Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans

Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, Hasso Plattner

Hasso-Plattner-Institut
August-Bebel-Str. 88
14482 Potsdam, Germany

markus.dreseler@hpi.de

***Abstract*—Recent work has started to combine two approaches for faster query execution: Vectorization and Just-in-Time Compilation (JIT). Combining the advantages of block-at-a-time and tuple-at-a-time processing opens up opportunities for performance improvements. In this context, we present the *Fused Table Scan*, a just-in-time-compiled scan operator for consecutive table scans. It makes use of the Intel AVX-512 instruction set to combine multiple predicates in a single scan. This reduces the number of instruction-level branches and makes memory access more efficient. We show that the Fused Table Scan doubles the scan performance in most cases and can achieve a speed-up of up to a factor of ten over sequential execution. Furthermore, we discuss which query plans profit from the Fused Table Scan and how the operator code can be generated at runtime.**

I. Introduction

Relational databases rely on efficient mechanisms to filter data in a table. While secondary indexes can speed up some queries, they consume memory, have high maintenance costs, and are suitable for queries that select few rows [9]. Thus, a significant share of queries requires a sequential scan of the table or intermediate data. For some applications, the need to enable fast scans on unindexed data is paramount. In fact, this is the core of the row versus column store debate for main memory databases [16].

Generally, the performance of operators in main memory databases is limited by either the CPU, the memory latency, or the memory bandwidth. One might expect the sequential scan to be limited by the memory bandwidth. As value comparisons in the CPU should be cheap, it should be an issue of the CPU stalling while waiting for more data to arrive. In Section II, we will show that this is incorrect and that the memory bus holds untapped resources that the scan can use.

Existing improvements of the sequential scan have been grouped into two categories [6]: *Block-at-a-Time Execution* scans a block of data from the table using vectorized operations and produces intermediary position lists. This enables the use of more efficient SIMD operations, but requires the results to be materialized and then consumed by a following operator. *Data-centric Compilation* uses Just-in-Time Compilation (JIT) to generate a tight, optimized loop that processes one tuple at a time. While this keeps data in memory for as long as possible and reduces the overhead of unnecessary method calls in the DBMS, it does not take advantage of SIMD. Recent work started to combine SIMD and JIT to vectorize across the boundaries of single operators [6, 13].

In this work, we combine these two concepts to improve consecutive table scans, that evaluate multiple predicates in a row. We show how the AVX-512 SIMD instruction set [7], which is available on the current generation of Intel Xeon Processors, allows to perform a series of vectorized scans without materializing. As a result, we present the *Fused Table Scan* operator, which achieves a 2x speedup in most cases and a 10x speedup in the best case when compared to traditional sequential scans.

The paper is organized as follows: In Section II, we present a naïve implementation of a consecutive table scan and analyze its limitations. We then present the implementation of a Fused Table Scan in Section III and discuss its use of AVX-512 instructions. We evaluate the performance in Section IV, where we compare our operator with SISD and AVX2 implementations. In Section V, we explain how the binary code for the Fused Table Scan can be generated during JIT compilation. Related work is presented in Section VI and a summary is given in Section VII.

II. Motivation

We will use the following simplified SQL query to demonstrate why a data-centric execution that does not use SIMD misses out on optimization opportunities:

```
SELECT COUNT(*) FROM tbl WHERE a = 5 AND b = 2
```

We make the following assumptions:

1) All data is in memory.
2) The table is stored in column-major format, i.e., the values of column *a* are stored contiguously in memory[1].
3) Values have a fixed size. This could either be because they are fixed-size by nature (such as the integers in this example) or because a compression scheme such as dictionary encoding [1] is used.

[1]It can, however, be horizontally partitioned into chunks or morsels.

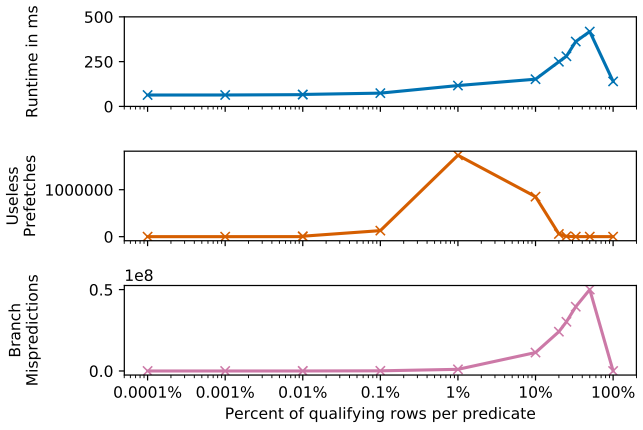

Fig. 1. For a fixed table size of 100 million rows and varying selectivities, the runtime correlates with the number of useless prefetches and the number of branch mispredictions.

