorithm chosen to implement it and the way the code was compiled. But it is also influenced by the environment: hardware, data distributions, query parameters, concurrent query workload, and the interactions between these. The high complexity of computer systems, with their complex cache hierarchies, out-of-order execution capabilities and constraints, SIMD instruction support etc. combined with the dynamic aspects of the environments where the primitives are applied, make it impossible to correctly choose one optimal implementation even for a known workload.

Micro Adaptivity. We introduce the Micro Adaptivity framework which addresses the above problem. In this solution the query executor stores many available implementations ("flavors") of the same primitive, and during query execution dynamically chooses one flavor to use at any given moment, based on historical and current performance of different flavors. This provides adaptation to different and changing environments, and results in both a software performance improvement over a wide number of scenarios (adapting to different and changing hardware, data sets and tasks), but also reduces development effort in the query optimizer (as less time is spent in finding and maintaining heuristics and cost model thresholds).

Motivating Example: Branch vs No-Branch. A good example of context-dependent performance are branching and non-branching implementations of Selection primitives. The branching primitives use the *if* statements to test a predicate while the non-branching primitives use logical operators and index arithmetic to completely remove any branching. The selection primitive in Listing 1 accepts as arguments a vector `col` of `ints` and its size `n`, a constant `val`, and a vector `res` where to store the result. 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. The Branching implementation in Listing 1 uses a branch while the primitive shown in Listing 2 is branch-free (No-Branching). These implementations are functionally equivalent: they always produce the same result.

Modern CPUs attempt to predict the outcome of branches, so each CPU cycle next instructions can be pushed into its execution pipeline, even though the branch instruction has not finished (left the pipeline) yet. Accurate branch prediction is thus a necessary element to make pipelined CPU designs efficient, because in case of a mispredicted branch, the entire CPU pipeline must be flushed and delays occur. Prediction will be accurate when the branch behavior can be easily predicted from the previous few executions, i.e. if the branch condition is mostly true or mostly false. Consequently, the performance of the Branching selection primitive depends on the input data distribution; see [13].

Listing 1: Branching less-than Selection primitive

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

Listing 2: No-Branching less-than Selection primitive

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

The No-Branching implementation always performs the same number of operations, while with Branching, this depends on the data. If the data is such that the branch is almost never taken, then the Branching implementation will do less work, as it avoids executing the code that generates a result. What is the fastest implementation depends on the data, as shown in Figure 1, where *selectivity* is the probability that the branch condition is true. For very low and very high selectivities Branching is faster, while No-Branching is better otherwise. Note that the same experiment ran on different hardware tends to produce performance curves of similar shape, but with different cross-over points (see e.g. [3], Figure 2).

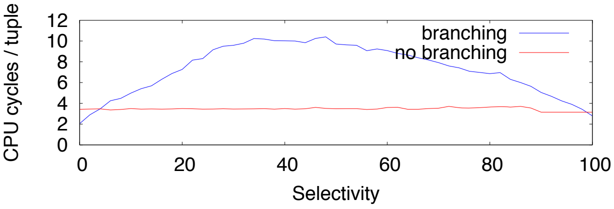

Figure 1: (No-)Branching primitive cost vs. selectivity [17].

One could hope that if the query optimizer could correctly estimate the selectivity for this query, it could pick the best solution (note that doing so would require significant tuning and instrumentation, because the cross-over points will vary

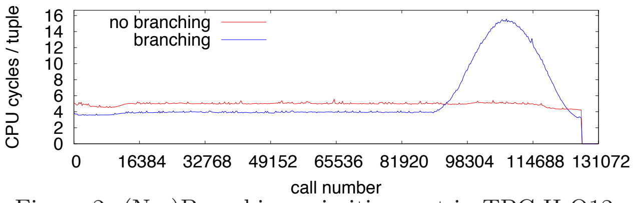

Figure 2: (No-)Branching primitive cost in TPC-H Q12.

between CPUs). These hopes are dashed in Figure 2, which plots the behavior of these primitives during the execution of TPC-H Query 12. Here, 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), however at the end it deteriorates to up to 16 cycles/tuple. What happens is that during almost the entire query the selection percentage is 100% but at the end it drops off towards 0%. During this drop, Branching strongly degrades, while No-Branching remains constant and thus becomes the best.

This example showed how two equivalent primitive implementations can perform differently, depending on selectivity (hence data distributions). Thus, statically determining a single best implementation when the database system is compiled will be sub-optimal. It also showed that even *during* execution, the context may change, so different primitive implementations may perform better during the lifetime of a query. Thus, determining a best implementation at query optimization time, even if we assume its cost could be predicted, will still not produce an optimal solution. Our proposal, Micro Adaptivity, addresses this challenge, because at the start of the query it will quickly determine that Branching selection performs best, but when its performance deteriorates, it will switch to No-Branching.

Contributions and Roadmap. Our overall contribution is the concept of Micro Adaptivity, where a database system comes equipped with many primitive flavors, between which it adaptively chooses during execution. This makes a system *self-tunable*, resulting in improved and more robust performance Additionally, it simplifies product engineering by removing the complex and error-prone decision process of which implementation of a given functionality to ship, improving performance in different scenarios and on different platforms, including the future ones, with as-of-yet unknown strengths and weaknesses.

Our technical contributions are enumerated as follows. In Section 2, we *(i)* characterize a number of factors that cause performance difference between primitive flavors, and thus provide an opportunity for Micro Adaptivity to make performance better and more robust. Then in Section 3 we *(ii)* describe an ϵ-greedy learning algorithm that casts flavor selection into a *multi-armed bandit* problem, and *(iii)* describe the software framework for Micro Adaptivity that we implemented in Vectorwise, which comprises the techniques used to generate many different primitive flavors, data structures to manage flavors inside the query executor, and changes inside the query evaluator to perform the adaptive choosing of the best flavor. Throughout the paper we provide micro-benchmark results, and in Section 4 evaluate its overall effect on the TPC-H workload. In Section 5 we discuss related work in adaptive query execution, as well as adaptive computational libraries from outside the realm of database system. Finally, in Section 6 we outline our conclusions and future work.

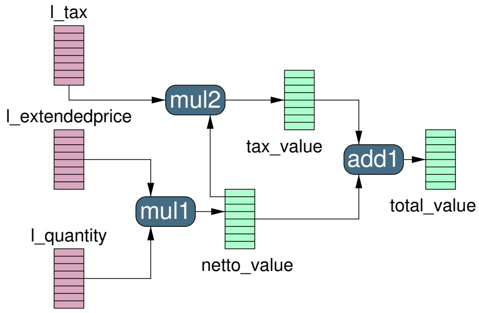

Figure 3: Projection expression with two "mul" instances.

1.1 Primitives in Vectorwise

To help with the understanding of Micro Adaptivity and our performance results, we first provide additional context on the role of vectorized primitives in Vectorwise.

