OPTIMIZING RELATIONAL QUERIES ON A BIT-PARALLEL DATABASE LAYOUT

Dominik Häner, Micheal Sommer, Sebastian Heinekamp, Matthew Weingarten

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Bit-parallel database layouts show significant performance gains for full table scans and accelerating training for Any-precision machine learning models by concurrently processing bits of multiple values packed into a single word. Currently, these layouts are optimized for narrow use cases and lack the flexibility to handle generic database workloads without incurring overheads. We introduce optimized implementations of a small set of complex relational queries on a bit-parallel layout to extend the versatility thereof.

1. INTRODUCTION

Emerging layouts for in-memory databases show promising results for scans and machine learning (ML) applications. Packing bits of multiple values into a single processor word, introduced in the *BitWeaving* layout, gives rise to more efficient scans by processing combined bits of data in every CPU cycle [1, 2]. It outperforms SIMD scans with basic comparison operators such as $=, \neq, \leq, <, \geq, >$ with constant values. This is in part due to SIMD not being able to fully utilize the width of a word (such that it must pad any word to the word-bit boundary). While the full table scan is a central element of any database management system (DBMS), it is still missing optimized implementations of more complex queries to support general-purpose database workloads.

Furthermore, bit-parallel techniques see applications to support training with low-precision, often used to decrease training time ML models. Bit-parallel layouts, such as *MLWeaving*[3], minimize the overhead of retrieving values at arbitrary precision levels. The layout ensures that only bit values that are of interest must be read, instead of loading the full-precision values and discarding the undesired bits.

Contribution. We present optimized implementations of three specific relational queries on a bit-parallel layout, including a dynamic scan, an aggregation, and a join operation.

```
Q1: SELECT * FROM R WHERE R.a < R.b;
Q2: SELECT SUM(c) FROM R WHERE R.a < R.b;
Q3: SELECT * FROM R, S WHERE R.a % S.b = S.c;
```

Q1 is a scan on non-constant values as a comparison, (i.e a comparison between two values in a tuple), Q2 returns an aggregate `SUM` of a scan, and Q3 is a join operation with a join condition containing a modulo operation.

2. BACKGROUND

Layout.

Our bit-parallel layout is a slight adaptation of the ML-Weaving layout. The layout is visualized in figure 1, where a table with two columns and 256 rows is encoded into a bit-parallel layout. We refer to a *block* of data as 512 values encoded into 32 cache-lines of 16 32-bit words, with each word containing bits of multiple tuple values. We use the term tuple and sample interchangeably, referring to the values of the pre-weaved table. The amount of tuples contained by a block depends on the number of columns. The fewer columns in the original format, the more dense the bit-parallel representation.

As in the *MLweaving* layout, we store up to 512 values in one block. 512 is a common factor of current cache line sizes, so given properly aligned data, we load (at least) a bit of each value per transferred cache line. To keep bit-wise operations consistent, we focus on a database of exclusively 32-bit unsigned integer values.

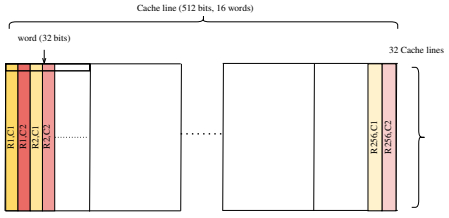

Fig. 1. *Illustrates a single block of the database layout. The upper figure shows the data in a standard row based format, which is converted to a bit-parallel layout. In the weaved format, each word (32 bits) contains a single bit of 32 different database values. Each block carries 512 32-bit word, with a total size of 2 KiB.*

Our implementation differs from MLWeaving in how individual samples are padded. While MLWeaving pads each

sample to at least 64 features[3], we pad such that no sample crosses a 64-bit border. If we have at most 32 features per sample, we can save on memory overhead, as well as compute over more samples in parallel at once.

This layout can also easily be extended to work as a version of a *column family store*[4]. In a column family store, values that are often accessed together, are stored together. One could easily store different column families in different blocks, maximizing memory efficiency (i.e. only loading relevant values), as well as potential parallelization of computation when doing full table scans. Without leaving a buffer for future features, the whole table would have to be recomputed every time new features are added to a column family, unlike regular column family stores.