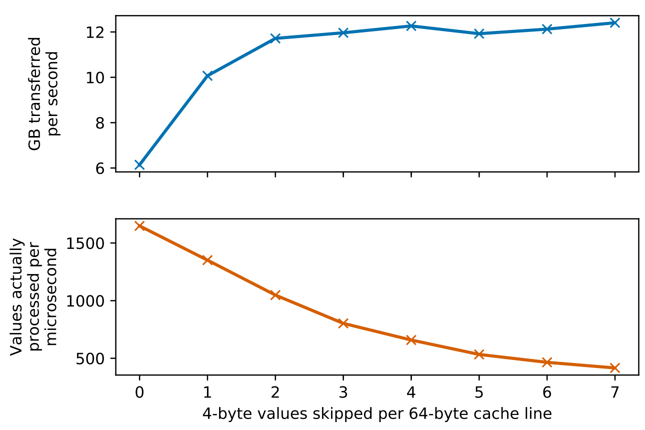

Fig. 2. A naïve SISD scan cannot utilize the available bandwidth. Only when parts of the cache line are skipped, it can reach the maximum available bandwidth. Of course, this reduces the number of actually processed values.

With these assumptions, a naïve data-centric implementation could result in code like this:

```
int total_results = 0;
for (pos_t i = 0; i < col_a.size(); ++i) {
  if (col_a[i] == 5 && col_b[i] == 2) {
    ++total_results;
  }
}
return total_results;
```

For now, let us leave the auto-vectorization that may be done by the compiler aside. In Section IV, we will show that it suffers from similar drawbacks. Two cursors are used for the two columns `col_a` and `col_b`. The first value is loaded into the CPU and compared to the search value. Because of short-circuit evaluation, the second value is only loaded and compared if the first value matches. So much for the theory.

In the real world, the prefetcher will speculatively load the value for the second column if it expects `col_a[i] == 5` to be true. If the prediction fails, most of the data loaded by the prefetcher is discarded. Most notably, this happens when half of the rows qualify and branch prediction becomes a 50/50 decision.

A second performance bottleneck is the condition itself, which gets translated into a compare `CMP` and a conditional jump `Jcc`. While both the compiler and the processor's branch predictor[2] try to optimize the execution order so that the number of jumps is reduced, they cannot guarantee a linear execution flow. If a branch misprediction occurs, the results of the speculative execution have to be rolled back.

We measured the number of rollbacks using PAPI and the `branch-misses` and `l2_lines_out.useless_hwpf` performance counters. The former gives us the number of branch mispredictions. It is available in PAPI as `PAPI_BR_MSP`. The latter measures the number of cache lines that were loaded by the hardware prefetcher but were never used and finally evicted. It was introduced with the Skylake architecture [8] and is not available in PAPI's master branch yet, but can be imported using `PAPI_event_name_to_code`.

Figure 1 shows how these counters are influenced by the selectivity of the first predicate[3]. As the number of qualifying rows and thus the probability of a match increases, the branch prediction gets worse. At 0.00001%, the branch prediction that assumes a non-match is almost always right. At 10%, even the best branch prediction will be correct in only 9 out of 10 cases. These mispredictions lead to expensive rollbacks and increase the overall runtime. At 100%, the branch prediction is always right, so the number of branch mispredictions, useless prefetches, and thus the runtime, decreases.

Finally, comparing the integers one by one is too slow to make use of the available memory bandwidth. We can show this with the following experiment: Instead of comparing every integer, only every n-th integer is scanned. This reduces the number of compares, but not the number of cache lines that have to be loaded. For $n = 4$, this doubles the bandwidth, as can be seen in Figure 2. Of course, skipping entries in the table is not a viable approach to improve the scan. Thus, to make use of the available bandwidth of 12 GB/s, more integers have to be scanned at the same time.

III. Implementation of the Fused Table Scan

In this section, we will describe the implementation of a new scan operator we refer to as *Fused Table Scan*. We use the same SQL query as before, searching for value 5 in the first column and 2 in the second column. The dataflow is visualized in Figure 3 and example code can be found online[4].

Our approach starts the same way that most SIMD-based sequential scans do. A part of the first column is read into an AVX register. For the example, we use 128-bit registers instead of the 512-bit registers offered by AVX, simply because they are easier to visualize. As we will show in Section IV, the actual execution would almost always be faster on bigger

[2]Of recent notoriety because of the Spectre and Meltdown attacks.

[3]The test system's configuration is given in Section IV.