Listing 3: Query with 2 multiplication primitive instances

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

Primitive Instances. The query in Listing 3 calculates a `total_value` using a relational Projection operator, involving three arithmetic functions, leading to three *primitive instances* in the query plan. We use the term "primitive instance" to distinguish calls to the same "primitive function" from different places in query plans. In this example, there are two instances of the same primitive that multiplies two integers (see `mul1` and `mul2` in Figure 3). It is important to make this distinction because different primitive instances process different streams of data, hence their performance behavior is different.

Listing 4: map_mul primitive with selection vector

```
map_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]; // get position 'i' from sel
            res[i] = col1[i] * col2[i];
        }
    else // no selection vector
        for(i = 0; i < n; ++i)
            res[i] = col1[i] * col2[i];
}
```

Selection Vector. Listing 4 shows Vectorwise code for the integer multiplication primitive. The name of primitive functions used by the Projection operator starts with `map`, Selection primitives start with `sel`, Aggregation with `aggr`, etc. Subsequently, `_mul` is derived from the predicate name (multiplication) and what follows are parameters, e.g. `_int_col` denotes a non-constant parameter of type `int` – a constant parameter would be `_int_val`. Each (non-constant) parameter is a vector, i.e., a simple array of values.

Many primitives accept an optional *selection vector* (last parameter) which defines the subset of input tuples that needs to be processed. The else-body just computes results for the selected tuples ("selective computation") – see also Figure 7 (left). Selection primitives such as those in Listing 1 and 2 generate these selection vectors, which are then passed to other primitives. This avoids copying all column vectors after a Selection to eliminate the tuples that did not pass.

machine:	**1**	**2**	**3**	**4**
CPU Manufacturer	Intel	Intel	AMD	Intel
CPU Architecture	Nehalem	Core2	Egypt	Sandy Bridge
Last Level Cache	12MB	4MB	1MB	8MB
RAM Size	48GB	8GB	64GB	16GB

Table 2: Test Machines

compiler	**version**	**flags**
gcc	4.6.2	-O6 -fomit-frame-pointer -falign-functions=4 -falign-loops=4 -falign-jumps=4 -ftree-vectorize -fexpensive-optimizations -frerun-cse-after-loop -funroll-loops -frerun-loop-opt -finline-functions
icc	11.0	-O5
clang	3.1	-m64 -no-integrated-as -O3

Table 3: Compilers used in Tests

Flavors. The primitive signature string is used in the *Primitive Dictionary* component of the query evaluator to implement function resolution; hence this dictionary maps signature strings into function pointers. As part of the Micro Adaptivity feature, we changed the Primitive Dictionary so as to allow it to store *multiple* function pointers for each signature; this allows us to have multiple implementations (*Primitive Flavors*) for the same primitive. Primitive Flavors are kept using additional meta-information, that might include: flavor source (e.g. code version, compiler used, etc.), number of times it was used and some performance characteristics, both delivered with primitive as well as gathered during runtime. The Primitive Dictionary provides a registration mechanism, where a software component can register additional Primitives and Primitive Flavors dynamically, allowing e.g. loading additional Primitive Flavors on startup or even while the system is active.

Approximated Performance History (APH). For each primitive instance in a query plan, Vectorwise keeps profiling data gathered at every call to the function. This data comprises the total amount of tuples processed, the total amount of primitive calls and the total CPU cycles spent in that primitive instance. As part of the Micro Adaptivity project, we expanded the profiling to keep not only totals, but also a performance histogram over time. Given that the typical vector size used in Vectorwise is around 1000 tuples, a query operator in an analytical query that processes e.g. 100M of tuples will call its primitives 100K times. Keeping a histogram of 100K measurements for each primitive instance is too heavyweight, so we create an *Approximated Performance History* (APH) of just 512 buckets. Initially, after each primitive call, Vectorwise adds one new bucket with performance statistics, but when all 512 buckets are used neighbors are merged such that 256 occupied buckets remain. From then on, each bucket contains data from two calls; when the process repeats k times, the APH still has at most 512 buckets, each representing 2^k subsequent calls. Figure 2 showed the APH with in the X-axis the amount of function calls made; in Figures 4 and 11 we omit this amount of calls; the X-axis just represents the query lifetime.

2. PERFORMANCE DIVERSITY

In this section we provide examples of factors that influence primitive performance. Two factors are related to creating different flavors for the same primitive: *(i)* algorithmic variations and *(ii)* using different compilers or compiler switches. A further factor is *(iii)* input data distribution; we already saw in Figure 2 that this can even cause changes in behavior such that different phases of a query, favor a different flavor; this will re-occur in Figure 4 (b) and (d). The final factor considered is *(iv)* different behavior on different hardware platforms. For this purpose, we experimented on four different machines as listed in Table 2. Finally, this section shows that even to have a query optimizer choose just a single flavor for each primitive instance before query execution (which is sub-optimal anyway), the cost modeling task for this is daunting because algorithmic variant, data distribution, compilation method and hardware all interact in determining primitive cost. These points together make the case for the Micro Adaptivity framework.

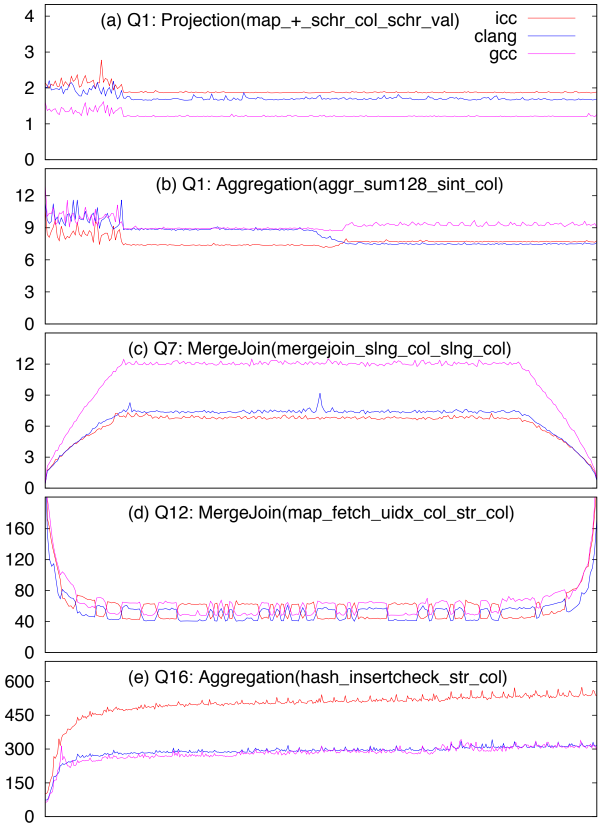

Figure 4: **Compiler differences**: sample APHs from TPC-H SF100 on machine 4 (avg. cycles/tuple during a query)

Compiler Variation. One of the easiest ways to obtain different flavors is to compile with different compilers and switches. Vectorwise can be compiled using a variety of C/C++ compilers, and we took those compilers and flags normally used to produce optimized Vectorwise builds (those supplied to customer) on Linux, as displayed in Table 3.

