Filter Representation in Vectorized Query Execution

Amadou Ngom♣, Prashanth Menon
Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, Andrew Pavlo
♣Massachusetts Institute of Technology, Carnegie Mellon University
{ngom@mit.edu,pmenon@cs.cmu.edu}

Abstract

Advances in memory technology have made it feasible for database management systems (DBMS) to store their working data set in main memory. This trend shifts the bottleneck for query execution from disk accesses to CPU efficiency. One technique to improve CPU efficiency is batch-oriented processing, or vectorization, as it reduces interpretation overhead. For each vector (batch) of tuples, the DBMS must track the set of valid (visible) tuples that survive all previous processing steps. To that end, existing systems employ one of two data structures, or filter representations: selection vectors or bitmaps. In this work, we analyze each approach's strengths and weaknesses and offer recommendations on how to implement vectorized operations. Through a wide range of micro-benchmarks, we determine that the optimal strategy is a function of many factors: the cost of iterating through tuples, the cost of the operation itself, and how amenable it is to SIMD vectorization. Our analysis shows that bitmaps perform better for operations that can be vectorized using SIMD instructions and that selection vectors perform better on all other operations due to cheaper iteration logic.

ACM Reference Format:
Amadou Ngom♣, Prashanth Menon and Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, Andrew Pavlo . 2021. Filter Representation in Vectorized Query Execution. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465998.3466009

1 Introduction

Modern DBMSs utilize the vectorized processing model pioneered by Vectorwise [17] to improve query execution performance. In this model, relational operators implement a uniform interface to iterate over its results in a Volcano-style manner [3]. However, unlike the original Volcano model, in a vectorized engine, relational operators exchange small vectors of typically 1–2k tuples in each invocation of the iterator. This simple enhancement (1) amortizes the iteration overhead across all tuples in the vector and (2) maximizes computation on tuple data while it is in the CPU's cache.

Vectorized relational operators exchange batches of tuple where each tuple attribute is stored separately in a compact vector. For instance, a filter operator applies a predicate on each input tuple and copies its attributes into an output vector if successful. This approach incurs memory overhead due to data copying. A common technique to overcome this is to augment batches with a data structure that *logically* masks out invalid tuples (i.e., a logical filter). We refer to this data structure as a *filter representation.* Two common representations are (1) Selection Vectors (SVs) and (2) Bitmaps (BMs). A SV is a dense sorted list of tuple identifiers (TID) indicating which tuples in the batch are valid during processing. With BMs, each tuple in the batch is assigned a positionally aligned bit; valid tuples have their bit set to 1. The DBMS marks tuples as invalid by modifying the filter representation alone without copying data.

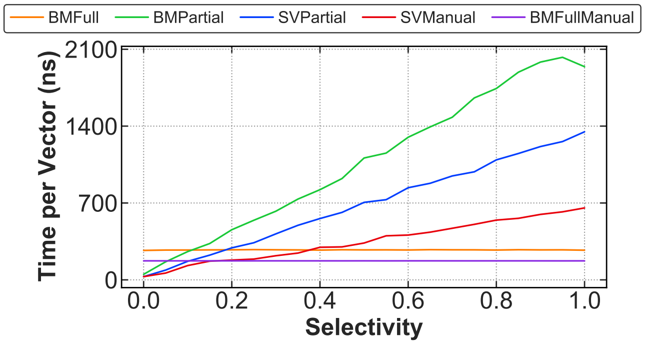


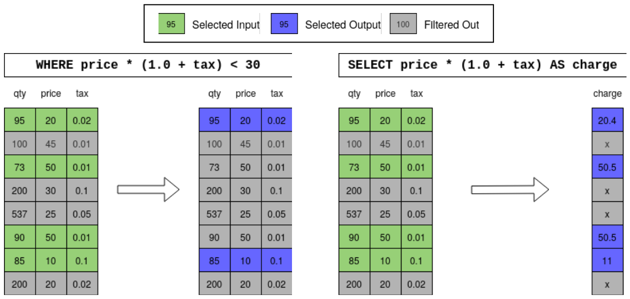
Figure 1: Motivating Example – We evaluate the time to apply a simple predicate filtering an arithmetic column with a constant value.

Interestingly, previous works choose a representation strategy without providing a clear (or empirical) justification. Vectorwise and its derivatives rely selection vectors [6, 14, 15, 17]. IBM DB2's BLU [12] and the more recent VIP [11] rely on bitmaps for the intermediary results of a table scan's filters and selection vectors for other relational operators. In this work, we find that supporting both representations and dynamically choosing between them results in better performance than static implementations. Depending on the specific primitive and the selectivity (i.e., the ratio of selected tuples) of its input vector, selection vectors can outperform bitmaps and vice-versa.