3. SELECT * FROM R WHERE R.A < R.B

As the simplest of the three queries, in Q1 we attempt to push the limits of parallelized computing and pipelining. Instead of reconstructing every feature of every sample, we return a binary result vector that indicates which samples match the query. This lets us focus on optimizing the bit-wise query, which would otherwise be outweighed by the reconstruction efforts. Reconstruction is handled in greater detail in the other queries.

The bit-wise less than computation can be seen in Algorithm 1. For this test, R.a and R.b are always neighboring features. We need to store two states (Res and $Temp$) for each sample handled. The computations using these states create a dependency between the bits.

Algorithm 1: *Bit-wise less than on a 32 bit uint*

Result: $R.a < R.b == Res$
$Res \leftarrow 0$;
$Temp \leftarrow 0$;
for $i \leftarrow 0$ *to* 31 **do**
  $xor \leftarrow R.a[i] \bigoplus R.b[i]$;
  $b \leftarrow R.b[i] \wedge xor$;
  $b \leftarrow \neg Temp \wedge b$;
  $Res \leftarrow Res \vee b$;
  $a \leftarrow R.a[i] \wedge xor$;
  $Temp \leftarrow Temp \vee a$;
end

Per block we compute over, besides reading each cache line once, we repeatedly access the state of each sample. For this, we need to store two binary values per sample in the block. Even in the naive worst case, this fits in the L1 cache of current processors. When computing in parallel, we only use 128 Bytes to store the state. Therefore, when handling this query, we only deal with compulsory cache misses in a cold cache scenario.

A Naive Start. An initial naive implementation uses three nested loops. An outer loop to indicate which block is being handled, a loop within to indicate which of the 32 bits is handled, and an inner loop which defines which sample within the block we are computing.

Finding the index of the correct word, and the correct bit within requires some calculations based on the number of features per sample, and use division and modulo to compute. Given the simple operations needed for the actual algorithm, this indexing work is a bottleneck. We alter the loop structure by replacing the inner loop with two loops: one over the 16 32-bit words per cache line, and an inner loop over samples per word. This lets us simplify the indexing to only use addition, and improve the loading pattern. We can further optimize this naive version by unrolling these loops and using scalar replacement on the state variables.

Parallelizing. The true potential of our layout only applies once we start computing over several bits in parallel. The speed-up factor is relative to the number of samples we compute over in parallel. As a first step, we use bit-wise operations over 64-bit words (to fit our padding). Our inner loop additionally requires one shift to align R.b with R.a and the bit-wise state.

This new approach means we can remove the loop over samples per word, but instead need another section where we read the result for each sample out of the state. This can either be done by shifting the state to the correct bit or using a mask to read the correct bit and shifting the mask accordingly.

We can again apply optimizations such as loop unrolling and scalar replacement. The goal of unrolling is to keep the pipeline full, as the simple logical functions can be done on several ports. Our main bottlenecks are the shift function's low throughput, as well as loading and storing our data and results, and the state dependency between bits.

Vectorization. We now further parallelize by using AVX vector intrinsics to compute over 256 bits in parallel. States are also computed as vectors, so we need an extra step where we read the Res state vector into 64-bit words we can then read from. Besides the greater level of parallelism, we also gain some speed as the vector instructions have the same, or better throughput than the regular ones (notably shifting).

We find the optimal unrolling factor of the computation section given the state dependency. The challenge is to keep the pipeline full. We test unrolling to read up to 16 cache lines per loop.

All Zero Feature Check. Given that all computations for a vector (theoretically) require at most 7 cycles after loading, any attempt at optimizing it away might be nullified by the overhead caused by failed branch predictions and extra tests. One such attempt is to check whether every relevant bit of a loaded vector is zero. If this is the case, we can skip computing the result, as we know it will not

change. An "all-one" features vector would have the same result. The fewer samples we have per vector, the higher the probability that we can skip computation.

We do this with a bit-wise AND with a mask vector, which picks out $R.a$ and $R.b$. If the result is all zero, we move on to the next vector. The test instruction takes 3 cycles. Different granularities are tested, attempting to find the ideal amount of if-statements per vector tested, such that we have less branch miss overhead than actual gain from skipping computation. We also try testing after computing the xor, which would cover all-one and all-zero cases, at the cost of a shift and an xor before the test. Loop unrolling is also attempted.