Figure 4 shows some selected Approximate Performance Histories (APHs) from running the TPC-H queries with the 100GB size on machine 4. Sub-figures (a) and (b) show that within Q1, the addition primitive compiled by gcc is fastest (a), while the integer aggregation primitive used inside this same query compiled by icc consistently beats gcc (b). Hence, there is no single best compiler for Q1. In (a) the performance difference is 30%. Even more interestingly, we see that the clang compiler for the aggregation primitive, while initially on the gcc level, halfway crosses over and becomes fastest, edging out icc.

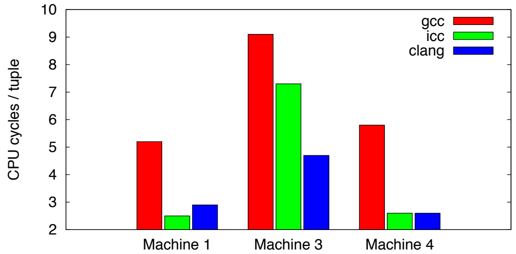

Figure 5: mergejoin: best compiler depends on machine

The third sub-figure (c), showing a mergejoin primitive in action in Q7 has gcc being almost 90% slower than icc and clang. The rising and falling curves in the border regions of the mergejoin are caused by the average population of the selection vectors on the join inputs, which result from a prior range-selection on a date column that has data locality. This means that at first no tuples qualify, then a rising percentage of tuples qualify (the left border), until on average all tuples qualify (the plateau), followed by the right border region with a falling percentage of qualifying tuples. The same kind of phenomenon causes performance degradation in the border regions of sub-figure (d). Selection vectors in these borders start out really small, with only a handful of tuples selected; in which case there is too little function call overhead amortization (because selective computation occurs on the only few selected tuples). In case of (c), the mergejoin does move faster through the inputs at the borders because it can then skip and avoid result generation due to tuples being not-selected.

Sub-figure (d), which depicts a "fetch" primitive that just copies selected values from an input to an output vector, has remarkable behavior: at exactly the same switch points, either the gcc or clang primitive achieves "best" behavior which seems to fall on a smooth curve; whereas the other compiler then also achieves performance forming another smooth curve which is 30% slower – icc being in the middle of both flavors but never the best.

Finally, sub-figure (e) shows an Aggregation primitive that inserts a string key in a hash-table; here performance gradually deteriorates as the hash-table grows and (presumably) cache and TLB misses increase in cost. For some reason, though, the icc compiled primitive is twice slower than those compiled by gcc and clang.

The unexplained behavior of the mergejoin primitive in Figure 4 (c) triggered some micro-benchmarking on multiple machines. The micro-benchmarks on the mergejoin primitive in Figure 5 confirm the results in Figure 4 (c): on machine 1 icc is much faster, however we see that on machine 3 (AMD), it is actually substantially slower than clang.

Without the ambition to understand everything at the deepest level[1], we can conclude that compilers cause significant variation in primitive performance, and the code they produce behaves differently on different machines, and even on a single platform in a single primitive instance in a single query there may not be a single best compiler.

We now describe a few ideas we pursued to generate different algorithmic primitive variations.

Branch vs. No-Branch. In the Introduction, we have already discussed Branch vs. No-Branch implementations of Selection primitives, so we will not repeat this here.

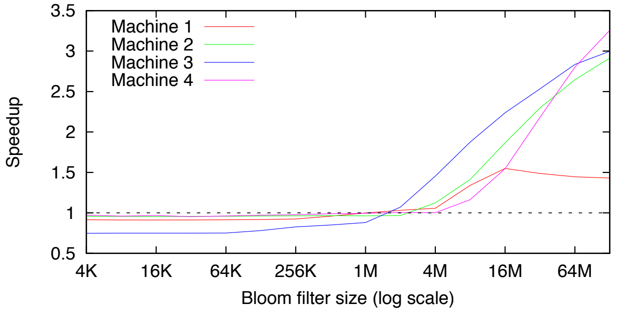

Figure 6: sel_bloomfilter speedup with loop fission

Loop Fission. Vectorwise uses bloom filters to accelerate hash-table lookups in situations where the key is often not found. Using a bloom-filter is faster because hash-table lookup is CPU-intensive and the hash-table may not fit in the CPU cache – whereas a bloom filter might fit as it is much smaller, and lookup is quick and simple. Still, the bloom filter may not fit the CPU cache, and the lookup primitive displayed in Listing 5 then incurs a cache miss in the `bf_get()` call. Note, that this is a Selection primitive, identifying tuples surviving the bloom-filter check, and uses a No-Branching code style.

Listing 5: Original bloom filter primitive

```
size_t sel_bloomfilter_sint_col(size_t n, size_t* res,
                                 char* bitmap,sint* keys)
{   size_t i, ret = 0;
    for (i = 0; i < n; i++) {
        slng hv = bf_hash(keys[i]);
        res[ret] = i;
        ret += // loop dependency
          bf_get(bitmap, hv);  // cache miss
    }
    return ret;
}
```

Listing 6: Bloom filter primitive with Loop Fission

```
size_t sel_bloomfilter_sint_col(size_t n, size_t* res,
                                 char* bitmap,sint* keys)
{   size_t i, ret = 0;
    for (i = 0; i < n; i++) {// independent iteration
        slng hv = bf_hash(keys[i]);
        tmp[i] = bf_get(bitmap, hv); // cache miss
    }
    for (i = 0; i < n; ++i) {
        res[ret] = i;
        ret += tmp[i];
    }
    return ret;
}
```

The loop-fission optimization of this bloom-filter check is displayed in Listing 6. Rather than identifying the selected tuples inside the same loop, it first just collects the boolean result of `bf_get()` in a temporary array, and then selects from there in a separate loop. The idea behind this loop-fission variant is that it removes all dependencies between iterations of the first loop.

The loop-fission variant, when it sustains a cache miss in `bf_get()`, allows the CPU to continue executing the next loop iteration(s), leveraging the large out-of-order execution capabilities of modern CPUs (> 100 instructions). This way the CPU will get multiple (up to 5, on Ivy Bridge) loop iterations in execution at any time, leading to 5 concurrent outstanding cache misses, maximizing memory bandwidth utilization. In contrast, the non-fission variant causes the iterations to wait on each other due to the loop-iteration

[1] Authors need to admit avoiding this temptation was an extraordinary effort on their side.