To illustrate the need for a deeper exploration of the impact of a chosen filter representation strategy, we present an experiment that measures the performance of evaluating a **WHERE** during a sequential table scan over a table composed of a single 64-bit integer column. For this experiment, we generate the column's data using a uniform distribution, and vary the input filter's selectivity between 0 and 1. We defer the full description of our experimental setup to Section 3.

We implement and measure five different execution strategies. BMPartial, BMFull, and BMFullManual all use bitmaps. BMPartial applies the operation only on selected tuples, while BMFull applies it on *all* tuples. Likewise, BMFullManual uses a hand-written SIMD kernel to apply the operation to all tuples in each vector. SVPartial

DAMON'21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8556-5/21/06.
https://doi.org/10.1145/3465998.3466009



(a) Update Operation **(b)** Map Operation

Figure 2: Operations on Filtered Vectors – Updates change the set of selected tuple. Maps apply a function to the set of selected tuples without updating the set. Selected input tuples are in green. Selected output tuples are in blue.

and SVManual both use selection vectors, but SVManual uses a custom hand-written SIMD kernel.

The results in Figure 1 show that no one strategy is globally optimal. In selectivity range [0.00,0.15], SVManual and SVPartial perform the best overall, with the latter being preferred due to simpler engineering overhead. Beyond 0.15, a hand-written SIMD kernel using bitmaps for the filter performs the best, up to 3-11× faster than the alternatives. If not relying on custom SIMD code at all, BMFull performs the best beyond a selectivity of 0.35 and is up to 2-7× faster than the alternatives. It also requires the least engineering effort to implement.

The previous results highlight that the choice of filter representation is not clear cut. This work's main contribution is a methodology on how to optimize the performance of arbitrary vectorized primitives by taking into account filter representation, selectivity, and loop optimizations. We also provide recommendations for developers working on vectorized query engines on how to best implement primitives. To evaluate our approach, we implement our framework into the **NoisePage** DBMS [1] and measure its performance on OLAP workloads.

2 Background

Figure 2 shows the two primitives for processing filters: (1) **Update** and (2) **Map**. With Update, the DBMS applies a filter to the vector and *modifies* the set of selected tuples accordingly. The example in Figure 2a shows the DBMS applying the WHERE clause to update the result set during a scan on a table. With Map the DBMS does not modify the input selected set, but computes a new data vector using a mapping function. A projection (i.e., SELECT clause) is an example of a map operation. Although there are other types of primitives, they often have side-effects (e.g., insertion in a hash table for joins or an array for sorting) and are, therefore, not amenable to our optimizations for correctness reasons.

In this section, we first discuss the factors that influence the performance of these primitives and the approaches for the DBMS to execute them. We then provide an analytical construct to explain these implementations' performance.

2.1 Compute Strategies

There are three approaches for the Map and Update primitives: (1) **Selective**, (2) **Full**, and (3) **Mixed** compute. With Selective, the DBMS only applies the given functions on selected input tuples; the non-selected tuples have undefined values in the output. Contrast this with the Full approach, where the DBMS computes updated values in its output for all input tuples regardless of whether they are selected or not. Although this seems wasteful, Full benefits from SIMD vectorization, simple loop structure that enables better loop unrolling, interleaving, and easier branch prediction. To exploit this trade-off, the Mixed strategy switches from Full to Selective when the selectivity (i.e., the ratio of selected tuples) goes below a threshold. We discuss how to derive this threshold in Section 3.

For Update primitives, Full first finds the set of tuples in the vector that pass the predicate, regardless of the input filter. The final filter is the intersection of this intermediary filter and the input filter. BM intersection is an efficient operation because the DBMS can use the AVX512 AND instruction to intersect 512 bits in one cycle [5]. On the other hand, SVs require a slower sorted-set intersection algorithm [4]. Thus, the Full approach is not compatible with SVs using Update primitives.

For Updates with SVs another decision is whether to use branching or branchless evaluation [13]. We do not focus on this as it has been studied extensively before [13–15]. Prior work identified branch misprediction cost as the main factor. Thus, a branching implementation is competitive when the same branch is taken ~90% of the time, meaning that less than 10% or more than 90% of the tuples satisfy the condition. We use the optimal strategy for Update primitives that use SVs in the rest of this paper.

