# Database Scan Variants on Modern CPUs: A Performance Study

David Broneske[1](✉), Sebastian Breß[1,2], and Gunter Saake[1]

[1] University of Magdeburg, Magdeburg, Germany
{david.broneske,gunter.saake}@ovgu.de,
sebastian.bress@cs.tu-dortmund.de
[2] TU Dortmund University, Dortmund, Germany

**Abstract.** Main-memory databases rely on highly tuned database operations to achieve peak performance. Recently, it has been shown that different code optimizations for database operations favor different processors. However, it is still not clear how the combination of code optimizations (e.g., loop unrolling and vectorization) will affect the performance of database algorithms on different processors.

In this paper, we extend prior studies by an in-depth performance analysis of different variants of the scan operator. We find that the performance of the scan operator for different processors gets even harder to predict when multiple code optimizations are combined. Since the scan is the most simple database operator, we expect the same effects for more complex operators such as joins. Based on these results, we identify practical problems for a query processor and discuss how we can counter these challenges in future work.

## 1 Introduction

Operators in a main-memory database are heavily tuned to meet performance needs of tomorrow. In the past, tuning operators for the underlying hardware has attracted much attention (e.g., implementing different join strategies [1–3]). Due to ever-increasing capabilities of modern CPUs (e.g., an increasing number of cores, size of caches, and width of vector registers), the behavior of database algorithms is hard to predict on a given machine [2].

Code optimizations, such as *loop unrolling* or *vectorization*, have different impacts on the performance depending on the given workload (e.g., selectivity) and processor [17]. Furthermore, considering the combination of different code optimizations, algorithm performance will get even more unpredictable, because of interactions between optimizations. In this paper, we perform a first experimental study on the performance impact of combined optimizations. We restrict our study to scans, because it is a very simple operator, where it is feasible to implement a high number of variants.

In our in-depth performance analysis, we analyze the impact of four common code optimizations – loop unrolling, branch-free code, vectorization, and parallelization – and *all* of their combinations. Thus, we contribute in this paper:

A. Jagatheesan et al. (Eds.): IMDM 2013/2014, LNCS 8921, pp. 97–111, 2015.
DOI: 10.1007/978-3-319-13960-9_8

1. A performance comparison of scan variants on different processors for varying workloads (e.g., selectivity and data volume)
2. A description of the relation between hardware characteristics and code optimizations for the scan operator

As a result, we discover that the optimal variant of the scan operator for a given workload is very likely to change across different processors.

Most importantly, there is no simple dependency between the properties of the hardware and the optimal scan operator, because a combined set of optimizations *interact* with each other. The variability in workloads, machines, and sets of code optimizations leads to a large optimization space for database systems and is an unused optimization potential that has not yet been considered to its whole extent. As a consequence, we argue that query execution engines should exploit these unused potentials.

The remainder of this paper is structured as follows. In the next section, we introduce four common code optimizations and present how we applied the optimizations on a simple scan operator in Sect. 3. We evaluate our scan variants on different machines and state important findings in Sect. 4. In Sect. 5, we discuss the impact of our results. We present related work in Sect. 6 and conclude our work in Sect. 7.

## 2 Code Optimizations

In this section, we discuss basics of the four common code optimizations that we apply on the scan operator, namely branch-free code, loop unrolling, vectorization, and parallelization. These code optimizations improve either pipeline or data parallelism to exploit different capabilities of modern CPUs [6]. Of course, there are numerous more code optimizations, such as loop fission, or full computation [7,17], but we limit them to a practically applicable subset in this work.

### 2.1 Branching vs. No-Branching

The usual way to include conditions in a program is to use if-statements. However, when the processor is filling its instruction pipeline, it has to decide whether to include an instruction which depends on the branch or to omit it. For this, CPUs use branch prediction to estimate the result of the branch condition. However, if the outcome of a branch is constantly changing (e.g., in a selection with 50 % selectivity), branch prediction often fails and the pipeline has to be flushed and refilled, which reduces instruction throughput.

As a consequence of the pitfalls of branching, a possible optimization is to write the code in a way that it does not contain any branches. A possible example is to use predication for selections [16]. Although omitting branches avoids branch mispredictions – and, thus, pipeline flushes – we need to execute more instructions than necessary. Thus, it may only be helpful for if-statements whose outcome is hard to predict.

### 2.2 Loop Unrolling

Loop unrolling is a well-known technique to reduce pipeline stalls in tight for-loops [9]. If a for-loop consists of a small amount of instructions (e.g., initializing an array: $array[i] = i$) the overhead of the instructions of the loop may deteriorate its whole performance. Thus, instead of having just one initialization for loop counter $i$ inside the loop body, we could replicate the body to also initialize the array entries of $i + 1$, $i + 2$, and $i + 3$. With this, we reduce stalls in the pipeline of the processor [9], but increase the code size, which may lead to a higher miss-rate in the instruction cache. Notably, modern compilers feature automatic unrolling of loops. Nevertheless, an adaptive unrolling which depends on the number of iterations in the loops cannot be achieved, because the number of iterations is often unknown at compile-time.