Table 4: map_mul: hand vs compiler unrolling (cyc/tuple)

Compiler / Hand		unroll-on		unroll-off	
		SIMD	no SIMD	SIMD	no SIMD
Machine 1:	unroll_8	1.73	1.73	1.73	1.73
	no_unroll	1.03	1.74	1.18	2.59
Machine 3:	unroll_8	2.02	2.02	2.02	2.02
	no_unroll	3.61	2.15	3.55	4.03

dependency, thus achieves less concurrent cache misses and therefore lower memory throughput.

We performed micro-benchmarks where we varied the number of unique keys from 2^{12} to 2^{27}, which required bloom filters with sizes from 4KB to 131072KB. Figure 6 shows the speedup obtained by fission, versus bloom-filter size, indicating a strong relation. For large bloom filters, fission performs better, indeed sometimes 50% faster; whereas for small bloom-filters where there are no cache misses fission can be slower, sometimes by 15%. The main point, though, is that the cross-over point depends on the machine: machine 1 crosses over at 1MB, but machine 4 at 4MB. One can conjecture that accurate CPU cache cost modeling could allow a query optimizer to pick the best variant, but we argue that it is hard to make such decisions robustly (also considering the program will run on future hardware) as the observed cross-over points do not trivially follow from the CPU cache sizes in Table 2.

Listing 7: Template-generated loop with hand-unrolling

```
#define BODY(i) res[i] = a[i] * b[i]
for(i = 0; i+7 < n; i+=8){
    BODY(i+0); BODY(i+1); BODY(i+2); BODY(i+3);
    BODY(i+4); BODY(i+5); BODY(i+6); BODY(i+7);
}
for(; i < n; i++)
    BODY(i);
```

Hand-Unrolling. The primitives in the Vectorwise engine are template-generated. The templates are used to provide type-specific versions of the same function (e.g. multiplication of integers, of doubles, of short integers, etc.). Additionally, these template macros instantiate all possible parameter combinations of vector vs. constant (e.g multiply values from two vectors, a vector with a constant or a constant with a vector). Template macros insert a body action, such as the multiplication of values, in a loop over the input vector(s). By changing these templates, as in Listing 7, we manually introduced a well known *loop-unrolling* optimization into all Vectorwise primitives – we call this *hand-unrolling*. In our experiments we consider hand unrolling by a factor of 8 (unroll_8) and no hand unrolling (no_unroll).

We performed micro-benchmarks to study the interaction between gcc compiler switches (see Table 3) for SIMD (-ftree-vectorize) and compiler-enforced unrolling (-funroll-loops) versus hand unrolling. The primitive being tested is integer multiplication from Listing 4, without selection vector, in which case the task is to multiply two dense arrays. Because of hand unrolling, gcc was unable to vectorize or unroll the loop (verified in the assembly and also reflected in the times measured). Table 4 shows that, surprisingly, for machine 3 unrolling is better than SIMD, while for machine 1 SIMD is clearly faster. We conclude that hand-unrolling does have effects, but it is hard to predict.

Full computation. Many primitives in Vectorwise accept a *selection* vector argument that contains the positions of the tuples in the current vector that need to be processed by the primitive.

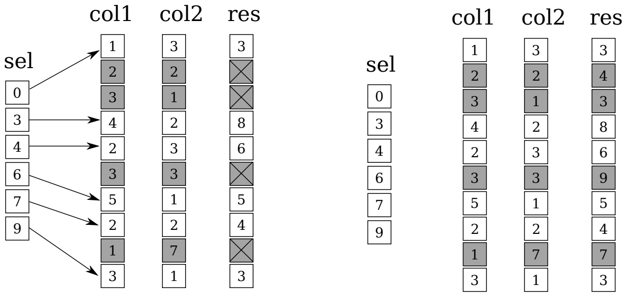

Figure 7: selective (left) vs. full computation (right)

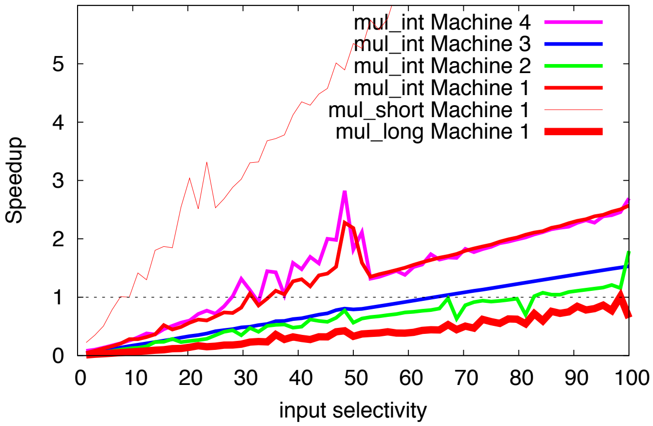

Figure 8: map_mul: full-computation speedup.

Figure 7 (left) shows that such "selective computation" only processes the required tuples (marked here with white background) and the result vector will have undefined values in the positions not in the selection vector (gray background). For some operations it is possible to ignore the selection vector and still produce correct results. The multiplication primitive can process all tuples, even though some of the results are not needed. Even though this strategy performs more work, it can be faster thanks to SIMD, because selective computation as in Figure 7 (left) is not being SIMD-ized by compilers, while *full computation* as in Figure 7 (right) trivially maps to SIMD, such that compilers generate it. Figure 8 shows the improvement obtained using this *full computation* policy compared with selective computation, as a function of the selectivity, on the various machines. It turns out that for 32-bits `int` multiplication full computation is faster on machine 1 if 30% or more tuples are in the selection vector; however this cross-over percentage is 80% on machine 2. We also show for machine 1 the performance for 16-bits `short` and 64-bits `long` multiplication; where the former benefits already from 10% on (and the benefit is much stronger), while the latter never benefits from full computation (and SIMD). So, if this choice for the best primitive would have to be made by an optimizer, or by a heuristic inside the code, it would be quite difficult to pick proper boundaries as it depends on the hardware, compiler settings, data type and possible hand unrolling.

Conclusions. We were able to create many primitive flavors with limited effort, and these show a jungle of hard-to-predict performance variation. This provides ample opportunity to improve performance robustness for our Micro Adaptivity framework – described in the rest of this paper.

3. MICRO ADAPTIVITY FRAMEWORK

The Micro Adaptivity framework modifies the architecture of Vectorwise in various places, including the template-based primitive generation in its system compilation environment, its dynamic library loader and, most importantly, in its query executor. Concerning the latter, the most important change is the introduction of a *vw-greedy* learning-algorithm inside the expression evaluator. This learning algorithm is the heart of Micro Adaptivity, which casts the optimization problem of selecting the best flavor at runtime into a *multi-armed bandit* (MAB) problem [15]. We now discuss these changes in turn.

3.1 Flavor Libraries.

One of the most effective and easy mechanisms to obtain different flavors is to compile the Vectorwise primitives with different compilers and compiler switches. The process of creating a final binary system therefore now takes object code from multiple build environments, extracting the flavor library from each. Flavor libraries are loaded with the POSIX `dlopen` function during the Vectorwise kernel initialization, before any query is received. 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 that library, whereas without the argument, symbols are resolved using previously loaded libraries. This mechanism is sufficient for implementing the two functions `get_random_flavor()` and `get_best_flavor()` needed for the learning algorithm (see later Listing 8).