2.2 Primitive Performance

We next analyze the primitives' performance according to their implementations. Our primitive implementations follow one of the patterns shown in Appendix A. These listings show various strategies to multiply a vector by a constant (a Map primitive) along with their optimized versions using SIMD instructions: Full (which does not depend on representation), Selective with SVs, and Selective with BMs. In each instance, the code is a loop divided in two parts: the *iteration logic* , and the *core operation*. The core operation (e.g., multiplication by a constant) is independent of the strategy. On the other hand, the iteration logic is strategy dependent; it determines how to iterate through tuples – whether selected ones in the case of Selective or all tuples in the case of Full– to perform the core operation and store its result.

The following equation computes the expected runtime R of a primitive where N is the number of tuples processed, I is the iteration logic time per tuple, and O is the operation time per tuple:

$$R = N \times (I + O) \tag{1}$$

This formula provides a framework to analyze any vectorized primitive. Let us consider how an implementation strategy influences each of the three factors.

Number of Tuples Processed: Selective processes only selected tuples while Full processes every tuple. Thus, Full performs worse at lower selectivity (i.e., ratio of selected tuples) due to wasted work.

Iteration Logic Time per Tuple: Although Full processes more tuples, it also has the simplest iteration logic. For Selective, iterating over a BM is more expensive than iterating over a SV because

Strategy	Filter	Compute	SIMD	Compatibility
SVPartial	SV	Selective	None	Update, Map
SVManual	SV	Selective	Manual	Update, Map
BMPartial	BM	Selective	None	Update, Map
BMFull	BM	Full	Automatic	Update, Map
BMFullManual	BM	Full	Manual	Update, Map
Full	Either	Full	Automatic	Map
FullManual	Either	Full	Manual	Map

Table 1: Implemented Strategies

of the extra operations required to identify set bits (i.e., valid tuples). Optimizations like loop unrolling and SIMD vectorization can further decrease the iteration logic time per tuple, but they are only available for Full and SVs. The BM’s selective iteration logic is too complex to perform such optimizations.

Core Operation Time per Tuple: The core operation is the same across all strategies. The only way to reduce its contribution to the running time is through data-parallel SIMD instructions, which are only available with Full and Selective with SVs. There are, however, situations where SIMD vectorization provides little gain or even hurts performance as is the case when the core operations contains complex memory accesses and/or branching code. In general, core operations benefit most from SIMD vectorization when they only contain straight-line (i.e., no branches), arithmetic, and bitwise instructions.

3 Optimal Strategies

In this section, we experimentally derive the optimal execution strategy for any given primitive. Table 1 summarizes all the strategies we implemented. As noted before, there is no Full strategy for Updates that works with SVs. Update experiments will thus always use a BM for Full. For Maps, Full neither reads nor updates the filter, so the representation is irrelevant. In addition to relying on compiler auto-vectorization, we also experimented with manual vectorization (e.g., with SVManual) for straight-line arithmetic and bitwise operations (as in Section 3.3).

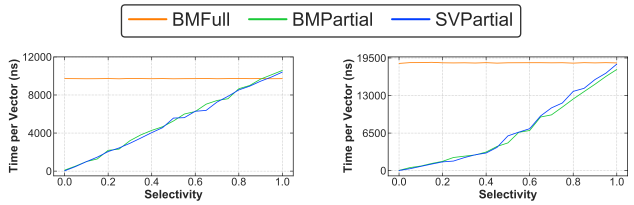
We run the experiments on an Intel Xeon Platinum 8124M CPU @ 3.00GHz with AVX512. We use the in-memory NoisePage DBMS [1] with a read-only column store tables[7], compiled with Clang v9.

In each experiment, we generate a synthetic data set and manually vary the selectivity of each primitive’s input vector from 0.0 to 1.0 in increments of 0.05 to show the impact of selectivity on optimal strategy. Each experiment executes its primitive on enough vectors to obtain a stable average running time. For example, fast primitives are executed in the order of 10^6 times, whereas the slower ones are executed in the order of 10^4 or 10^5 times. Map primitives materialize their results on a vector in-memory, but Update primitives only modify the input filter without materializing a new vector.

3.1 Non Data-Parallel Core Operations

We first consider core operations that are not data-parallel. We evaluate two kinds of operations: (1) those that operate on variable-length data (e.g., string operations) and (2) those that use instructions without a SIMD counterpart (e.g., integer division, modulo).