Early Pruning. As testing shows we are memory-bound in cold-cache scenarios, one possible optimization is to reduce the amount of data we need to load, let alone process. This can be achieved with early pruning; if we have a result after some amount of bits processed, we can skip loading and computing the lower bits. To do this, we need to check whether $Temp$ or Res has been set after every round of computation. We do this with a bit-wise AND with a mask that picks out the relevant bits. Thankfully, this intrinsic returns an integer value, skipping a conversion step. The readout nonetheless causes considerable overhead.

We again test different granularities, as well as unrolling and combining with the all-zero check, to better handle small feature values (with a lot of all zero lines).

4. SELECT SUM(R.C) FROM R WHERE R.A $<$ R.B

Since they are very similar, we build on top of our implementation of query 1 for this task. This includes creating the result vector as described in algorithm 1, while at the same time reconstructing the third feature of the original data, which is our hard-coding of R.c. Having done this, we simply perform a conditional summation of the reconstructed values based on the binary result vector.

Naive versions.

As a first optimization, we also simplified index computations, as well as using scalar replacement for state variables. This process was entirely analogous to how it was for query 1. Loop unrolling was also performed.

Vectorization of final summation. Since loop accumulation is a textbook example for a highly parallelizable operation, the final, conditional summation of reconstructed values in R.c was identified as an easy target for vectorization. Using the binary result array as a mask, we can easily accumulate up to 8 values at once. This optimization was tested separately for the versions using scalar replacement and loop unrolling.

Complete vectorization. Finally, we move towards parallelizing the entire computation using AVX2 instructions. For calculating the binary vector, this can be done as described for query 1. When reconstructing the values to be summed up, we are at odds with the modified MLWeaving format: we allow multiple samples per 64-bit word, but for each sample, we need to reconstruct a 32-bit value to be summed up later. Therefore, for simplicity and maximum performance, the fully vectorized version of query 2 only works for the case that the number of features is 32 so that the 8 32-bit values reconstructed from an AVX2 input vector all fit in a single AVX2 output vector. Further loop unrolling and early pruning of the vectorized version were omitted for this query since in most cases they were not shown to have a large impact on performance in Query 1.

5. SELECT * FROM R,S WHERE R.A % S.B = S.C

The join operation is the most challenging operation of the three queries, especially considering a join with the modulo operation. For the scope of this paper, the query takes as an input two relational tables R and S, both in bit-parallel layout and an output table containing the indices of the tuples fulfilling the join condition. The query is limited to any relation with up-to 32 columns m and any row size n that fits divided into blocks. Note that any cost analysis in this section is made with an Intel Haswell CPU in mind.
Fundamentally, how the values are stored form the basis of a bottleneck, as it is unclear whether one can efficiently compute the modulo in a bit-by-bit fashion, allowing us to fully exploit the bit-parallel nature of the values. Early on we explored a bit-by-bit modulo evaluation, however after early benchmarking we discarded this idea after showing poor performance. We present our implementation of query 3, which sees the join split up into two parts: in phase one we reconstruct the data and phase two evaluates the modulo join condition on the reconstructed values.

Algorithm 2: *Block reconstruction*

```
block ← start of block in bit-parallel layout;
buffer ← buffer of resulting reconstructed data;
cols ← number of columns in original relation;
tuples_per_word ← 32/cols;
for k ← 0 to 31 do
    for m ← 0 to 16 do
        next_word ←block[k ∗ 16 + m];
        for n ← 0 to tuples_per_word do
            kth_bit ←next_word>> (n ∗ cols)&1;
            buffer[m∗ tuples_per_word]+ =
              (kth_bit<< (k − 1);
        end
    end
end
```

Reconstruction. The first phase of the join query is reconstructing the data by extracting the individual bit values

from the bit-parallel layout and packing them into the original representation of a word. Essentially, every tuple must extract a bit from 32 different words. This requires an `AND` operation, a `SHIFT` and an `ADD` operation for each word from which we extract a bit. The straightforward implementation is shown in algorithm 2. We observe reconstruction is very expensive, with each value to reconstruct requiring 64 shifts, 32 bit-wise and operations, and 32 additions. Here it is in essence not feasible to exploit the bit-parallel layout, meaning each bit of interest must be extracted with operations on full words. This is caused by the fact that all bits in the bit-parallel word correspond to the same bit value for the same position for the resulting reconstructed words, making it impossible for a single operation to aid in the reconstruction of more than one word without hardware support.