### 2.3 Vectorization

The ability to execute a single instruction on multiple data items (called *SIMD*) is an important property of modern CPUs to improve data parallelism. Their benefit has already been shown for applications such as database operations [22] and compression techniques in combination with database scans [19,20]. These SIMD registers offer small load and store latencies [22] and execute one instruction on several data items, for instance, four 32-bit integer values. Since compilers are sometimes not able to vectorize instructions themselves [22], special compiler intrinsics (e.g., *SSE* instructions) are used to explicitly exploit SIMD functionality.

### 2.4 Parallelization

Modern CPUs can execute several threads in parallel. Thus, exploiting thread parallelism in a database is of high importance for improving its performance [12]. Parallelizing database operations implies that data can be partitioned over several threads which work in parallel to achieve lower response times. However, the results of each thread have to be combined to form the end result making parallelization less beneficial for big result sizes. Furthermore, for small jobs, the overhead of coordinating the threads may consume the benefit of parallelization [18].

## 3 Variants for Database Scans

For the implementation of the database scan variants, we chose the database management system CoGaDB (*Column-oriented GPU-accelerated DBMS* [5]) which already offers the basic variants of the scan operator. Hence, we only had to extend this operator set by the combination of optimizations. For simplicity, we present an excerpt of supported types and predicates of a database scan, which we limit here to predicates of the form $x < c$, where $c$ is a constant. Our implemented scan extracts a position list with the tuple identifiers of matching tuples. Of course, this is only one variant of a scan and other approaches of a scan such as extracting a bitmap from the input are worth to evaluate in future work.

## 3.1 Implementation of Single Optimizations

The simple serial implementation of the scan is straightforward; we sketch the code in Listing 1.1. The main component of the serial scan is the for-loop which iterates over the input `array` of size *array_size*.

```
for(int i = 0; i < array_size; ++i) {
  SELECTION_BODY(array,comp_val,i,result,pos,<);
}
```

**Listing 1.1.** Serial scan for comparator less than.

Inside the for-loop, we use a macro (cf. Listing 1.2) to be able to switch the code between branching and branch-free code during compile time. Both macros evaluate whether the array value is smaller than the comparison value `comp_val`, and if true, it writes the position `pos` into the array `result`.

Using these macros allows to either have a branch in the code that conditionally inserts the positions into the positionlist, or else to have a branch-free version of the conditional insertion. In fact, the branch-free version has a stable number of executed instructions and, thus, no branch mispredictions can happen, which increases instruction throughput. Nevertheless, if the comparison is often evaluated as false, we incur an overhead compared to the code with branches.

```
#define SELECTION_BODY_BRANCH(array,value,i,result,pos,COMPARATOR) if(array[i]
     COMPARATOR value){result[pos++]=i;}
#define SELECTION_BODY_NOBRANCH(array,value,i,result,pos,COMPARATOR) result[pos]=i;
      pos+=(array[i] COMPARATOR value);
```

**Listing 1.2.** Macros for branching or branch-free code.

Apart from code with or without branching, another possible variant can be generated by unrolling the loop. In Listing 1.3, we sketch the schema for unrolling the macro inside the loop. The exact code depends on the number of unrolled loop bodies $k$ and has to be implemented for every $k$ that has to be supported in the scan. Notably, each variant of the unrolled scan is also available with branch-free code, since we can use the same macro as in the simple serial scan.

```
  for(int i = 0; i < array_size; i+=k) {
    SELECTION_BODY(array,comp_val,i,result,pos,<);
    ...
    SELECTION_BODY(array,comp_val,i+(k-1),result,pos,<);
  }
... //process remaining tuples in a normal loop
```

**Listing 1.3.** $k$-times loop-unrolled serial scan.

Apart from reducing pipeline stalls by using loop unrolling, our next serial variant uses SSE intrinsics to implement vectorization. Our algorithm in Listing 1.4 is based on the SIMD scan by Zhou and Ross [22]. Since SIMD operations work on 16-byte aligned memory, we first have to process tuples that are not aligned. For this, we use the serial variant, since only a few tuples have to be processed. The same procedure is executed for the remaining tuples that do not completely fill one SIMD register. The presented code snippet evaluates the elements of an SIMD array and retrieves a bit mask for each comparison (cf. Line 4). After that, the mask is evaluated for the four data items and if there is a match, the corresponding position is inserted into the position list (cf. Line 6–10). Notably, similar to the algorithm by Zhou and Ross, we also use an if statement for evaluating whether there has been a match at all, which could reduce executed instructions if the selectivity is high.

```
 1 ... // Code for unaligned tuples
 2 for(int i=0;i < simd_array_size;++i)
 3 {
 4  mask=SIMD_COMPARISON(SIMD_array[i],
        comp_val);
 5  if(mask){
 6   for (int j=0;j < SIMD_Length;++j)
 7   {
 8    if((mask >> j) & 1)
 9     result_array[pos++]=j+offsets;
10   }
11  }
12 }
13 ... // Code for remaining tuples
```