[4]https://github.com/hyrise/fused_table_scans

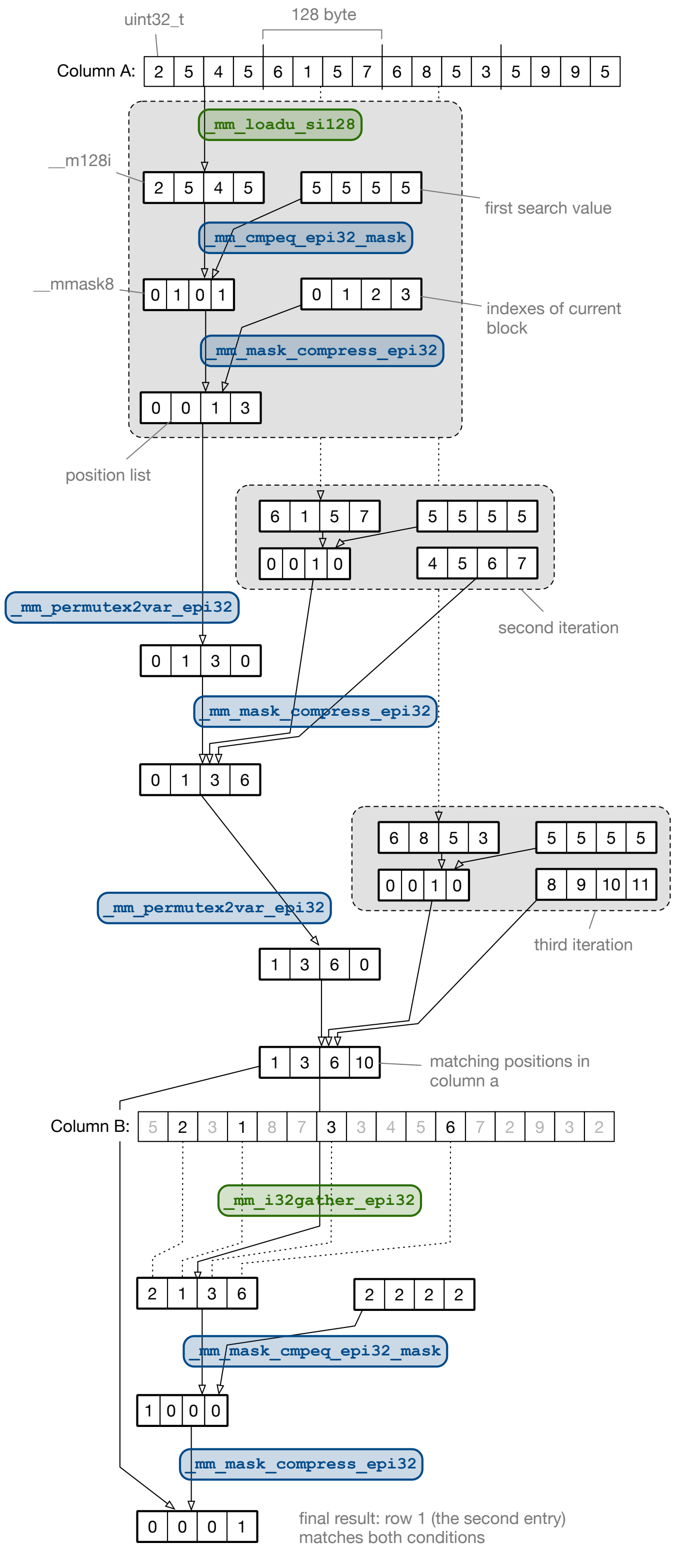

Fig. 3. Data Flow of the Fused Table Scan. Instructions printed in blue are AVX-512 instructions.

512-bit registers. The function `_mm_loadu_si128` loads 16 bytes into a 128-bit register. In our example, we look at 4-byte integers, so this makes four values per register. We compare these four values with the search value using the `_mm_cmpeq_epi32_mask` function. This gives us a bitmask with four entries in which `1` stands for a match on the first search criterion. Traditional block-at-a-time (i.e., vectorized) implementations would now scan the remaining blocks and finally return the complete bitmask to the next operator.

Instead, we use this bitmask to perform the next scan in the chain. Doing so has the advantage that the bitmask can remain in the AVX register instead of being written into a CPU cache or, even worse, to memory. First, we convert the bitmask into a position list (an offset list of the matching positions). For this, we prepare an AVX register that holds all positions in the current block. In the first iteration of the loop, this results in the register `(0, 1, 2, 3)`. Applying a bitmask where the second and the fourth row match the criterion to that register gives us `(0, 1, 0, 3)`. Because the position list should be dense, we shift the register so that we get `(0, 0, 1, 3)`. AVX-512 can do these two steps in one instruction, `_mm_mask_compress_epi32`. As we keep track of the number of entries in the mask (here it is two), the leading zeroes can be ignored in following instructions. This removes any ambiguity between `0` as an empty value or as the offset of the first row.

Now that we have a list of all positions from the first block, we repeat the previous steps for the next four values. The indexes from the second iteration are appended to the position list by first shifting the existing values using `_mm_permutex2var_epi32` and then bringing in the new values with `_mm_mask_compress_epi32`.

We repeat this until no more entries can be appended to the position list. The position list may hold less than four entries, for example if it already held three entries and the iteration produced two more results. In these cases, we first process the incomplete list and then start a new list with the two new results. Because we keep track of the number of entries, we do not have to worry about leading zeroes.