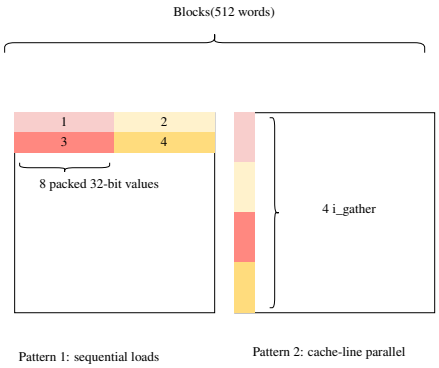

Fig. 2. *Illustrates two access patterns of the bit-parallel block, on the left is the sequential access pattern and on the right is the cache-line parallel access pattern, which is more optimal.*

Optimizing reconstruction. While the amount of operations that must be performed for reconstruction is unalterable, we can still exploit SIMD for performance gain. To minimize the overhead of the reconstruction we apply two optimization techniques: vectorization and a more efficient access pattern. Figure 2 illustrates the two variants of access patterns. The first loads the SIMD packed 8 32-bit values sequentially on the layout and performs the necessary operations as in algorithm 3. However, this has two major disadvantages. Firstly, for each word processed, we load and store the intermediate values into the results buffer, causing an additional 32 loads and stores. Secondly, the indices of the tuples are not preserved in the results buffer, which causes an explosion in runtime at high selectivities (observe figure 3, on the right for the cost). Both these factors can be avoided using the second access pattern, shown on the right in figure 2. In this scenario, we load one word from all 32 cache lines into 4 packed 256-bit vectors. Since all bit-values in a tuple are contained by these 4 vectors, we require no loads from the buffer and only a single store once reconstructed. This comes at a trade-off of a more expensive load operation, meaning we use 4 `_mm256_i32gather_epi32` instructions to fetch the bit-parallel data, and we require a horizontal accumulation of the final values. The number of loads from the bit-parallel layout stays the same, however, each of the loads has a much lower throughput, with a standard load instruction of 0.25 *cycles per instruction* (CPI) and `i_gather` a CPI of 10 [5]. This, however, is a worthwhile trade-off, since loading from the bit-parallel layout is *not* on the critical path of instruction dependencies, while the intermediate loads and stores from the buffer are.
Furthermore, both have the same cache-miss rate, as a single block, and the buffer for the reconstructed set, meaning the working set of the reconstruction, easily fits into the 32-KB L1 cache. This means both access patterns only cause compulsory misses.

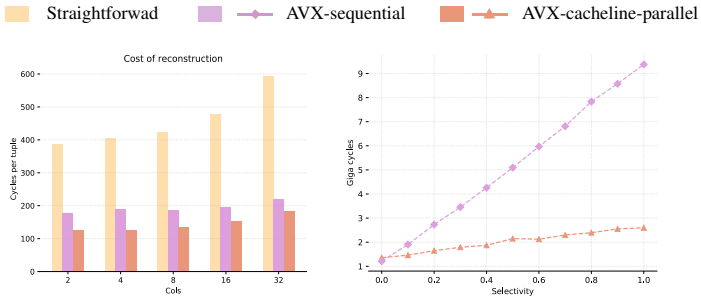

Fig. 3. *Benchmarks of reconstruction phase, the left diagram shows the cycles per tuple and the right diagram shows the cost of rearranging the indices of the original database. Selectivity is defined as the size of the output relation over the Cartesian product: $|R| * |S|$. A selectivity of 1.0 would represent a query, in which every tuple pair fulfill the join condition.*

Reflecting our analysis previously discussed, figure 3 shows benchmarks for the reconstruction phase of the join query. We gain around 3x speedup by applying SIMD, with a further roughly 25% speedup from the cache-line parallel access pattern. The figure also shows how the number of columns affects the performance. The lower the number of columns, the more densely packed the values are inside our layout, meaning we need fewer loads per tuple we wish to reconstruct. In conclusion, even with all optimizations above, we require 100 - 200 cycles per tuple to reconstruct the data. This is a far cry from the memory barrier, which lies around 8 cycles per tuple reading directly from DRAM in the least densely packed version (i.e 32 columns).