As we described in the context of Listing 7, the Vectorwise primitive source code is generated using a template-expansion tool that allows to specify basic code patterns. This tool expands 340KB of template code 50x into 16MB of primitive source code, even without Micro Adaptivity. We modified these templates by adding the described algorithmic variations, activated by compilation flags. For instance, by defining `-DFULL_COMPUTATION_FLAVOR`, the whole primitive library is compiled such that all Projection primitives follow the full computation strategy (selective computation is default). For No-Branching and loop fission that affect Selection primitives there are similar defines. Note that these defines affect disjoint sets of primitives, hence only two different builds are needed to generate all variants. On top of this there is a compilation flag to enable hand-unrolling (which affects all primitives), so we need four different builds to generate all previously described algorithmic variations.

3.2 Learning Algorithm

The key component in the Micro Adaptivity is the addition of a learning algorithm inside the expression evaluator. The expression evaluator is the component that calls implementation functions for primitives. It is activated as a part of vectorized execution, (recursively) evaluating an expression for a vector of input values, producing a vector of output values. Whereas in unmodified Vectorwise this used to call the same primitive function for a particular expression for every vector, we now added a *learning algorithm* that at each call chooses one of the flavors available for that primitive.

We presented earlier how performance can be dependent on data distributions, query parameters, hardware etc., so supervised learning strategies (e.g. artificial neural networks) which rely on an off-line training phase are not a good fit for our learning algorithm, because we will not be able to build a representative training data set for every possible workload. Therefore, we seek a real-time learning algorithm, i.e. one that is able to adjust to sudden context changes.

Reward and Regret. 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 learning algorithm, has to be the flavor that will lead to the maximum total reward. After each choice, the system updates its 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* [12].

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 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 its expected value. For certain reward distributions, it can be proved that the regret grows at least logarithmically with the number of iterations, i.e. $R_T = \Omega(lnT)$ [8] .

Exploitation vs. exploration After a number of iterations, the system will have some knowledge about the rewards of each flavor. A danger is that if the system keeps choosing the same flavor, it will build more knowledge about it, but the knowledge about the other flavors will become stale. In the meantime, contextual parameters that determine the primitive performance (e.g. selectivity, cache/memory traffic) may change, so another flavor could become the best, yet 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. Of course, this cannot be done too often, because it is likely that the chosen flavor is indeed not optimal, so it might 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 issue is to determine how to balance exploration and exploitation.

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 [2], 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. 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 tweak to becoming non-stationary resistant.

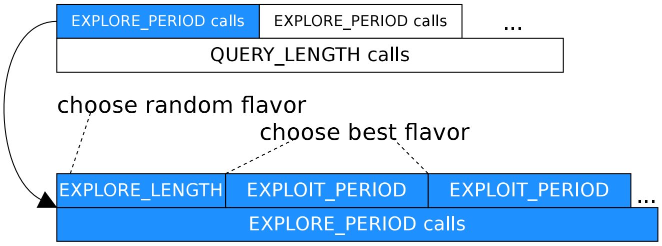

Figure 9: `vw-greedy` algorithm pattern.

ε-greedy strategy 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 [16]. 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. 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 in case performance characteristics change.

The ε-greedy strategy is not optimal, because its regret increases linearly with the number of iterations. With the ε-decreasing strategy, ε is decreased after every iteration and [2] shows that this achieves optimal regret if ε decreases at a rate of $1/n$, n being the number of calls so far.

vw-greedy We designed the `vw-greedy` algorithm (Listing 8), based on ε-greedy, but with the following differences:

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

The goal of these changes is to improve handling of the non-stationary case. The standard ε-greedy method computes the mean performance for each flavor using all calls since the beginning of the query. In the stationary case, this mean value will eventually converge because the true mean is constant. But in our case, we compute the mean of only recent calls, making the algorithm handle sudden changes in performance. 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.

`vw-greedy` 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 $\frac{EXPLORE\_LENGTH}{EXPLORE\_PERIOD}$. 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 9.

Listing 8: `vw-greedy` algorithm

```
function vw-greedy(prim, tuples, cycles) {
    // classical primitive profiling
    prim.tot_cycles += cycles;
    prim.tot_tuples += tuples;
    prim.calls++;

    // vw-greedy switching
    if (prim.calls == prim.calc_end) {
        // calc average cost in previous period
        prim.flavor.avg_cost =
          (prim.tot_cycles - prim.prev_cycles)/
           (prim.tot_tuples - prim.prev_tuples);

        if (prim.calls > prim.explore_period){
            // perform exploration
            prim.explore_period += EXPLORE_PERIOD;
            prim.flavor = get_random_flavor()
            prim.calc_end = EXPLORE_LENGTH;
        } else {
            // perform exploitation
            prim.flavor = get_best_flavor()
            prim.calc_end = EXPLOIT_PERIOD;
        }
        // ignore first 2 calls to avoid
        // measuring instruction cache misses
        prim.calc_start = prim.calls + 2;
        prim.calc_end += prim.calc_start;
    }
    if (prim.calls == prim.calc_start) {
        prim.prev_tuples = prim.tot_tuples;
        prim.prev_cycles = prim.tot_cycles;
    }
}
```

The normal Vectorwise profiling already kept per primitive the cumulative cycles `prim.cycles`, the cumulative number of tuples processed `prim.tuples` and the cumulative amount of calls `prim.calls`. During a given phase, the chosen flavor is called a number of times (either `EXPLORE_LENGTH` or `EXPLOIT_PERIOD`). At the end of the phase, we choose a new flavor: either a random flavor with `get_random_flavor()` or the "best" one with `get_best_flavor()`. The best flavor is the flavor with the lowest average cycle/tuple cost. This average cost of a flavor is computed only for the calls in the same phase, at the end of each phase, and ignoring the first two calls since these tend to start off higher due to instruction cache misses. The `vw-greedy` function assumes that `EXPLORE_PERIOD` is larger than `EXPLOIT_PERIOD` and both are multiples of `EXPLORE_LENGTH`, which in turn should be > 2. In our implementation, all parameters are powers of two, such that the if-then tests are a comparison between a constant and a bitwise-and; i.e. low-cost operations.

Demonstration. We now demonstrate the algorithm on a synthetic example scenario. We used `EXPLORE_PERIOD`=1024, `EXPLOIT_PERIOD`=256 and `EXPLORE_LENGTH`=32 as settings.

The example query scenario has a primitive with three flavors that are non-stationary in performance, and where one is the best at the start and the end of the query, but another one is better in the middle. In Figure 10, the orange curve is the performance trace of the `vw-greedy` algorithm. 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 short spikes in the performance. Most importantly, we see that our algorithm consistently covers the minimum of the various performance lines, switching in the middle segment to the primitive that is the best there.