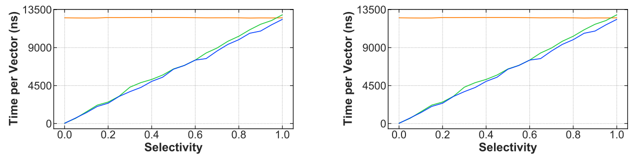
Variable-Length Data: We first analyze the performance of operations on variable-length data, as is often the case for string operations (e.g., string comparison, sub-string). Let us consider the factors in Equation (1). The core operation executes a variable number of instructions depending on the length of the input strings. As we show in the next experiment, it dominates the iteration logic time, even for short strings. Because data parallelism is not available, the core operation time per tuple is the same across all strategies. The principal factor that differentiates strategies is the number of tuples processed, which favors Selective strategies over Full.



(a) Update Operation: `WHERE country < ‘Japan’`

(b) Map Operation: `SELECT POSITION(‘a’ in country)`

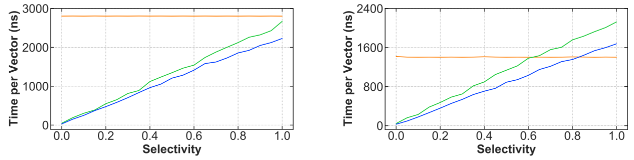
Figure 3: Operations on Variable Length Data



(a) Update Operation: `WHERE col1 % col2 < val`

(b) Map Operation: `SELECT col1 / col2`

Figure 4: Operations with Integer Division



(a) Update Operation: `WHERE col1 < val1 || col2 < val2`

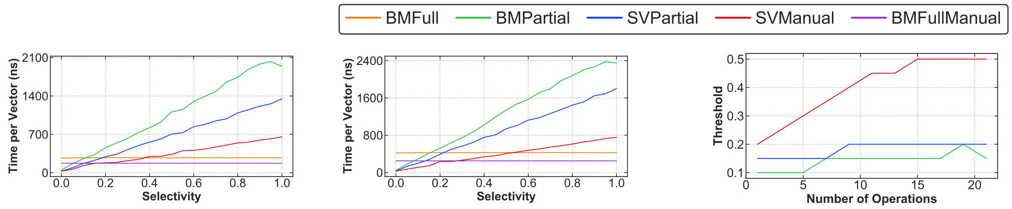
(b) Map Operation :`SELECT col1 < val1 || col2 < val2`

Figure 5: Operations with Branches

To confirm this intuition, we run a micro-benchmark using simple operations on short strings. The Update micro-benchmark performs string comparison, and the Map micro-benchmark performs character location. The strings are all short country names (e.g., USA, Senegal, China). The purpose is to show that the number of tuples processed is the principal performance factor, even for relatively cheap variable-length operations.

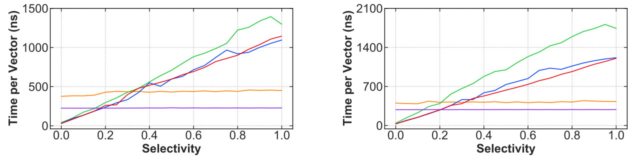
The results are shown in Figure 3. We see that Full consistently performs worst when it processes more tuples. At full selectivity, Full and Selective process the same number of tuples, but the former slightly benefits from a simpler iteration logic. We also see that the performance of SVPartial is similar to that of BMPartial, despite the former’s simpler iteration logic, meaning that the core operation is the dominating cost.

The optimal strategy in this scenario is SVPartial. It processes fewer tuples than Full strategies and has a slightly cheaper iteration logic than BMPartial (though the performance difference is small). Full is only marginally competitive at full selectivity. Its benefits are mostly negligible: the more expansive the core operation, the worse Full becomes.



(a) One operation **(b)** Five operations

Figure 6: SNBS Update with Varying Operations per Iteration



(a) One operation **(b)** Five operations

Figure 7: SNBS Map with a Varying Operations per Iteration

Integer Division: We next analyze the integer division instruction as it does not have a SIMD counterpart. Integer division is so expensive that the iteration logic time is negligible compared to the core operation time. Because data parallelism is not available, its cost per tuple is the same across all strategies. Once again, the main factor that differentiates strategies is the number of tuples processed, which favors Selective.

The results for the micro-benchmarks that evaluate an Update and a Map primitive that both use integer division are shown in Figure 4. Our analysis is similar to Figure 3: the cost of the integer division and the unavailability of data-parallelism make Full inefficient. SVPartial is again the optimal strategy, but only has a slight edge over BMPartial due to the former's simpler iteration logic.