At this point, we have four positions within the table (i.e., row ids) where the first criterion `a = 5` is fulfilled. We now use `_mm_i32gather_epi32`, which allows us to use the position list as indexes into the second column. This saves us from issuing multiple load instructions and sequentially building an AVX register by hand. To our understanding, it reduces the number of instructions and involved registers, but not the number of cache lines loaded. We expect bigger speed-ups if the gather instructions could somehow assemble the register somewhere closer to the memory hardware.

We use `_mm_mask_cmpeq_epi32_mask` to scan the column for the search value and end up with yet another bitmask. This bitmask, applied to the position list that we used as input for the second column, is `1` in places where `a = 5 AND b = 2`. By using the `_mm_mask_compress_epi32` and `_mm_permutex2var_epi32` steps from before, it can be converted into a position list. That list can then be used as input for the next operator in the execution plan.

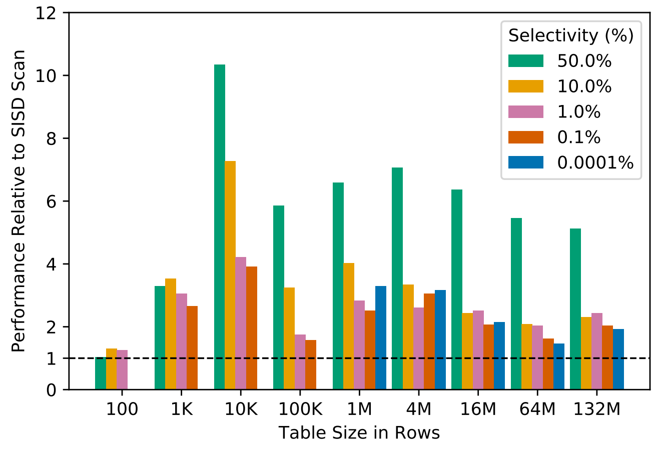

Fig. 4. Across different table sizes, our Fused Table Scan outperforms data-centric SISD (with auto-vectorization) for all measured selectivities. In most cases, the relative performance is doubled.

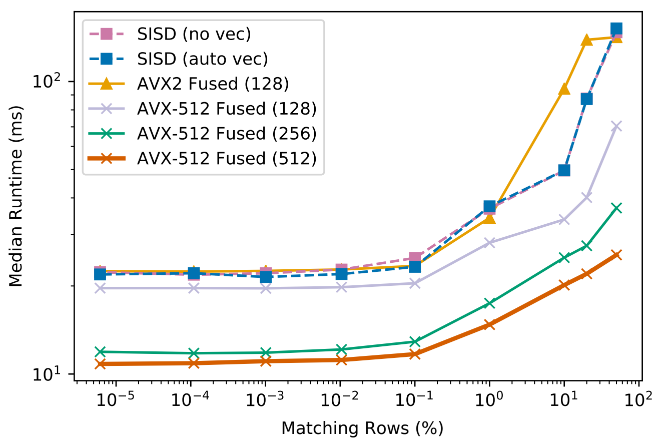

Fig. 5. When comparing different instruction set extensions and register sizes, we see that (a) AVX-512 has an advantage over AVX-2 and (b) the register size contributes significantly to the performance benefit.

If we had a third consecutive scan, we would likely have too few positions remaining to fill an entire register. Thus, we first repeat the previous steps until we have enough results to continue with the third column. The next steps are the same as the ones executed for the second column.

The presented approach is an improvement over the naïve code in a number of ways:

1) By using SIMD instructions, more comparisons are performed per cycle. This allows us to break through the bandwidth ceiling described in Section II.
2) Because less data is prefetched but not used, the available bandwidth is used more efficiently.
3) As the code has fewer conditions, the number and cost of branch mispredictions is significantly reduced.
4) In contrast to existing SIMD implementations, the results of the comparisons (i.e., the bitmasks) are not materialized but remain in the AVX registers. This reduces the transfer cost.

In the example, the values neatly fit into a native data type. Other optimizations, such as null suppression [18] and SIMD-BP128 [12], change this layout to fit more data into a given amount of memory. Our solution is not limited to native data types and could incorporate the needed unpacking steps.

Even though the example uses only 128-bit registers, it relies on AVX-512 instructions. This is because AVX-512 can perform multiple computations in one step where previous AVX versions would have required multiple instructions. First, we use masked instructions to apply the bitmask from previous instructions to the next step. An example of this is `_mm_mask_cmpeq_epi32_mask`. It compares two packed integers for equality, but only outputs `1` for those where the corresponding field in the input mask is set. Before AVX-512, this required two steps: `_mm_cmpeq_epi32` and `_mm_and_ps`.

Second, we use `_mm_mask_compress_epi32`, a new swizzle instruction. It allows us to select values from an AVX register, move them to the first slots and fill up the result vector with elements from a second input vector. This is key to the step where we build a full position list for the second predicate.

For measuring the benefit of these new AVX-512 instructions, we built equivalent functions using AVX2. Because something as short as `_mm_mask_compress_epi32` became 32 lines, we do not print the equivalent functions here and instead refer to `_mmX_mask_compress_epi32` in the `REG == 128 && !AVX512` configuration in `avx_scan.cpp`. The performance advantages of AVX-512 will be discussed in Section IV.