Modulo strength reduction. As the join query in this section is already strongly compute-bound, most gains in performance can be achieved by focusing on the efficiency

of evaluating the join condition. The modulo operation is inherently expensive, with the *DIV* instruction having a gap of 9-11 cycles [6]. Without any form of early pruning, $|R| * |S|$ modulo computations must be made, a major bottleneck in the query. Furthermore, neither the integer division nor the modulo operation is supported by the AVX2 library, making it difficult to exploit SIMD for any performance gain.
Our method to improve performance is to replace the modulo operations with less expensive and easily vectorized operations. This approach is inspired by fast unsigned integer division introduced [7, 8, 9] and further extended to our use case of the modulo join.

Algorithm 3: *Modulo strength reduction*

Result: R.a % S.b == S.c
Find K,L such that $\frac{R.a}{S.b} = \frac{(R.a*K)}{2^L}$;
Q := $\frac{R.a}{S.b}$ = R.a * K >> L;
return R.a - Q * S.b == S.c;

Cost analysis of strength reduction. Algorithm 3 shows how we forego computing a modulo operation. However, computing K and L is an expensive operation: for the non-trivial case of S.b > 2 we require exactly 4 additions/ subtractions, 1 log base two operation (BSRL instruction), 2 shifts and 1 division. Notice, this is considerably more expensive than a single modulo operation (implemented by a single DIV instruction). In a nutshell, the key insight is that finding the values K and L can be done independently of the second operand, in our case R.a. From this, we deduce that relation S should *always* be treated as the outer relation of our query, regardless of $|R|$ and $|S|$. This approach reduces the amount of division/modulo operations that are made throughout the execution from $(|R|*|S|)$ to $|S|$, meaning the expensive portion of algorithm 3 can be amortized.

Evaluating the join condition with K and L. Once values K and L are computed, we can evaluate the join condition while foregoing any expensive `DIV` instruction. In theory, this can be done using a subtraction, 2 multiplications, a shift, and a comparison instruction. Having said that, we do not account for possible overflows that may occur during the multiplication of $R.a \times K$. The possibility of overflow incurs additional complexity and thus additional cost in the critical code section. Refer to figure 4 for the vectorized join condition implementation.

Taking into account the instruction latencies on the Intel Haswell microarchitecture [10, 5], a single step of a loop iteration, shown in figure 4, has an overall latency of 22 cycles, taking into account instruction dependencies. with the `mullo_epi32` contributing the most with a 10 cycle latency. Furthermore, inter-loop step execution of these operations can be pipelined, giving us better *Instruction level parallelism* (ILP) than just using a single DIV instruction. Theoretically, a strict lower bound for the gap for a single

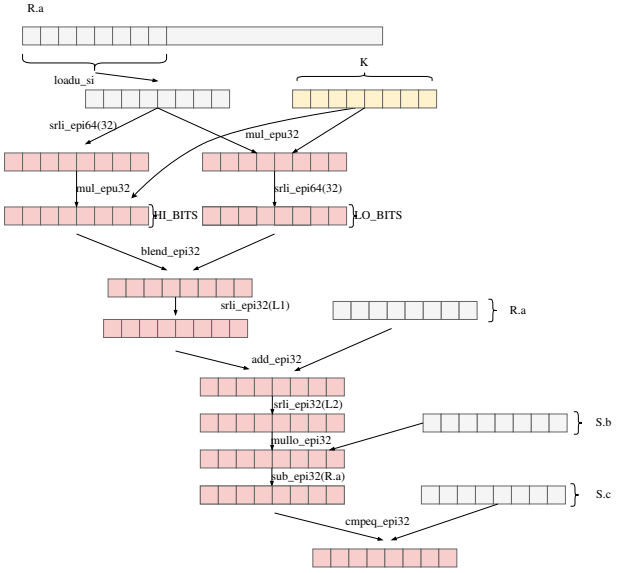

Fig. 4. *AVX2 implementation of join condition evaluation with precomputed K and L values for modulo strength reduction. Both the multiplication steps are split into two to deal with possible overflows.*

loop iteration may reach 2 cycles, limited by the `mullo_epi32` instruction. In conclusion, we can pack 8 modulo operations into a latency of 22 cycles instead of performing 1 modulo operation with a gap of 9-11 cycles and latency of 22-25 cycles, according to Agner Fog [6].