**Listing 1.4.** Vectorized serial scan.

```
 1 //build local result in parallel
 2 for(int i=0;i < num_of_threads;++i) {
 3  do parallel: serial_selection(...);
 4 }
 5 //build prefix sum
 6 prefix_sum[0]=0;
 7 for(int i=0;i < num_of_threads;++i) {
 8  prefix_sum[i]=prefix_sum[i−1]+
        result_sizes[i−1];
 9 }
10 //merge local results in parallel
11 for(int i=0;i < num_of_threads;++i) {
12  do parallel: write_thread_result(
        prefix_sum[i],...);
13 }
```

**Listing 1.5.** Simple parallel scan.

The parallel version of the scan forwards the data array to a number of threads (cf. Listing 1.5, Line 2–4) that build up a local result for the selection on their chunks of the integer array. To allow parallel writing of the local results into a global result without locking, we have to compute the prefix sum (cf. Line 6–9). With this, each thread knows where to copy its local results in the final result array, which is done in parallel (cf. Line 11–13).

## 3.2 Possible Scan Variants

By combining our four code optimizations, we are able to build a total of 16 variants. The implementation concept of most of the combined variants is straightforward. For instance, adding parallelization to all variants is implemented by changing the work that a single thread is doing. E.g., when combining parallelization and SIMD acceleration, each thread is executing its selection using the SIMD algorithm in Listing 1.4 with some adaptions. Furthermore, implementing branch-free code implies to change the used macro. More challenging is the combination of SIMD and loop unrolling. Here, we took the `for`-loop (cf. Listing 1.4), put it into another macro and unrolled it for several iterations. To allow reproducibility of our results, we provide our variants as open source implementation.[1]

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/supplemental.php.

**Table 1.** Used evaluation machines.

| | Machine 1 | Machine 2 | Machine 3 | Machine 4 |
|---|---|---|---|---|
| CPU | Intel Core 2 Quad Q9550 | Intel Core i5-2500 | 2*Intel Xeon E5-2609 v2 | 2*Intel Xeon E5-2690 |
| Architecture | Yorkfield | Sandy Bridge | Ivy Bridge - EP | Sandy Bridge - EP |
| #Sockets | 1 | 1 | 2 | 2 |
| #Cores per Socket | 4 | 4 | 4 | 8 |
| #Threads per Core | 1 | 1 | 1 | 2 |
| CPU Frequency | 2.83 GHz | 3.3 GHz | 2.5 GHz | 2.9 GHz |
| L1-Cache per Core | 128 Kb | 256 Kb | 256 Kb | 512 Kb |
| L2-Cache per CPU | 12 Mb | 4*256 Kb | 4*256 Kb | 8*256 Kb |
| L3-Cache per CPU | — | 6 Mb | 10 Mb | 20 Mb |

## 4 Performance Comparison

For our performance evaluation, we took four different CPUs to test the hardware's impact on the performance of the scan variants. Each machine runs *Ubuntu 10.04.3 LTS 64-bit* as operating system. We compiled our scan variants with the GNU C++ compiler 4.6.4 with the same flags as used by Răducanu et al. [17]. Our workload consists of in-memory columns with integer values internally stored as 32-bit integer arrays containing between 6 million and 60 million values which is about the cardinality of a column of the Lineorder table in the Star Schema Benchmark of scale factors 1–10. Generated values are equally distributed over the range $[0, 999]$. Another parameter is the selectivity factor which we vary in steps of 10 % between 0 % and 100 % to evaluate its impact. The number of used threads for parallelized scans is equal to the number of available threads on each machine. To reach stable results, we repeated each experiment 100 times and applied a gamma-trimming which omits the slowest and fastest 10 results.

**CPU Differences.** To provide an overview of the characteristics of the CPUs of used machines, we summarize necessary information in Table 1. For our evaluation, we choose two commodity CPUs and two server CPUs. While machine 1 has only the L2 cache as last level cache and a little bit lower clock speed, machine 2 has three cache levels and the highest clock speed. Machine 3 offers four cores on each of the two sockets, but has the lowest clock frequency per CPU. The server CPU in machine 4 with an octa core on each of the two sockets allows to process 32 threads with enabled Hyper-Threading. Thus, machine 4 should have the best parallelization potential. Furthermore, our chosen CPUs have different architectures, where the newest architecture is built in on machine 3, being the Ivy Bridge.