IV. Evaluation

In this section, we evaluate several aspects of the Fused Table Scan's performance. Unless noted otherwise, we use the query that was introduced in Section III. We will discuss the impact of the tables' sizes, the predicates' selectivities, the number of predicates in the query, and the AVX register width with several experiments. All measurements were executed on an Intel Xeon Platinum 8180 CPU with 2.5 GHz base frequency, 3.8 GHz maximum turbo frequency and 2 TB of PC4-2666 main memory. The caches hold 32 KB (L1 data), 1024 KB (L2), and 38.5 MB (L3), but were flushed after each benchmark. Our code was compiled with gcc 7.2 and `-O3`.

Figure 4 shows the relative performance (i.e., the reduction in runtime) of the Fused Table Scan compared to that of a data-centric SISD implementation (as seen in Section II). The x-axis shows varying table sizes, while differently colored bars show different selectivities. Some bars are omitted, as 0.1% of qualifying rows would not return any rows for a table with 100 rows. The dashed line at 1x is the the performance of a data-centric SISD-scan which is used as the baseline. All configurations were measured at least 100 times. Our figure displays the median of these runs.

For all measured configurations, the Fused Table Scan outperforms the SISD-scan. In 32 of the 40 benchmark configurations, the performance is improved by at least a factor of two. This validates the claim made in this paper's title.

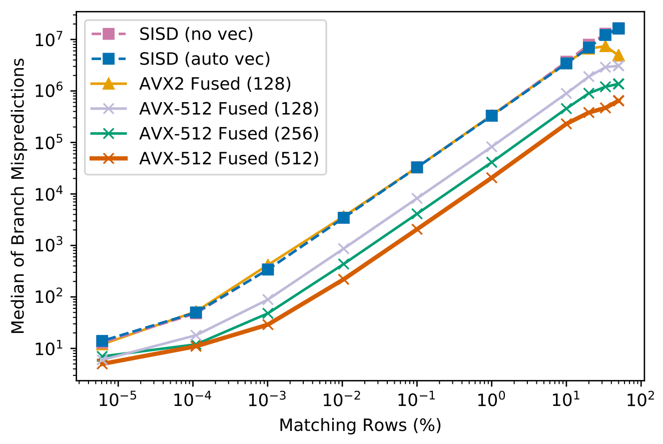

Fig. 6. The Fused Table Scan has a lower number of branch mispredictions. Because fewer speculative executions have to be rolled back, the performance is improved. The graph shows results for a table size of 32 million rows.

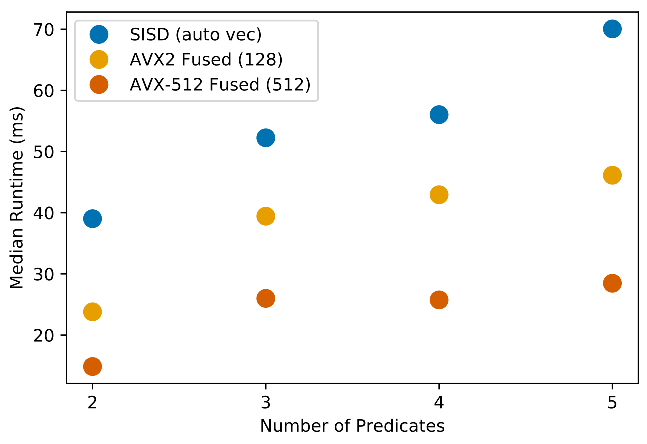

Fig. 7. The performance benefit grows with the number of predicates. Here, the first predicate matches for 1% of the rows and following predicates match for 50% of the rows. Similar results are measured for different configurations.

In Figure 5, a fixed table size of 32 million rows and a varying selectivity were chosen. Different implementations are compared: *SISD (no vec)* denotes traditional tuple-at-a-time approaches. *SISD (auto vec)* is the same code but uses auto-vectorization by the compiler. *AVX2 Fused (128)* is a reimplementation of the Fused Table Scan where AVX-512 instructions are replaced by their AVX2 equivalent. This is not necessarily the fastest implementation possible with AVX2, but gives us an idea how AVX-512 instructions that combine multiple AVX2 instructions improve the performance. Finally, *AVX-512 Fused (x)* is our Fused Table Scan with *x* being the size of the used registers.

We can see that the Fused Table Scan is faster in all configurations. It outperforms both the auto-vectorized SISD scan and the AVX2 backport. Furthermore, the influence of larger register sizes can be seen. Interestingly, the gap from *AVX-512 Fused* with 128-bit registers to 256-bit registers is bigger than that of 256-bit registers to 512-bit registers. Because this distance remains similar even as the runtime goes up, we do not believe that this is because of the memory barrier. Instead, it appears as if there are some 512-bit instructions that take longer than their corresponding 256-bit instruction.