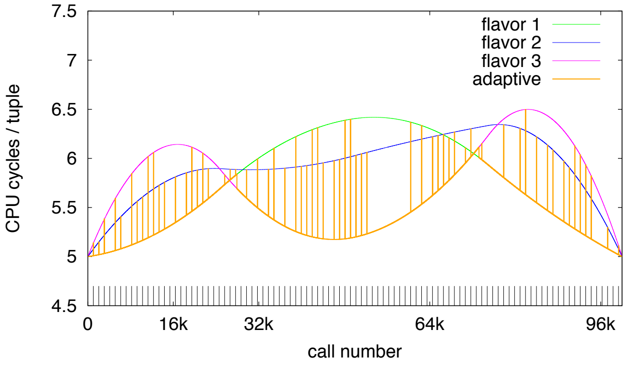

Figure 10: `vw-greedy` in action on 3 flavors (APHs).

Algorithm	Absolute/OPT	Relative/OPT	Average
vw-greedy(1024,8,2)	1.015	1.011	1.013
eps-first(0.001)	1.012	1.016	1.014
eps-greedy(0.001)	1.015	1.015	1.015
vw-greedy(2048,8,1)	1.015	1.015	1.015
eps-decreasing(1.0)	1.015	1.016	1.015
eps-decreasing(0.1)	1.015	1.016	1.015
vw-greedy(2048,8,2)	1.018	1.013	1.015
eps-greedy(0.05)	1.017	1.015	1.016
eps-decreasing(5.0)	1.022	1.015	1.018
eps-greedy(0.1)	1.018	1.021	1.019
eps-first(0.05)	1.020	1.019	1.019
eps-first(0.1)	1.017	1.023	1.020

Table 5: Performance of MAB algorithms with different parameters (as factor of OPT), simulated on a TPC-H trace.

Simulations on traces. We studied the behavior of various MAB algorithms using profile data gathered from a TPC-H SF-100 run on Machine 1. This workload contains over 300 primitive instances, and the number of calls to these primitives ranged from $16K$ to $32K$. For this evaluation we used only three flavors generated by using different compilers (gcc, icc and clang). This profile data consists of measurements of all three flavors, each taken in a run where the systems sticks to one flavor only. To decide how good an algorithm is, we compare its performance with OPT, i.e. the theoretical optimal performance obtained by picking for each call the primitive flavor that performed best.

We compute two scores: an absolute score and a relative score. The absolute score sums all flavor times selected by the MAB algorithm being tested and then divides this by the sum of the times that OPT selects. The Absolute/OPT score for a query is the ratio between the overall workload time obtained by the algorithm and the optimal performance, i.e. the lower the score, the better. The Relative/OPT score looks at each primitive instance individually: it divides the performance achieved on that primitive instance by the MAB algorithm and divides it by the cost achieved by OPT for that primitive instance. Subsequently, the relative score is the average of all these factors. The difference between Absolute/OPT and Relative/OPT is that the former shows the overall impact of the MAB algorithm on the entire workload, whereas the latter characterizes the average benefit on each primitive. For instance, if some primitives do not benefit much from Micro Adaptivity, but happen to take many cycles in this workload overall, they may cause the Absolute/OPT score to be lower than the Relative/OPT score.

To see how our algorithm compares with alternatives, we evaluated many parameter settings of each algorithm on this data set. The top-12 average scores are given in Table 5. The parameters for `vw-greedy` are (`EXPLORE_PERIOD`, `EXPLOIT_PERIOD`, `EXPLORE_LENGTH`). For the others, we chose parameters similar to the ones evaluated in [15]. For the ε-greedy and ε-first algorithms we also chose one parameter that we thought is equivalent to the `vw-greedy` parameters (e.g. for `EXPLORE_PERIOD` of 1024 we chose $\varepsilon = 0.001$).

This simulation shows that the best overall performance can be obtained by the `vw-greedy` algorithm, suggesting parameters (1024,8,2). We note that on this benchmark the scores for all algorithms are quite similar because testing only the compiler flavors does not favor `vw-greedy`, as compiler differences less often lead to cross-over points than the other algorithmic variants. The fact that cross-over during a query is often not needed shows because ε-first finishes as a runner-up, while it only tests all flavors at the beginning and then sticks to its choice. This simulation did trigger us to add an initial exploration phase to `vw-greedy`: in the first `EXPLORE_PERIOD` calls, we also start by first testing all available flavors, for `EXPLORE_LENGTH` each. Algorithm Listing 8 can be trivially extended with this functionality.

4. EVALUATION

To evaluate Micro Adaptivity, we measured the impact on the TPC-H benchmark[2] at the 100GB scale factor on machine 1. Vectorwise can run queries in parallel on multiple cores, but for these experiments we configured it to run on a single core, to get more accurate measurements.

4.1 Detailed Impact of Micro Adaptivity

We separately benchmarked the five sources of creating different primitive flavor sets explored in this paper, i.e. (No-)Branching Selection primitives, the use of loop fission in hash-table lookup, different compilers, full computation and with or without hand loop unrolling. We measured the times and speedups for those primitives that are actually targeted by these flavor sets. For each flavor set, we first tested each flavor individually and then ran Micro Adaptivity with the whole set. Using the Approximate Performance History (APH) statistics that Vectorwise gathers, we computed an approximated optimal (OPT) performance for each primitive instance, by taking the minimum time among all flavors for each APH bucket. This time is shown in the OPT column in the following tables.

Each table shows the number of CPU cycles spent by one flavor, in total over all the TPC-H queries, but only for the primitives actually affected by the flavor set. The first column contains the default behavior and shows the total amount of cycles spent in the affected primitives, as well as the percentage of the overall workload – in order to give an impression of the potential impact of accelerating these primitives. The other columns in the table show the performance improvement factor compared over the first column, obtained by switching to the non-default flavor, by Micro Adaptivity choosing between them in realtime, and by OPT.

[2] Note that we used TPC-H schema and queries for demonstration purposes only, and these results should in no way be considered to be compliant or official TPC-H results

Always Branching	Always No-Branching	Micro Adaptive	OPT
57 bn. (8.58%)	1.12	1.22	1.23

Table 6: Cost in cycles (and as % of workload), and improvement factor in primitives with (No-)Branching flavors.

only gcc	only clang	only icc	Micro Adaptive	OPT
348 bn. (51.29%)	0.99	0.99	1.11	1.11

Table 7: Cost in cycles (and as % of workload), and improvement factor in primitives with Compiler flavors.

Never Loop Fission	Always Loop Fission	Micro Adaptive	OPT
71 bn. (11.30%)	1.40	1.57	1.57

Table 8: Cost in cycles (and as % of workload), and improvement factor in primitives with Loop Fission flavors.