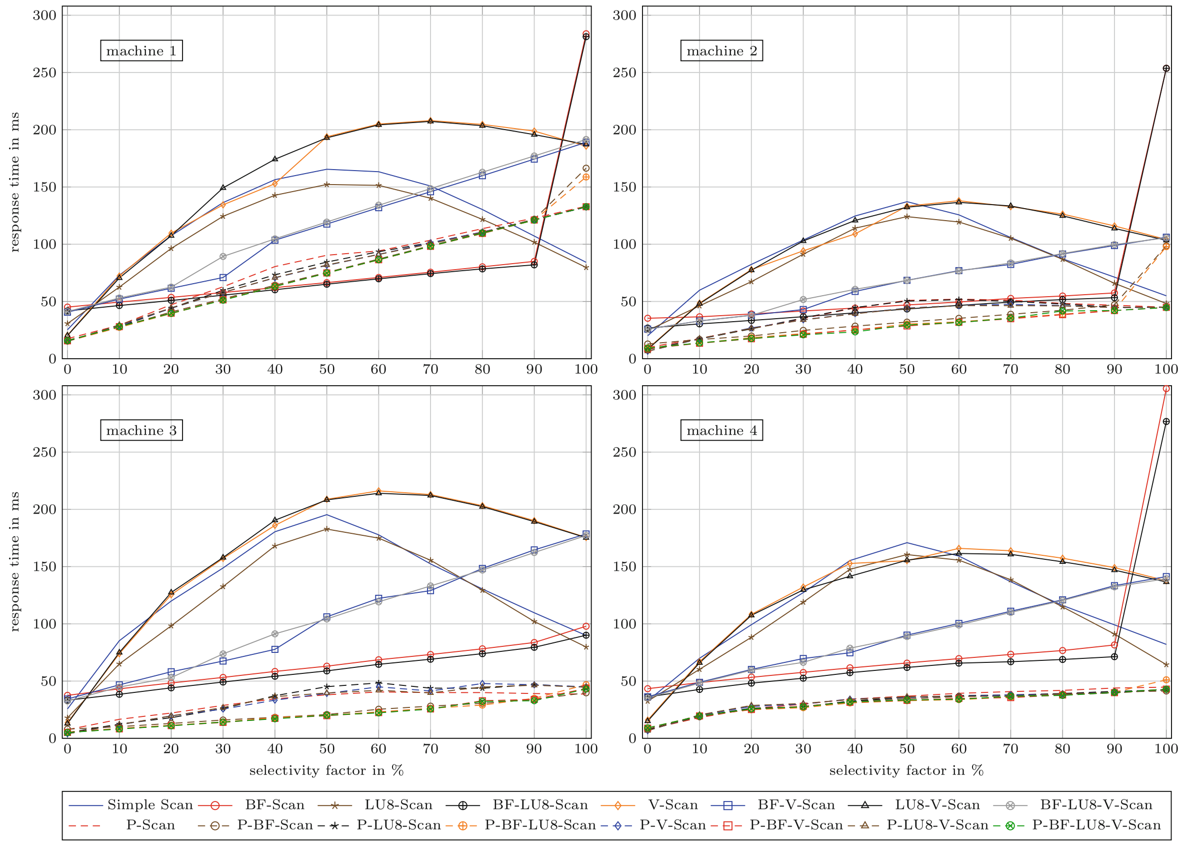

**Fig. 1.** Response time for 30 million data items (BF = branch-free, LU8 = 8-times loop-unrolled, P = parallelized, V= vectorized).

## 4.1 Varying Selectivity

In our first experiment, we focus on the performance of our variants for filtering 30 million tuples with different selectivities. For our variants, we implemented a loop unrolling of depth 8 similar to Răducanu et al. [17] and set the number of used threads to the number of available threads on the machine. To produce increasing selectivity factors over our equally distributed values, we evaluate the predicate $x < c$ with increasing comparison constant $c$. The response times of our 16 algorithms on the four machines are shown in Fig. 1.

From the performance diagrams, we can see that at a selectivity factor smaller than 20 %, serial and parallel selections have similar execution times. It is also visible that serial algorithms can outperform parallel algorithms at a selectivity factor of 100 %. This performance difference is a result of the overhead produced by the result combination of parallel algorithms which worsens for increasing result sizes.