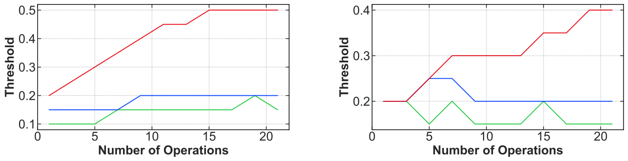
We conclude that the number of tuples processed is the dominant factor for non-data-parallel primitives, making Full impractical. Because of the small impact of iteration logic time, SVPartial has a performance edge over BMPartial, but developers can choose either representation without much affecting performance.

3.2 Inefficient Data Parallelism

To study the effect of inefficient use of SIMD instructions, we consider core operations with branching code. Section 4 will discuss primitives that contain pointer manipulation and memory accesses. Here, SIMD strategies benefit from processing multiple tuples at a time, but suffer from executing more code than SISD strategies. In some cases, core operation time per tuple may even increase depending on the branching structure (e.g., having to execute all branches of a switch statement). Thus, if Full can outperform Selective at all, it will only do so at the highest selectivities because the small reduction in time spent per tuple cannot compensate for the higher number of tuples processed.

To determine the impact of a single branch within the core operation, we implemented primitives that perform logical and/or operations. Due to boolean short-circuiting, these primitives will contain exactly one branch. We confirmed that the compiler correctly auto-vectorizes the Full strategies.

The results are shown in Figure 5. We can see that Full is either always the worst strategy in Figure 5a, or only competitive at high selectivities (>= 0.85) in Figure 5b. The next section shows that branchless primitive are competitive at much lower selectivities (>= 0.2). Although Full is competitive at the highest selectivities, we recommend, for the sake of simplicity, the SVPartial strategy.



(a) Update Operations **(b)** Map Operations

Figure 8: Mixed Compute Thresholds

3.3 Straight-Line & Data-Parallel Operations

We now consider core operations with straight-line and data-parallel (SLDP) code (i.e., non-branching code that leverages SIMD instructions). Primitives that only perform arithmetic (without integer division) and bitwise instructions fall under this category. We use Equation (1) to analyze the performance of each strategy in this scenario. Full strategies (e.g., BMFull) leverage SIMD instructions to reduce the iteration logic and core operation time per tuple. For example, AVX512 integer addition instructions perform eight 64-bit additions in a single cycle [5]. Therefore, we expect Selective strategies to only be competitive when the selectivity is low, meaning that Full processes far more tuples. SVManual is the exception: it is a Selective strategy that uses SIMD instructions. Its gain in iteration logic time is, however, not as high as that of Full strategies because it uses a `gather` instruction [5] to collect the elements at the selected indices (see Figure 10). The higher the core operations time per tuple, the less important the `gather` overhead because iteration logic time becomes more insignificant. Thus, we expect SVManual to be competitive with Full at medium or below selectivities depending on the core operation. The next experiments will quantify the differences between each strategy.

We implemented Update and Map primitives in which the core operation contains single-cycle arithmetic and bitwise instructions. We progressively increase the number of such instructions to increase the core operations time per tuple and quantify the selectivity thresholds above which Full Compute strategies become faster than Selective ones. The results are shown in Figures 6 and 7 (for Updates and Maps). The selectivity thresholds for each operation type are respectively in Figures 8a and 8b.

In all experiments, SVPartial performs better than BMPartial. As explained in the previous section, this is only due to a simpler iteration logic. SVManual is always the best Selective strategy because it uses data-parallelism to reduce the time spent per tuple.

Manual Vectorization sometimes performs better than Auto-Vectorization with Full and BMFull. For example, the results in Figure 7a show that BMFullManual is 1.8× faster than BMFull. Upon investigation of the generated code, we discovered that the compiler is overly conservative when it comes to using AVX512. AVX512 registers result in decreased CPU frequency [10], so the compiler does not always use them. It often uses AVX2 registers instead. We, on the other hand, always use AVX512. As we will see in the next section, the decreased CPU frequency can slow down the query.

The thresholds for SVPartial and BMPartial, shown in Figures 8a and 8b, prove that these strategies are only competitive with Full at low selectivities (<= 0.2) because, as explained above, they do not use SIMD instructions.

The threshold for SVManual, on the other hand, increases with the number of operations because its iteration logic overhead becomes more and more insignificant as the core operation time increases. SVManual thus benefits from data-parallelism while processing fewer tuples than Full strategies. The iteration logic difference is never entirely negligible, though; the threshold is medium at best (0.5 at 15 operations in Figure 8a).

Our analysis shows the importance of Mixed: a DBMS should use Selective below the selectivity thresholds, and Full otherwise. The next section details our implementation of the Mixed strategy.