The Fused Table Scan still requires some branching, for example when checking if new matches can be appended to the current position list or if a new position list has to be started. However, when comparing the number of branch mispredictions across the different implementations, it has a lower number of branch mispredictions than the SISD and AVX2 versions. Figure 6 shows this effect, which is part of the Fused Table Scan's performance advantage.

Finally, for Figure 7, we measured how the performance is impacted when the number of predicates is increased. Because of the gather optimizations made in the Fused Table Scan, we expect it to be less affected by a higher number of predicates than the SISD implementation. Not only is this of interest when looking at queries with multiple predicates (such as TPC-H Query 6), but also when the DBMS uses multi-version concurrency control (MVCC) and the validation of the visibility vectors is treated as a follow-up predicate. We scanned 32 million rows with a varying number of predicates. Here, for the first predicate, 1% of all rows qualify and for following predicates 50% of the remaining rows match. The results confirm our expectation that the benefit of the Fused Table Scan improves with a growing number of predicates.

V. Runtime Code Specialization

In the previous section, we have shown that the Fused Table Scan outperforms a tuple-at-a-time SISD implementation. Before the implementation given in Section III can become part of a query plan, it has to be generated by the JIT compiler. This is because a number of parameters are only known at runtime:

- The size of the scanned values. In the example, we used 4-byte integers, but other sizes are also possible.
- The data type of the values. While this makes no difference for the equality scan executed in the example, it matters for other comparisons. A four byte integer has to be compared differently than a four byte float.
- The comparison operator itself, where different AVX instructions need to be used for =, <, and so on.

For a single scan, this gives us ten data types[5] and six comparison operators. While sixty instantiations of the operator code might still be feasible, this number explodes once we look at the following scan in the chain of consecutive scans. As our performance benefit is largely the result of using the output from one scan as the input of the next scan, the code of the operator has to be tailored to the characteristics of both scans. If we disregard some minor optimizations, this leaves us with 3600 possibilities for two predicates. Many queries have more predicates, so generating all combinations is infeasible.

This problem can be overcome by using Just-in-Time Compilation to generate the code at runtime. Because the

[5]Signed and unsigned ints with 1, 2, 4, or 8 bytes plus float and double

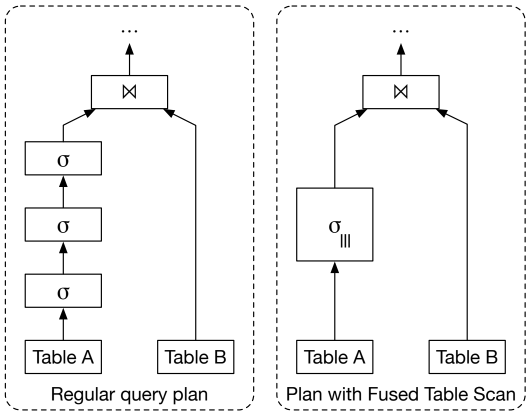

Fig. 8. Logical Query Plans for non-optimized and SIMD-optimized scans

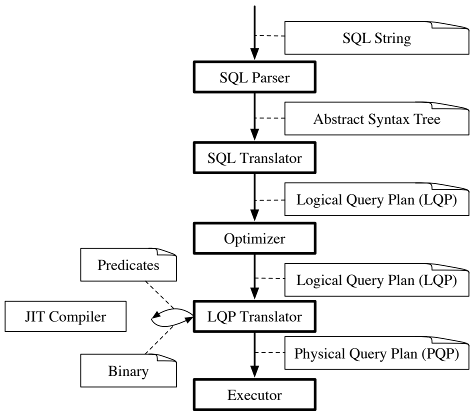

Fig. 9. The Hyrise optimizer works on logical query plans that contain relational algebra operators. These are then translated into executable operators by the LQP Translator.

JIT support in our research database Hyrise[6] is still under development, we will discuss what we consider to be the optimal setup. It generates an executable operator that acts as a drop-in replacement for consecutive scans as seen in Figure 8.

Hyrise uses a JIT execution model where generated code has the same interface as traditional database operators. This is comparable to Peloton's *Relaxed Operator Fusion* model [13]. Operators are written in C++, translated to LLVM IR, and shipped with the DBMS. During runtime, the IR is specialized with runtime information such as data types and actual values. The code from all operators in one pipeline is then inlined into a single, monolithic function. Subsequent calls to LLVM's optimizer and compiler then generate binary code that is optimized across operator boundaries.

For the Fused Table Scan, we deviate from this model and use code generation, where the actual code is generated during runtime. This is because code specialization would require us to have unspecialized code that compiles and executes properly. In turn, this would require all parameter combinations to exist in the unspecialized code. Luckily, the code that has to be generated follows a very static pattern and can easily be expressed as a code template. Because of this, we decided for code generation instead of code specialization.