Furthermore, branching code gets high penalties for medium selectivity factors, making branch-free algorithms superior to them. Nevertheless, the performance of branch-free code is steadily getting worse with increasing selectivity factor till the branching counterpart becomes superior again at a selectivity factor of 100 %. Considering unrolling, there are only slight differences between normal loops and unrolled loops. Additionally, the use of SIMD instructions for serial algorithms does not improve the performance as expected. Especially for selectivity factors higher than 50 %, the performance of the vectorized scan is almost the worst. This is probably incurred by the expensive mask evaluation which worsens when the selectivity factor increases. However, if we apply loop unrolling and omit branches, we improve the performance significantly, but still, it is not superior to the serial branch-free version.

In summary, a variant that is performing best under all circumstances cannot be found. Although the parallel branch-free loop-unrolled vectorized scan is the best one for machine 3 and 4, it is not for machine 1 at a selectivity factor more than 50 %. Here, the serial branch-free scan performs best.

**Differences Between Machines.** In contrast to the other machines, machine 1 shows that for selectivity factors above 50 % the serial branch-free and the serial unrolled branch-free selection execute up to 32 % faster than parallel algorithms. Additionally, at a selectivity factor of 100 %, even the branching selection and unrolled selection outperform the best parallel algorithm by 39 % while the performance of the two branch-free versions deteriorate.

The deterioration of the branch-free serial version for a selectivity factor of 100 % is only visible for machine 1, 2, 4. In contrast, machine 3 is not affected, although at this point, the branch-free serial versions are beat by the branching versions. This effect is probably due to the new *next-page prefetcher (NPP)* in the Ivy Bridge architecture in this machine [10]. The NPP prefetches the next cache line if in a sequential access the end of the current cache line is almost reached.

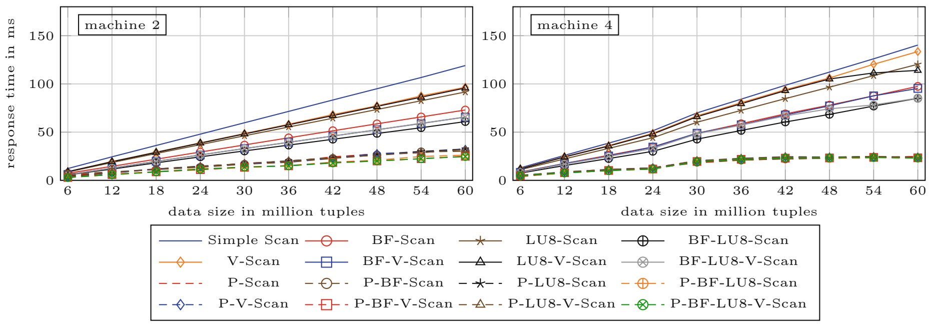

**Fig. 2.** Response time for different amount of data items for selectivity factor 10 % (BF = branch-free, LU8 = 8-times loop-unrolled, P = parallelized, V= vectorized).

Additionally, while the performance of the branching parallel versions is mostly visibly worse than for the branch-free counterparts (cf. machine 2 & 3), these differences disappear for machine 4. Furthermore, the best performance for serial selections is achieved on machine 2 and for parallel algorithms with machine 4. In addition, on machine 3 & 4 all parallel algorithms perform constantly better than the serial ones.

### 4.2 Varying Data Size

We analyzed the impact of different data sizes from 6 to 60 million rows for selectivity factors from 0 % to 100 %. Regardless of the selectivity factor, the optimal algorithm does not change with an increasing amount of data. Therefore, we exemplary show our result for selectivity factor 10 % in Fig. 2 for machine 2 and 4.

All variants show increasing response times for increasing data sizes. Furthermore, with increasing data sizes, the performance advantage of parallel algorithms increases compared to serial algorithms. From this, we can conclude that the main impact factor for the optimality of scan-algorithm variants is the selectivity factor; data size has only a minor impact.

**Differences Between Machines.** Comparing the results from machine 2 with those for machine 4, a big gap between the serial and parallel algorithms is visible on machine 4 that is more severe than on the other machines. The reason for that is that machine 4 has the highest amount of cores and available threads. Thus, machine 4 has the best parallelization capability.

### 4.3 Different Unrolling Depths

In the overall performance evaluation, we decided to use a common unrolling depth of 8 for the loops [9,17]. However, the number of unrolled executions can be varied, which opens another tuning dimension. In this section, we repeated the evaluation of the serial scan variant and compared it to 2–8 times unrolled serial scans.