3.4 Implementing Mixed Compute

The previous section showed that for a given SLDP primitive, there is a threshold below which Selective is optimal and above which Full is optimal. To find this threshold, we can run a microbenchmark similar to the one in Figure 8. The lowest selectivity at which the runtime of Full strategies exceeds that of Selective strategies represents the Mixed threshold for a given primitive. In summary, the optimal SLDP strategy is as follows:

For Maps: Full when the selectivity is above the threshold, and SVManual when the selectivity is below the threshold.

For Updates: BMFull or BMFullManual when the selectivity is above the threshold, and SVManual when the selectivity is below the threshold. Note that the BMFull and SVManual have different filter representations. There is a small cost associated with the conversion from BM to SV, but, in practice, other operations always follow an update (e.g., there is a projection after a scan filter). The subsequent operations often amortize the conversion cost. For small queries, in which the amortization does not neutralize the conversion cost, BMPartial can slightly outperform SVManual because it maintains the BM representation. Nonetheless, as we show in Section 4, its gains are mostly negligible.

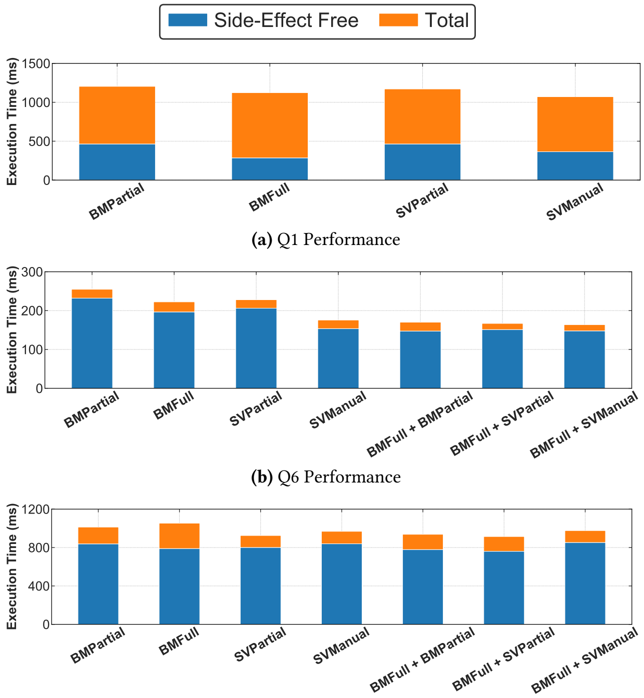
4 Experimental Evaluation

We now evaluate the conclusions of Section 3 in a full DBMS. For this evaluation, we implemented the strategies for a subset of queries from the TPC-H benchmark [16] in the in-memory NoisePage DBMS [1, 8, 9]. Each query contains primitive with side-effects (e.g., hash table insertion), and primitives without side-effects (e.g., multiplication of two columns).

We load a TPC-H database with scale factor 10 (~10 GB) into NoisePage. The system stores data in the Apache Arrow format [7] with dictionary compression for string columns. We run the benchmark workload on the same benchmark machine as in Section 3. If the compiler fails to automatically vectorize a data-parallel primitive or incorrectly uses AVX2, we hand-write an implementation using AVX512 intrinsics [5].

4.1 Q1 – High Selectivity SLDP Primitives

Q1 consists of a set of SLDP operations on vectors with high selectivity (>= 0.95) followed by a side-effect full aggregation that dominates the running time.



(a) Q1 Performance

(b) Q6 Performance

(c) Q4 Performance

Figure 9: TPC-H Performance – Performance breakdown of all applicable filter strategies for TPC-H queries (a) Q1, (b) Q6, and (c) Q4. For each query we plot the total running time and the time to evaluate the side-effect free portion of the query.

Figure 9a shows the performance measurements in NoisePage for each of the strategies. We omit Mixed because no switching occurs during execution. As predicted, BMFull performs the best for the side-effect free primitives. It outperforms SVPartial and SVManual by 1.6× and 1.3×, respectively.

The total running times are similar for all queries for two reasons. First, the (side-effect) aggregation is the query's dominant component in its execution. Second, the usage of AVX512 instructions on our benchmark machines incurs CPU throttling effects. Our version of Intel's Xeon slows down its clock speed when it executes AVX512 instructions [10]. The BMPartial and BMFull strategies have the same code for the primitives with side-effects, but BMFull performs them in 838 ms, whereas BMPartial performs them in 739 ms. A more careful analysis is thus required to balance the benefits and disadvantages of AVX512 instructions.

4.2 Q6 – Mixed Selectivity SLDP Primitives

Q6 contains five SLDP filters, an arithmetic SLDP projection, and a final aggregation. Across all input vectors, the second filter leads to a selectivity smaller than 0.15, triggering a threshold-based switch in the filter's representation.