Blocking. We designed a block-nested loop join implementation to increase cache hit rate and reduce the number of reconstructions per tuple, additionally taking into account the fact that reconstructing data is more expensive than loading data, as shown by figure 3. The initial approach follows intuitively from the layout and joins block-by-block with values encoded by 512-bit values in the bit-parallel layout. Assume we have two relations R, S, with L1-cache size γ, such that $|R| >> \gamma$ and $|S| >> \gamma$. R and S are encoded into bit-parallel block such that $tuples\_per\_block := 512/\#cols$. Block nested loop joins better exploit cache-locality, minimizing the number of capacity misses. Cache analysis concludes we reduce the miss rate by a factor 512 (loading 32 cache-lines for each value we access) in the loading of relation S and maintain compulsory misses in relation R. A strict upper bound for working set for the block nested loop join $< 512*7*4 = 14336B$, which is less than γ, taking into account both blocks of R and S and the intermediate buffers of reconstructed data $R.a$, $S.b$, $S.c$ and buffers for strength reduction values K and L. Moreover, the number of reconstructions is also reduced from $|S| + |R| * |S|$ to $|S| + (|R| * |S|)/(tuples\_per\_block)$.

Experimentation with additional levels of blocking shows negligible performance gains, except for adding another block-

ing level to the reconstruction of the outer relation $|S|$. The second blocking level increases the number of bit-parallel blocks reconstructed, before subsequently reconstructing the values in the inner relation R and evaluating the join condition. Let B be the total number of bit-parallel blocks for the table S and b the number of bit-parallel blocks computed inside the second level of blocking. The second blocking layer reduces the number of reconstructions by an additional factor b. Figure 5 illustrates benchmarking for different block sizes. Intuitively, this means reconstructing (and computing K and L for) *all* values in S before computing the join condition is most efficient, however, we deduce that there is a saturation point in speedup at a block size b at around 8 for any row/column combination. While $b \lim |B|$ will increasingly reduce the number of reconstructions, it requires high memory overhead, which may be an issue for large in-memory databases. This motivates our choice to set $b := 8$ for the final version of query 3.

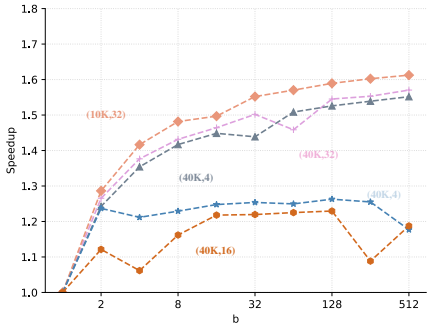

Fig. 5. *Illustrates the speedup from different blocksizes b. Each line represents different relation size of R and S, denoted by the pair (#rows,#cols)*

Bitwise Modulo.

While bit-by-bit modulo computation is not used in any optimization steps, we discuss our attempt at implementing an efficient algorithm and the major performance issues thereof.

To achieve a recursive bitwise algorithm, we reformulate the modulo condition:

$$a \mod b = c \Leftrightarrow \exists\lambda \in \mathbb{N}_0 : (a-c) - \lambda b = 0$$
$$\Leftrightarrow \exists\lambda \in \mathbb{N}_0 : \frac{(a-c)}{b} = \lambda \quad (1)$$

Instead of calculating the modulo, we try to answer if a λ fulfilling Eq 1 exists. This question can be answered (partially) by considering if b or $(a-c)$ contain the factor 2, which is indicated by the last bit. In the following we explain the four cases and how they help to answer the question or point to a sub-problem of the same form.

The factor 2 is in both. When factor 2 is in the numerator and denominator, we can pull out $\frac{1}{2}$ without changing the lambda that we are checking for. $\exists\lambda \in \mathbb{N}_0 : \frac{2(a-c)}{2b} = \lambda$

The factor 2 is only in the denominator.