Three abstraction levels of code could be used for generation: C(++), LLVM IR, or ASM. Because of the hardware-oblivious nature of LLVM IR, it is not a good choice for writing code that is tuned to a specific instruction set extension and vectorized by hand. The advantages of C(++) code are automatic register allocation and type safety, whereas ASM has the advantage of being closer to the hardware and requiring fewer steps to compile. For now, we found C(++) code easier to write and maintain. Especially when compiled operators are cached for future use, we do not see the additional compile time as a deciding bottleneck.

Before any actual code generation occurs, the DBMS has to identify potential uses for the Fused Table Scan in the query plan. This is done by Hyrise's optimizer, which operates on logical query plans (LQPs). These contain relational operators (e.g., joins) but do not define the actual implementation (e.g., a hash-join). The rule-based part of the optimizer translates the LQP using techniques such as predicate pushdown and predicate reordering, which make sure that predicates are evaluated as early as possible and in the most efficient order. When multiple predicates (σ in Figure 8) are identified as a chain, they are tagged to be translated as a Fused Table Scan.

The LQP Translator then invokes the JIT compiler, which generates code as visualized in Figure 3. Doing so consists of choosing the correct data sizes for the AVX intrinsics (e.g., `_epi32` when scanning four byte integers or `_ps` for single-precision floats). Also, the placeholder for the comparison (`_mm_cmpXX_epi32_mask`) has to be replaced according to the chosen comparison operator. Finally, the different predicates have to be fused. At this point, the JIT compiler has to make sure that the length of the generated position lists matches the data type of the column scanned in the following step. Looking at the example, this becomes important if the first column uses 4-byte integers and the second column 8-byte integers. The first predicate would generate four indexes into the second column, but the 128-bit AVX register can only hold two of the 8-byte integers. In this case, the JIT compiler has to split the list of indexes and perform twice the number of iterations when evaluating the following predicate.

VI. Related Work

Previous work has explored different ways of improving the performance of sequential scans. These use a variety of approaches, including compressing the data to shift work from the memory bus to the CPU [1], using SIMD to scan multiple values in a CPU cycle [20], using Just-in-Time Compilation (JIT) [3, 5, 14], or even offloading the scan to dedicated hardware [2].

A. SIMD in Main Memory Databases

Willhalm et al. [19, 20] use SIMD instructions to perform fast scans of bit-packed data (also known as null-suppression). They combine the steps needed to unpack bit-packed integers

[6]https://github.com/hyrise/hyrise

with the comparison step in a single SIMD pipeline. This allows them to benefit from the higher number of unpacked and compared integers than SISD and from a better utilization of the memory bandwidth due to the compression.

Broneske et al. [4] compare different implementations of a sequential scan that make use of modern CPU features. They compare how branch removal, loop unrolling, parallelization, and vectorization improve the effective throughput of scans.

Polychroniou et al. [17] discuss how new SIMD instructions, including the gather operations used in our approach, can be used to improve the performance of single operators. Included are algorithms for scans, hashing, bloom filters, partitioning, and sorting. For all of these, significant improvements can be seen when vectorization is used.

B. JIT-Compilation for Query Plans

Just-in-time (JIT) compilation is an established technique in modern database systems for reducing the interpretation overhead of traditional query execution. Given a query plan as input, query-specific code is produced based on runtime knowledge about the data schema, the query structure, and operator parameterization. This produces a low-level representation of the query-specific code that is subsequently compiled to native machine instructions and executed.

Many databases that use JIT focus on optimizing the hot loops of individual operators. Cloudera Impala [10] identifies tuple materialization, user-defined functions, and complex arithmetic and logic expressions as good candidates for such hot-spot optimizations. Query-specific versions for these operations are created by cross-compiling generic implementations to the LLVM intermediate representation and substituting known runtime values. Invocations of the generic functions can then be redirected to the specialized version.

Butterstein et al. [5] use a similar idea, with a focus on arithmetic and selection expressions in the PostgreSQL database. SparkSQL [3] follows a more generative approach to produce query-specific code. Nested arithmetic and logic expressions are represented as a tree data structure. These trees are programmatically transformed into Scala abstract syntax trees (ASTs) in a number of steps. These ASTs are fed to the Scala compiler to produce executable Java bytecode.

While these approaches provide some performance improvements, their optimizations are local to individual operators. To avoid tuple materialization between operators, HyPer [14] introduced a data-centric optimization approach. Complex query plans are split into multiple linear pipelines at materialization points (aka. pipeline breakers), which are necessary due to the structure of the query (e.g. aggregations). For each pipeline, a single tight loop is generated. Inside this loop, all operators in the pipeline work on the tuple before the next tuple is processed. Thus, tuple values stay close to the CPU and operator boundaries are dissolved.

C. Combinations of Vectorization and JIT

In the past, vectorization and JIT were considered to be irreconcilable [15]. Only recently, the interest in combining vectorization and JIT has risen.

Lang et al. [11] modified HyPer so that instead of working on a single tuple at a time, it now works on small blocks. These blocks are the basis for a vectorized scan that can be part of a JIT pipeline. Similar to our approach, that scan focuses on reducing the number of branches and enabling the use of SIMD instructions. The main difference to our work is our use of AVX-512 masks and swizzle instructions. This allows us to avoid an inner loop that iterates over single rows.