Selective Computation	Full Computation	Micro Adaptive	OPT
33 bn. (5.25%)	0.57	1.09	1.10

Table 9: Cost in cycles (and as % of workload), and improvement factor in primitives with Full Computation flavors.

unroll_8	no_unroll	Micro Adaptive	OPT
348 bn. (51.29%)	1.01	1.07	1.07

Table 10: Cost in cycles (and as % of workload), and improvement factor in primitives with Hand-Unrolling flavors.

Branching vs. No-Branching. In this TPC-H run, less than 10% of CPU time was spent in Selection primitives. Thus, speeding up selections can bring only moderate increase in the overall TPC-H score. But, we can see how Micro Adaptivity takes advantage of Branching vs No-Branching algorithmic variants in selection primitives. Table 6 shows that using No-Branching always is faster than Branching always, making selections 12% faster. Micro Adaptivity makes use of both implementations to obtain a 22% performance increase, which is close to the optimal.

Compilers. Table 7 presents the speedups of all compilers compared to gcc. Apparently, all compilers lead to similar performance, so normally one does not gain much by changing the compiler. However, using Micro Adaptivity, we get a noticeable speedup of 11%, which means that at the primitive level there are in fact differences between the compilers. The primitives affected by Micro Adaptivity use 51% of the total CPU time so we expect to get 5% improvement overall. It may be surprising that varying compilers does not affect all primitives (which account for > 90% of all CPU cycles). The reason is that in the Vectorwise codebase, some operators still bypass the expression evaluator for calling certain primitives. This holds for all decompression code in Scans, and also for (important) primitives in hash-table lookup and insertion, affecting Hash Joins and Aggregations. This requires some additional engineering to fix.

Loop fission. From Table 8 we see that loop fission is a good optimization for bloom filter lookups, reducing the total CPU time by a factor of 40%. But there is more to gain here by using Micro Adaptivity, which brings the reduction in CPU cycles spent on bloom filter lookups to 57%.

Full Computation. Full Computation is a risky optimization as it can lead to significant performance degradation. In Table 9 we see that if it is forced always on, there is a performance degradation of 43% in the map primitives that are affected by full computation. Individual primitive instances can be 13 times slower! With Micro Adaptivity, we mitigate this risk and exploit any possible speedup that this optimization may bring. In this TPC-H run we get a speedup of 9%, compared to not using full computation at all. Unfortunately, in TPC-H, map primitives are responsible for only a small fraction of the query time, so the impact of this optimization is not large.

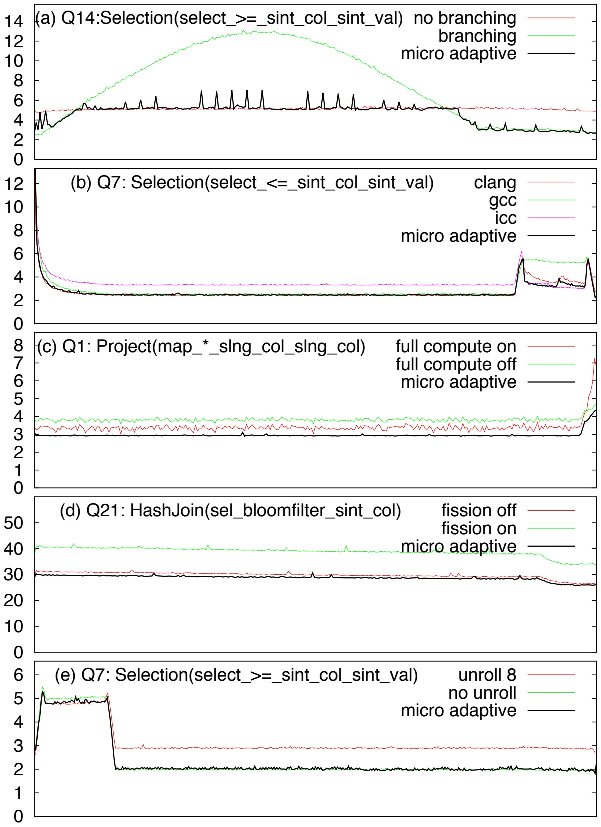

Figure 11: **Micro Adaptive execution**: sample APHs from TPC-H, machine 1 (avg. cycles/tuple during a query)

Hand unrolling. Hand unrolling is on by default in Vectorwise (with unroll factor 8) but it turns out that on this workload it would be 1% faster to leave it off always. However, because hand unrolling is sometimes better and sometimes worse, Micro Adaptivity can come up with 7% improvement, the optimum.

Sample Performance Histories. Since the workload contains more than 300 primitive instances, we only show a few samples of how Micro Adaptivity behaves; one for each flavor set. Figure 11 shows that Micro Adaptivity closely tracks the lower bound of any of the different flavors, switching from one flavor to the other when that is beneficial.

The astute observer can see in case (a) that Micro Adaptivity is more quick in detecting *deterioration* of the current algorithm, than in detecting *improvement* of a flavor that is not considered best at the moment: it immediately switches from Branching to Non-Branching, but takes a little more time to switch back. This follows from the `vw-greedy` algorithm, as finding out about a better-performing alternative takes multiple `EXPLORE_PERIOD` phases, while detecting deterioration of the current best flavor happens every `EXPLOIT_PERIOD` calls, which is a smaller timeframe. This is hard to avoid, because checking non-best flavors usually causes regret and therefore should be limited.

4.2 TPC-H Overall

We now look at the overall improvement of all individual TPC-H queries and the overall Power score (i.e. the geometric mean), comparing Micro Adaptive Vectorwise with standard Vectorwise with and without *heuristics*.

Heuristics. A competing approach to Micro Adaptivity is to try to invent heuristics that hard-code the cross-over points for various algorithmic variants. This is not really possible for flavors stemming from different Compilers, nor really for Hand Unrolling. However, one could for instance hard-code to use No-Branching selection implementations between 10% and 90% observed selectivity – note that the selectivity is always available to the primitives from comparing the selection vector length to the vector-size. Similarly, above 30% selectivity a primitive like `map_mul` in Listing 4 could ignore the selection vector by setting `sel = NULL`, effectively following the Full Computation code path. As this code path would often use SIMD, the cross-over selectivity threshold could even be shifted forward or backwards depending on the data size, exploiting the observation in Figure 8 that the benefits of SIMD are proportionally higher with smaller data types. Finally, depending on the bloom filter size, we could decide to use Fission or not. We developed such heuristics, tuning them to the characteristics of Machine 1, making that a best-case approach.