$\nexists\lambda \in \mathbb{N}_0 : \frac{(a-c)}{b} = \frac{\prod_{i\in\mathbb{S}_1\subset\mathbb{P}\setminus 2} i}{2\prod_{i\in\mathbb{S}_1\subset\mathbb{P}} i} = \lambda$

In this case $\lambda \notin \mathbb{N}_0$ and we can definitely answer the where-clause with false.

The factor 2 is only in the numerator.

$\exists\lambda \in \mathbb{N}_0 : \frac{(a-c)}{b} = \lambda = 2\lambda' \Leftrightarrow \exists\lambda' \in \mathbb{N}_0 : \frac{(a-c)}{2b} = \lambda'$

In this case we can divide both λ and $(a-c)$ by two and look for the new $\lambda' \in \mathbb{N}_0$.

Last case. The factor 2 is neither in the denominator nor in the numerator.

$$\text{Eq. (1)} \Leftrightarrow \exists\lambda' \in \mathbb{N}_0 : \frac{(a-c)-b}{b} = \lambda - 1 = \lambda' \quad (2)$$

This case can still be reformulated as the new search for a new λ', but it has its costs in a bit-wise setting.

Problems with the last case. While the first three cases can easily be handled, by dropping or keeping the last bit, the last case with the factor 2 in neither the numerator nor denominator, we need to do a subtraction on the left-hand side of Equation 2 in future iterations, before the case selection can be done in those iterations. Doing these subtractions on the fly while loading new bits of a and c, requires a lot of bookkeeping for the carried-over bits, the exact number still to subtract as well as shifts and integer additions. In this regard, this approach is limited by the number of registers and the Intel instruction set. Furthermore, the pattern in which the troublesome case without a factor 2 appears, depends on the trailing bits of a and c. Therefore, there is no straightforward way to group similar cases or reasonable assumptions about the ordering of the data in the database that can bring speedups.

The unpredictable branching of the recursive loop over the cases together with the relatively high number of registered needed for the bookkeeping, bar this algorithm from making use of SIMD. This leaves this approach far behind the alternative modulo power reduction, which can make use of fast reconstruction and amortize its calculation over larger datasets.

6. EXPERIMENTAL RESULTS

Experimental setup. The different queries are individually tested and optimized on different processors with turbo boost disabled. Query 1 is run on an Intel Kaby Lake i7-7700K, query 2 on an Intel i5-8265U Coffee Lake, and query 3 on an Intel Haswell i7-4770K@3.5GHz. Memory boundary for query 1 was measured using IntelMemoryChecker as 9.14 Bytes/cycle and as 10.45 Bytes/cycle for query 2. The compiler used for the benchmarks is GCC and all "-oX" optimization flags are tested, and as -o3 performs the best consistently, all measurements are taken with that compiler flag set. We test both cold and warm cache scenarios,

to see how close to a memory independent lower bound we can optimize.

Results Query 1. The naive version of this query is compute-bound, even in a cold-cache scenario. At 32 features per sample, the most naive implementation takes ∼490 cycles per sample. The loop restructuring, indexing improvements, and related scalar replacement reduced this to ∼400 cycles per sample. Loop unrolling further improved this statistic to ∼130 cycles per sample.

Parallelized over 64-bit words, again with 32 features, unrolling and scalar replacement optimizations brought no speed up to computation (clearly well optimized by the compiler), though brought a 1.6x speedup to result readout. At this point we are still strictly computation bound, as cold cache and warm cache produce the same performance. The two different readout methods (shift mask or shift state) produce no difference in performance.

The computation over one 64 bit word takes ∼110 cycles, so ∼3.5 cycles to compute each inner loop. Going by instruction latencies and dependencies within the loop, we would expect at least 5.5 cycles (without loading!), so we are making good use of our CPU's available ports, and pipelining effectively.

Moving on to the vector implementation. We now reach a computation speed where we are limited by memory bandwidth in a cold cache scenario. In table 1, we can see the effect of different unrolling factors on computational speed given data in different cache levels. In figure 6, we show a comparison of our implementations' runtimes to the memory bandwidth lower-bound. Before the L3 boundary, a warm cache scenario, we are compute-bound. After the boundary, we are reading from memory, and memory bound.