For the results shown in Figure 9b, our first observation is that BMFull+SVManual is slower than BMFull+BMPartial by 0.3% even though SVManual performs better than BMPartial on the individual primitives in this query. This difference is due to the cost of converting the representation from BM to SV (see Section 3.4). Similarly, BMFull+SVPartial also suffers from the conversion overhead, but the difference only amounts to 2%.

Next, we observe that all mixed strategies perform better than non-mixed ones. BMFull+BMPartial and BMFull+SVManual are faster than BMFull by 1.3× because Full becomes wasteful after the selectivity drops below 0.15. SVManual using explicit SIMD is slower by 1.04× because it is only sub-optimal in the first two filters. The SISD Selective strategies BMPartial and SVPartial are slower by 1.6× and 1.4×, respectively, because they do not take advantage SIMD instructions for SLDP operations.

Finally, the SLDP primitives dominate the total running time for this query, so we do not observe the slowdown caused by AVX512 registers. Thus, for queries with a structure similar to Q6, the benefits of AVX512 outweigh its disadvantages.

4.3 Q4 – Low Selectivity, Inefficient Parallelism

Q4 is a join of two tables (LINEITEM, ORDERS) followed an aggregation and an order-by operator. The join's build side (i.e., ORDERS) has two SLDP filters. The first filter induces a selectivity of 0.3 and the second further reduces it to 0.1. The join's probe side (i.e., LINEITEM) has a filter with a selectivity >0.6 and a hash table probe that contains the multiple primitives. Although the probe's hashing primitive is an SLDP operation, its other primitives contain multiple indirect lookups (e.g., accessing hash table entries or the keys within those entries for exact comparison). Because these operations are called in a loop to find all potential matches during the join, they constitute the bulk of the running time. Unfortunately, their data-parallel versions are inefficient. For example, the comparison primitive, which we manually implemented for the SVManual strategy, contains three GATHER instructions: one to collect the keys from the right side of the join, one to collect the hash table entries, and one to collect the keys within the hash table entries for comparison. SIMD instructions can cause performance degradations in these primitives. Thus, we predict that a DBMS should use BMFull to perform the SLDP filters and hashing, then switch to using SVPartial, not the SIMD SVManual, for the complex primitives within the join probe.

The results are shown in Figure 9c. For the primitives we optimized, our predicted strategy, BMFull+SVPartial, is indeed the optimal one. The second best choice is BMFull+BMPartial, which also switches from SIMD to SISD code for complex primitives. Notice the performance degradation caused by SIMD vectorization, as mentioned in Section 3.2. Unlike the previous queries' results, BMFull+SVPartial performs better than the BMFull+SVManual by 1.1× because the latter relies on SIMD vectorization for complex primitives. BMFull is also slightly worse (by 1.04×).

Its performance degradation is attenuated because it requires fewer gather instructions than SVManual (e.g., one instead of three in the comparison primitive). Instead, it relies on faster vector_load instructions to load all elements in a vector, regardless of whether they are selected or not. The SISD Selective strategies BMPartial and SVPartial are worse by 1.1× and 1.05×, respectively. This degradation is because they do not optimize SLDP primitives.

When considering the total running time, AVX512 registers cause a slowdown due to CPU frequency throttling. Thus, BMFull becomes slower than BMPartial despite sharing the same code for side-effect-full primitives. As such, the disadvantages that the DBMS incurs with AVX512 outweigh its benefits on the full query.

5 Related Work

There has been previous work dedicated to optimizing primitives in the Vectorization processing model [2]. They mostly differ from this paper in that they do not consider the impact of filter representation in their performance. We now discuss this work and how they relate to our analysis.

Sompolski et al. compared the cost of vectorization to data-centric query compilation [15] and combined the two approaches to generate new vectorized primitive at runtime, which is the method we use in Section 3.3. Using only the SV representation, the authors compare vectorized primitives' performance under various compute strategies (e.g., Full versus Selective). They, however, do not provide a way to switch between these strategies.

Kersten et al. performs a similar analysis for whole queries in addition to individual primitives [6]. They developed a SIMD implementation of primitives using SVs, just like our SVManual strategy. Like ours, their implementation was effective on compute-heavy queries (e.g., Q6 in Section 4.2) and ineffective on memory access-heavy queries (e.g., Q4 in Section 4.3).