Results. Table 11 shows see that Micro Adaptivity improves the geometric mean (power score) by 9%, clearly beating the heuristics approach. Given this, we argue that Micro Adaptivity is more attractive than spending time in inventing, tuning and maintaining heuristics, because Micro Adaptivity is by nature more robust, future-proof and less labor-intensive from the development side. Note that the overall improvement is less than what could be expected from Tables 6-10 because different optimizations do not always add up, and because the overall results presented here are end-to-end and also include the interpretation overhead of the extra instrumentation to keep APHs and the extra performance statistics in `vw-greedy`. A 9% improvement may not seem that large, but we think it is very promising since even the most generic flavor sets such as hand unrolling and compiler variation are currently not fully rolled out to all applicable places in the codebase (they cover only 51% of the cycles spent in primitives; so their reach could almost be doubled still).

Future. We think we only scratched the surface in terms of creating algorithmic variations; specifically because besides the fission optimization in bloom filter, all other hash lookup primitives are not covered, though they dominate TPC-H time. As an example, we are thinking about inserting prefetch instructions into hash lookups. Such prefetch instructions are sensitive to the right prefetch depth and stride; and the optimum can only be obtained by tuning to the hardware [4]. Thus, by encoding multiple prefetching approaches and distances in separate primitive instances, we could exploit Micro Adaptivity to automatically find the best combination for the hardware the system runs on and for the data characteristics. Further, it would be interesting to combine Micro Adaptivity with just-in-time (JIT) query compilation techniques [14]. Often, multiple compilation strategies apply for a particular (sub)-query or expression, and it is very hard to create (reliable) cost models for JIT code on the fly (and in case of vectorized execution, some JIT code may not always be superior to vectorized execution). Micro Adaptivity makes robust choices, yet does not rely on a cost model, making it a promising direction to address this challenge.

Query	No Heuristics (sec)	Heuristics	Micro Adaptive
Q01	29.22	1.02	1.10
Q02	1.58	1.10	1.00
Q03	1.43	1.08	1.13
Q04	1.39	1.02	1.14
Q05	5.05	1.07	1.08
Q06	2.42	1.42	1.62
Q07	7.38	1.06	1.06
Q08	7.07	1.09	1.08
Q09	48.72	1.11	1.09
Q10	8.18	1.12	1.07
Q11	2.28	1.10	1.07
Q12	5.67	0.97	1.05
Q13	41.9	1.08	1.03
Q14	3.55	1.18	1.20
Q15	1.43	0.90	1.12
Q16	9.03	0.88	1.00
Q17	9.47	1.00	0.99
Q18	20.21	0.97	1.02
Q19	18.52	0.99	1.01
Q20	5.74	1.03	1.03
Q21	29.9	1.02	1.08
Q22	8.74	1.05	1.09
Geo Avg		**1.05**	**1.09**

Table 11: VW without heuristics (base result) vs. VW with heuristics and Micro Adaptivity (improvement factors).

5. RELATED WORK

Scientific Computing. FFTW [7] is a library that computes the Discrete Fourier Transform and tunes itself to the hardware that it is running. 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, which 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. These codelets are similar to Vectorwise primitives; but FFTW is not adaptive during computation.

The ATLAS project [1] uses an approach similar to FFTW for linear algebra operations. ATLAS has an install phase where it first probes the hardware, and 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). [5] presents a system for computing stencil operations that is able to generate different implementations targeting various architecture specific features (e.g. NUMA, DMA).

In [9], an adaptive sorting library was presented, highlighting unlike the previous examples that input data distributions affects on performance. Therefore, based on some statistics derived from the data, this library chooses a sorting algorithm. This is similar to Micro Adaptivity, but we note that a DBMS cannot assume to have prior statistics, because query plans and data values vary between queries.

Database Research. 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. Adaptive query processing (AQP) attempts to overcome the difficulties encountered by DBMS-es that use the traditional optimize-then-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). Additionally, when executing long running queries, the context (data characteristics, system state) may change, so a static approach leads to poor performance. We discuss a few well-known approaches, but point the interested reader to the excellent survey in [6].

The Eddy operator [10] 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 tuples back to the Eddy, which routes them between operators, based on the observed tuple arrival rates.

The MJoin operator dynamically changes the join order in a multi-way join plan during query execution, exploiting (changing) observed differences in join hit rates.

These adaptive query processing methods are complemented by Micro Adaptivity, as they work on different levels that are largely independent.

Compiler technology. Some compilers (e.g. icc) allow creation of a single binary containing multiple versions of the same function. On runtime, the version matching a given platform is chosen. Micro Adaptivity improves on such static and error-prone decisions, testing multiple versions dynamically and using the one performing best.

6. CONCLUSIONS AND FUTURE WORK

We presented Micro Adaptivity, a technique that can be employed inside query engines that use some form of block-oriented processing to increase query *performance robustness*. The robustness comes from the fact that the system no longer needs to depend on cost models modeling micro features of database operators and their interaction with hardware, or magical thresholds in heuristics. Such approaches tend to be unreliable and non-future-proof as new hardware with new characteristics comes along. Additionally, maintaining cost models and heuristics is also labor-intensive for the development team of a database product. The ability of Micro Adaptivity to mitigate these problems constitutes its “soft” selling-point and comes in addition to its “hard” selling point of performance gains over e.g. heuristics.

We proposed and investigated various ways to create multiple *flavor sets*: different equivalent implementations for the same kind of task. We showed that these different flavor sets cause many performance variations that are hard to capture in a robust cost model. Given multiple flavors, the query evaluator is faced with a real-time optimization task known as the Multi-Armed Bandit (MAB) problem. Here we contributed a new MAB algorithm called *vw-greedy* that specifically targets non-stationary cost distributions (i.e. cost distributions that change during the query). This algorithm was then evaluated on the TPC-workload, using simulation on traces to tune parameters. We investigated the impact of each individual flavor set that we proposed, as well as the overall effect on the TPC-H workload. Here, we also compared with the alternative approach of hard-coded heuristics. Our conclusions are that Micro Adaptivity can significantly reduce the computational cost of primitives with multiple flavors, and moderately but consistently improves overall performance; more than hand-tuned heuristics do.

In future work, we plan to roll out Micro Adaptivity in additional parts of Vectorwise, including performance-critical components like hash-table processing, compression, and even DML handling code. We are also interested in adding more code optimizations e.g. hardware prefetching and just-in-time query compilation techniques. Further, the vw-greedy algorithm can still be improved in various directions, including exploring better performing flavors or flavors known to be dynamic more often. Similarly, for initial primitive choices as well as for the exploration phases one could exploit the history of these flavors in other queries.

7. REFERENCES

[1] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, May 2002.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *TODS*, 32(3), Aug. 2007.

[5] Datta, et. al. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *ACM/IEEE Conf. on Supercomputing*, 2008.

[6] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, Jan. 2007.

[7] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Conf. on Acoustics, Speech and Signal Processing*, 1998.

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

[9] X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library, 2004.

[10] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[11] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

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

[13] K. A. Ross. Selection conditions in main memory. *TODS*, 29(1):132–161, Mar. 2004.

[14] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DAMON*, 2011.

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

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

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

[18] M. Zukowski and P. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.