Gubner et al. [6] are among the first to use JITted AVX-512 for database operators. They show how compacting the data types from what is defined in the schema can help with vectorization. Furthermore, they present an AVX-512-based aggregation operator both in existing databases and as a standalone benchmark.

Menon et al. [13] show that careful materialization between operators can be better than strictly following the tuple-at-a-time approach. They precisely identify the problem that we have addressed and identify two solution methods:

> In the first method, the operator breaks out of SIMD code to iterate over the results in the individual SIMD lanes one-at-a-time. [...] In the second method, [...] the operator delivers its results in a SIMD register to the next operator in the stage. Both methods are not ideal. Breaking out of SIMD code unnecessarily ties up the registers for the duration of the stage. Delivering the entire register risks underutilization if not all input tuples pass the operator, resulting in unnecessary computation.

We have solved this problem by using gather instructions, which allow us to retrieve only the rows that have made it through the predicate. Figure 3 shows how the use of `_mm_i32gather_epi32` makes it possible to continue without breaking out of SIMD code even if parts of the intermediary result get thrown out by a predicate.

VII. Summary

Traditionally, vectorization and Just-in-Time compilation were seen as incompatible. Even with recent work, an important question remained: What happens if a predicate removes half the tuples pointed to by a SIMD register? Traditional approaches would either leave the SIMD mode and iterate over the register or leave these tuples until the end, causing unnecessary loads and comparisons on the way.

This paper presents the Fused Table Scan, which uses new AVX-512 gather, compress, and swizzle instructions to remove tuples from the AVX register without leaving SIMD mode. We evaluated it with varying register sizes, selectivities, table sizes, and number of predicates. In 32 of 40 measured cases, the performance was at least doubled. Drilling down into the causes for the runtime differences, we found the number of branch mispredictions to be an important factor and showed how our approach reduces these mispredictions by roughly an order of magnitude.

AVX intrinsics are needed as the compiler's auto-vectorization does not reach the same performance. Because of this, the operator code has to be generated by a JIT compiler in the DBMS. We described how the DBMS can analyze the logical query plan to identify use cases for the Fused Table Scan and how its code is generated.

Future Work shall include the integration of orthogonal optimization methods. Among these, we believe that the concept of bit-packing (aka. null suppression) can be most beneficial for our approach. The main challenge for this will be the extraction of single values as part of the gather step. This is because a bit-compressed value can span multiple bytes and will have to be decompressed before it can be processed.

Acknowledgements

We thank Intel for granting early access to a system with AVX-512 support and Thomas Willhalm for fruitful discussions on the topic.

References

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. “Integrating Compression and Execution in Column-oriented Database Systems”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*.

[2] Kathirgamar Aingaran, Sumti Jairath, and David Lutz. “Software in Silicon in the Oracle SPARC M7 processor”. In: *2016 IEEE Hot Chips 28 Symposium (HCS)*.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[4] David Broneske, Sebastian Breß, and Gunter Saake. “Database Scan Variants on Modern CPUs: A Performance Study”. In: *In Memory Data Management and Analysis (IMDM)*. 2015.

[5] Dennis Butterstein and Torsten Grust. “Precision Performance Surgery for PostgreSQL: LLVM-based Expression Compilation, Just in Time”. In: *Proceedings of the VLDB* 9.13 (2016).

[6] Tim Gubner and Peter Boncz. “Exploring Query Execution Strategies for JIT, Vectorization and SIMD”. In: *Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 2017.

[7] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Tech. rep. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (visited on 01/19/2018).

[8] Intel. *Intel Microarchitecture Code Named Skylake Events*. Tech. rep. URL: https://download.01.org/perfmon/index/skylake.html (visited on 01/30/2018).

[9] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. “Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?” In: *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*.

[10] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. “Impala: A Modern, Open-Source SQL Engine for Hadoop”. In: *Seventh Biennial Conference on Innovative Data Systems (CIDR)*. 2015.

[11] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation”. In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*.

[12] Daniel Lemire and Leonid Boytsov. “Decoding Billions of Integers Per Second Through Vectorization”. In: *Software - Practice and Experience* 45.1 (2012).

[13] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. “Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last”. In: *Proceedings of the VLDB* 11.1 (2017).

[14] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proceedings of the VLDB* 4.9 (2011).

[15] Styliani Pantela and Stratos Idreos. “One Loop Does Not Fit All”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[16] Hasso Plattner. “A Common Database Approach for OLTP and OLAP Using an In-memory Column Database”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*.

[17] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. “Rethinking SIMD Vectorization for In-Memory Databases”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[18] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. “The Implementation and Performance of Compressed Databases”. English. In: *SIGMOD Record* 29.3 (2000).

[19] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. “Vectorizing Database Column Scans with Complex Predicates”. In: *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 2013.

[20] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. “SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units”. In: *Proceedings of the VLDB* 2.1 (2009).