Răducanu et al. recognized the importance of switching compute strategy at run-time [14]. For each operation, the authors implement several *flavors* (i.e., different ways to perform the same operations). They then use reinforcement learning (RL) to switch between the best *flavors* dynamically at runtime. This approach is more general than our micro-benchmarks because it takes into account changes in system load. SV is the only representation considered, limiting the range of adaptivity on Update tasks. As future work, we could employ a similarly dynamic strategy to determine our thresholds.

6 Conclusion

This work analyzed the impact of filter representation (i.e., Bitmap vs. Selection Vector) and compute strategy (i.e., Full vs. Selective) on the performance of the vectorized primitives in an in-memory analytical DBMS. We identified the factors that influence performance: number of tuples processed, iteration logic, and core operation time per tuple. We explained how each combination of representation and compute strategy balances between these three factors. Full has the cheapest iteration logic, processes all tuples, but spends less time on each tuple when SIMD vectorization is possible. Full is, however, only available with Bitmaps on Update primitives. Selective with SVs has a cheaper iteration logic than Selective with Bitmaps, and is more amenable to SIMD vectorization. We confirmed these observations with several micro-benchmarks. Finally, we showcased the benefits of our analysis on OLAP queries with multiple primitives and consistently achieved the best performance. Our performance gains over the best techniques that do not adapt filter representation and compute strategy can be up to 1.3×.

Acknowledgments

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822933), Google DAPA Research Grants, Siebel Scholars program, and the Alfred P. Sloan Research Fellowship program. **TKBM**.

A Code Examples

```
MultVecByConst(Vec<double> in, double val,
               Vec<double> out, SelVec sel):
  // Assume size is multiple of 8
  for (i = 0; i < sel.size; i += 8):
    idxs = avx512_load(sel + i)
    in_vec = avx512_gather(in, idxs)
    result = avx512_mul(in_vec, val)
    avx512_store(out + i, result)
    // Alternative:
    // avx512_scatter(out, result, idxs)
```

Figure 10: Multiply Vector by Constant with SV and SIMD – Reading an element involves an indirection to first obtain its index using the GATHER instruction. The last SCATTER instruction is only necessary to maintain consistent indexes between the input and output vector during multi-step Map operations. It should otherwise be avoided due to its slowness.

```
MultVecByConst(Vec<double> in, double val,
               Vec<double> out, SelVec sel):
  for (i = 0; i < sel.size; i++):
    idx = sel[i]
    result = val * in[idx]
    out[idx] = result
```

Figure 11: Multiply Vector by Constant with SV and without SIMD – Reading an element involves an indirection to first obtain its index.

```
MultVecByConst(Vec<double> in, double val,
               Vec<double> out, Bitmap bitmap):
  for (word in bitmap):
    while (word != 0):
      t = word & -word
      r = __builtin_ctzl(word)
      idx = k * 64 + r
      result = in[idx] * val
      out[idx] = result
      word ^= t
```

Figure 12: Multiply Vector by Constant with BM

```
MultVecByConst(Vec<double> in, double val,
               Vec<double> out):
  // Assume size is multiple of 8
  for (i = 0; i < in.size; i += 8):
     in_vec = avx512_load(in + i)
     // Actual Operation
     result = avx512_mul(in_vec, val)
     avx512_store(out + i, out_vec)
```

Figure 13: Multiply Vector by Constant Full Compute + with SIMD

```
MultVecByConst(Vec<double> in, double val,
               Vec<double> out):
  for (i = 0; i < in.size; i++):
     result = val * in[idx] # Actual Operation
     out[idx] = result
```

Figure 14: Multiply Vector by Constant Full Compute + without SIMD

References

[1] 2021. NoisePage – Database Management System Project. https://noise.page.

[2] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[3] G. Graefe. 1994. Volcano An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135.

[4] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 293–304.

[5] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer’s Manual.*

[6] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222.

[7] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wes McKinney, and Andrew Pavlo. 2021. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4, 534–546.

[8] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 1–13.

[9] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 101–113. https://doi.org/10.14778/3425879.3425882

[10] Michael Larabel. 2018. Intel Tightens Up Its AVX-512 Behavior For The LLVM Clang 10 Compiler. https://www.phoronix.com/scan.php?page=news_item&px=LLVM-Clang-10-AVX512-Change.

[11] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN’19).* Article 10, 7 pages.

[12] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091.

[13] Kenneth A. Ross. 2004. Selection Conditions in Main Memory. *ACM Trans. Database Syst.* 29, 1 (March 2004), 132–161.

[14] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD’13).* 1231–1242.

[15] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN’11).* 33–40.

[16] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/.

[17] M. Zukowski and P. Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.* 35 (2012), 21–27.