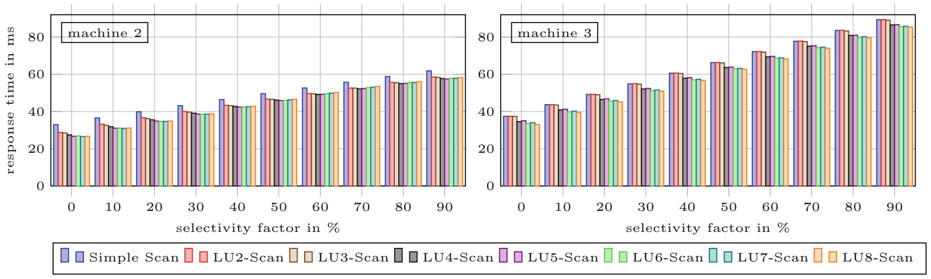

**Fig. 3.** Response time of branch-free scans with different unrolling depths for varying selectivities for 30 million data items (LUn = n-times loop-unrolled).

**Branch-Free Unrolled Scans.** The benefit of unrolling depends on the number of executed instructions inside a loop. Thus, we first evaluated the branch-free version of the serial scan for different unrolling depths, because the number of instructions inside the loop does not depend on branching. With this, we assure that we will find the best unrolling depth for a specific machine independent from the selectivity.

In Fig. 3, we visualize the response times for our serial branch-free scans with different unrolling depths on 30 million data items with selectivity factors between 0 % and 90 % for machine 2 and 3. Here, we skipped the selectivity factor 100 %, since the response time behaves the same as for lower selectivity factors, but its overall value is often double as much. Thus, it would deteriorate values in the diagram.

From the performance diagram in Fig. 3, it can be seen that for each machine, there is an optimal unrolling depth. On machine 2, there is in general a huge difference between the serial scan and the unrolled variants. Here, the generally best unrolling depth is five. In contrast, machine 3 benefits from larger unrolling, having its optimum at 8 times unrolling for the considered depths. This circumstance is probably caused by the new Ivy Bridge architecture, because it offers the possibility to combine the micro-op queue of two cores for a single-threaded task in order to process bigger loops more efficiently [10].

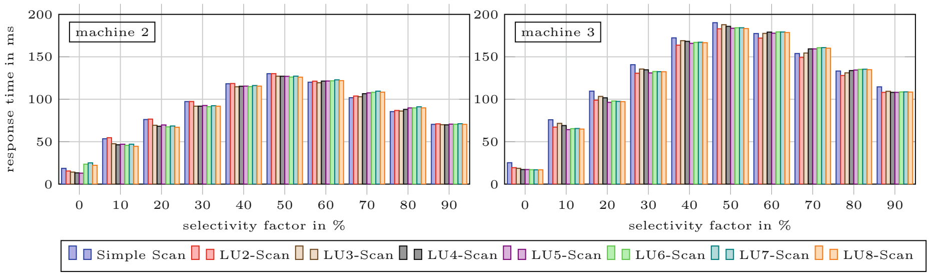

**Fig. 4.** Response time of branching scans with different unrolling depths for varying selectivities for 30 million data items (LUn = n-times loop-unrolled).

**Branching Unrolled Scans.** When including branches in our code, the number of executed instructions inside a loop varies depending on the selectivity. Thus, changing behaviors for machine 2 and 3 can be seen in the performance diagrams in Fig. 4. For instance, on machine 2, for selectivity factors from 0 % to 50 % the serial version behaves worse than an unrolling of depth four and for a selectivity factor higher than 50 %, an unrolling of depth three behaves best. Machine 3 shows good performance for 8 times unrolled loops to a selectivity factor of 30 %, where two-times unrolled code gets best till 90 %.

## 5 Discussion

In the last sections, we presented the evaluation of our scan variants on different machines. We have shown that there are convincing performance differences with respect to varying selectivities and different machines. In this section, we discuss our findings and their impact on a broader view.

### 5.1 Variant Performance

Our evaluation revealed that there is no optimal scan variant for different CPUs, and for each CPU, it is not trivial to select the optimal variant. Additionally, the optimal variant may change depending on the scan's selectivity.

**Branch-Free Code.** From the evaluation, we can conclude that performance benefits of branch-free code strongly depends on the selectivity. Nevertheless, we can observe, that branch-free code may degrade performance for the serial or unrolled scan on some machines (cf. Fig. 1; machine 1, 2, 4: selectivity factor 100 %). Instead, for loop unrolling, branch-free code assures that there is an optimal unrolling depth independent of the selectivity.

**Loop Unrolling.** Loop unrolling offers performance improvements, if (1) the unrolling depth is adjusted to the used processor, and (2) the number of executed instructions in the loop is stable. If the executed instructions in the loop is unstable, the perfect unrolling depth has to be chosen during runtime, for instance, by the hybrid query processing engine HyPE [4]. Nevertheless, loop unrolling does not severely worsen the performance and, thus, it is a valuable optimization that should be considered in every application.

**Parallelization.** Our results indicate that, in general, parallelization offers a good opportunity for accelerating the scan if the CPU offers enough cores (e.g., on machine 3 or 4). Nevertheless, when parallelized, the scan employs the whole processing capacity of the CPU. With this, response times are maximized, but throughput may be insufficient. Consequently, it has to be carefully weighed whether a parallel scan should be preferred to a serial scan.

**Vectorization.** Our vectorized scan is most of the times not competitive to other scan variants. However, at low selectivity factors the vectorized scan is the best serial scan, because the probability of excluding several data items in one step is high and beneficial for performance. Its performance loss at higher selectivity factors is caused by the bad result extraction from the bit mask. Hence, instead of expecting a position list as a result, we should rather use a bitmap to represent the result for efficient vectorization.

Concluding, different code optimizations have a varying impact on the performance of a simple scan. Therefore, it is even more challenging to choose an optimal algorithm for more complex operators.

## 5.2 Threats to Validity

To assure internal validity, we cautiously implemented each variant and equally optimized the code of all variants for performance. We used plain C arrays instead of containers and ensured that the compiler does not perform loop unrolling or auto-vectorization. Our evaluation setup assures that array sizes exceed available cache sizes. Thus, higher sizes should not change the behavior of the variants. However, we executed our tests on machine 3 another time with data sizes of 500 million values without any impact on the general variant performance behaviors.

To reach a high external validity, we extensively show our implementation concepts in Sect. 2, our evaluation environment in Sect. 4 and provide the code to allow for reproducing of our results. However, CoGaDB operates in an operator-at-a-time fashion, which means the whole input is consumed by the operator and the result is then pushed to the next operator. Thus, our results apply to systems that follow this processing paradigm and we expect similar results for vectorized execution.

## 5.3 Toward Adaptive Variant Selection

As a consequence of the performance differences depending on the used machine and the workload, we need to solve two challenges. First, code optimizations have hardly predictable impacts between machines, which does not allow us to build a simple cost model for an operator. Consequently, we can choose the optimal variant at run-time only by executing and measuring the performance of variants. Second, the number of possible variants is to high to keep them all available during run-time. In fact, for each additional independent optimization, the number of produced variants increases by factor two. Furthermore, possible points where code optimizations make sense will increase with increasing complexity of the optimized operator.

As a solution, we argue to keep a pool of variants for each operator during run-time (cf. Fig. 5). The system generates new variants using optimizations that are likely to be beneficial on the current machine. Variants that perform poor w.r.t. the other variants are deleted and replaced by new variants. As a

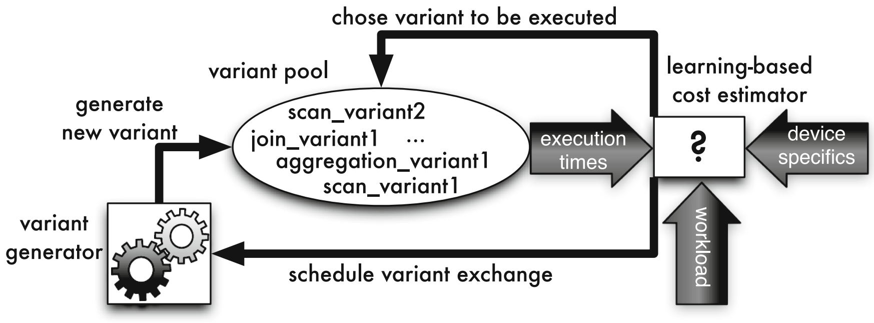

**Fig. 5.** Adaptive query processing engine

consequence, we also have to think of a suitable approach to generate the variants instead of implementing them by hand.

To select the variant to be executed from the pool, we propose to use a learning-based query execution engine, such as HyPE [4], which learns cost-models for each variant depending on the given machine and workload. With this, we achieve optimized performance due to the usage of best-performing variants from the variant pool for the used machine and also for the current workload. The variant pool itself has to be limited, because learning cost models for many variants introduces too much overhead. Thus, we propose to let the query execution engine decide which algorithm has to be deleted and which one has to be generated, in case it is beneficial for the current or future workload. With this, we achieve a run-time adaptability of our system with respect to the workload and used machine.

**Limitations.** Our proposed approach relies on good selectivity estimations to choose the optimal variant of the scan and query plan for the given workload. However, we argue that approaches such as kernel-density estimation by Heimel and Markl [8], or work of Markl et al. [14,15] should make it possible to overcome these challenges.

## 6 Related Work

Răducanu et al. tested different variants of database operations in Vectorwise [17]. Because of the vectorized execution model in Vectorwise, they are able to execute different variants of one database operation during the processing of one column, arguing that different code optimizations are favored by different machines and workloads. Nevertheless, their findings do not reveal the impact of the *combination of code optimizations*, which we expose for the scan operator. In fact, they did not consider different unrolling depths as we do. Furthermore, although we come to the same conclusion as they do, we want to tackle the problem by learning cost models instead of only the execution time of a variant, because we find it more appropriate for our use case.

Related work in the area of code optimizations for stencil computation can be found in the work of Datta et al. [7] and improving scans is topic of the work of Li and Patel [13]. Furthermore, there is much work on applying vectorization on database operations, such as SIMD-accelerated scans for compressed data by Willhalm et al. [19,20], using SIMD instructions for database operations by Zhou and Ross [22], and also using SIMD for accelerating index search by Kim et al. [11] or Zeuch et al. [21]. Their ideas help implementing vectorized database operations, but they compare their implementations only to the serial variant and do not include other code optimizations or machines as we do.

## 7 Conclusion and Future Work

With the growing heterogeneity of modern processors, it becomes increasingly difficult to exploit their capabilities. Thus, we need an understanding on which hardware characteristics favor which set of code optimizations to achieve the best performance of database operators. Due to interactions between optimizations, this is a non trivial problem.

In this work, we investigated the impact of four different code optimizations and their combinations on the scan operator. We evaluated the performance of the resulting 16 database scan variants on different machines for different workloads. Our results indicate that the performance of most of the algorithms is depending on the selectivity of the scan and also on the used machine. However, when combining code optimizations (e.g. branch-free code and varying loop unrolling depths), simply changing the used machine favors a different algorithm variant. As a consequence, we have to include these variants in the optimization space of our query engine. However, because there are numerous code optimizations and because of their exponential amount of combinations, we run into several problems: building a cost model including each variant is hardly possible, and providing executable code for each variant during run-time is not feasible because of the large number of variants and their respected memory consumption.

Thus, future work includes to learn execution behaviors of the variants by a suitable query engine (e.g., HyPE) that choses the best-performing variant from an algorithm pool and schedules a rejuvenation of the pool which exchanges variants that perform badly for the current workload.

**Acknowledgments.** We thank Jens Teubner from TU Dortmund and Max Heimel from TU Berlin for helpful feedback and discussions.

## References

1. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB **5**(10), 1064–1075 (2012)
2. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. PVLDB **7**(1), 85–96 (2013)

3. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: ICDE, pp. 362–373 (2013)
4. Breß, S., Beier, F., Rauhe, H., Sattler, K.U., Schallehn, E., Saake, G.: Efficient co-processor utilization in database query processing. Inf. Sys. **38**(8), 1084–1096 (2013)
5. Breß, S., Siegmund, N., Heimel, M., Saecker, M., Lauer, T., Bellatreche, L., Saake, G.: Load-aware inter-co-processor parallelism in database query processing. Data Knowl. Eng. (2014). doi:10.1016/j.datak.2014.07.003
6. Broneske, D., Breß, S., Heimel, M., Saake, G.: Toward hardware-sensitive database operations. In: EDBT, pp. 229–234 (2014)
7. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC, pp. 1–12 (2008)
8. Heimel, M., Markl, V.: A first step towards GPU-assisted query optimization. In: ADMS, pp. 33–44 (2012)
9. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann Publishers Inc, San Francisco (2007)
10. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (April 2012). http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf
11. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In: SIGMOD, pp. 339–350 (2010)
12. Leis, V., Boncz, P., Kemper, A., Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD, pp. 743–754 (2014)
13. Li, Y., Patel, J.M.: BitWeaving: fast scans for main memory data processing. In: SIGMOD, pp. 289–300 (2013)
14. Markl, V., Lohman, G.M., Raman, V.: LEO: an autonomic query optimizer for DB2. IBM Syst. J. **42**(1), 98–106 (2003)
15. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: SIGMOD, pp. 659–670 (2004)
16. Ross, K.A.: Selection conditions in main-memory. TODS **29**, 132–161 (2004)
17. Răducanu, B., Boncz, P., Zukowski, M.: Micro adaptivity in vectorwise. In: SIGMOD, pp. 1231–1242 (2013)
18. Teubner, J., Mueller, R., Alonso, G.: Frequent item computation on a chip. TDKE **23**(8), 1169–1181 (2011)
19. Willhalm, T., Boshmaf, Y., Plattner, H., Popovici, N., Zeier, A., Schaffner, J.: SIMD-Scan: ultra fast in-memory table scan using on-chip vector processing units. PVLDB **2**(1), 385–394 (2009)
20. Willhalm, T., Oukid, I., Müller, I., Faerber, F.: Vectorizing database column scans with complex predicates. In: ADMS, pp. 1–12 (2013)
21. Zeuch, S., Freytag, J.C., Huber, F.: Adapting tree structures for processing with SIMD instructions. In: EDBT, pp. 97–108 (2014)
22. Zhou, J., Ross, K.A.: Implementing database operations using SIMD instructions. In: SIGMOD, pp. 145–156 (2002)