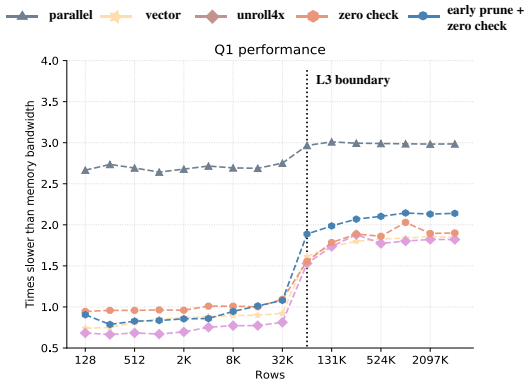

Fig. 6. *Comparison of Q1 to memory bandwidth, 32 features*

At 32 features per sample, we are now computing 8 samples in parallel, and with a warm cache need just 12 cycles per sample. This further decreases to just 3 cycles per sample in the 4 feature case. When unrolled 4x, computation over one 256 bit vector takes ∼92 cycles (∼2.9 cycles per inner loop), in comparison to the parallel implementation's 110 cycles, while computing 4x as many samples. The lower bound of cycles per inner loop theoretically is: shift to align b, then xor, which takes two cycles. Following this, we can compute Res and $Temp$ in parallel, where Res needs 3 instructions. Thus the lower bound is 5 cycles without considering loading. We have three ports that can do logical vector operations and just one which can do shifts. We need one shift per vector, so this doesn't present a significant bottleneck. The rest of the operations are pipelined well enough that we almost halve the theoretical lower-bound, which would be optimal given the third port, and have amortized the loads away.

Unroll factor	L1	L2	L3	Memory
1	1	1	1	1
2	0.87	1	0.99	0.75
4	0.81	0.88	0.74	0.71
8	0.76	0.73	0.74	0.75

Table 1. *Relative compute cycles for data in different cache levels*

Before applying "-o3" compilation flags, the zero-check, and early pruning both handily outperform the simpler implementation. With -o3 though, the picture changes. Given all-zero features in the L1 cache, we need ∼72 cycles per vector in computation when using optimally unrolled all-zero tests. On data where the top 16 bits are guaranteed to be 0 (rest random), we reach ∼104 cycles per vector, still being outperformed by the regular implementation. We only break even once the top 22 bits are all zero. The best granularity found is branching when both vectors of a cache line are all 0. First computing the xor of each sample, and then testing that, is slower than just an all-zero test in all cases.

For early pruning on random data with a cold cache, we are a bit under twice as slow as the standard implementation (large variance given input). We break even if the decision is made by the 12th bit, so if numbers are on average larger than 2^{19}. In a warm cache scenario, it performs as good or up to 40% slower than the standard implementation (similar input variance). An inner cycle where we do not prune is ∼7.5 cycles, up from ∼2.9.

When combining early pruning and all zero check, we are outperformed by the all zero-check in all cases. Similarly, unrolling the result readout only worsens performance in both cold and warm cache scenarios by about 13%. Reading out the result using vector intrinsics and a mask per sample within, is also always slower than the basic implementation.

A comparison of our implementation to that which provided the ground truth (a simple C implementation, reading an array, comparing, and setting result 1 or 0 accordingly), can be seen in Table 2. While the parallelism initially helps outperform the simple implementation, once we reach 32

Features	Naive C	Optimal Weaving Vector
4	14	5
8	16	9
16	20	17
32	30	33
64	36	61

Table 2. *Number of cycles per sample in a cold cache Q1*

features, the naive implementation outperforms us. Given a warm cache, the naive implementation vastly outperforms, highlighting the optimal memory use of the layout. From our most naive to our most optimal, we achieved a speedup factor of 40.

Results Query 2.

Our naive version, utilizing only some simplifications in index calculation, runs for 820 cycles per sample (a sample having 32 features). From here, scalar replacement only made a marginal difference in our experiments. On the other hand, loop unrolling brought a big improvement, bringing runtime down to around 210 cycles per sample.

Interestingly, vectorizing the final summation/accumulation did not yield a consistent speedup, especially when the number of samples was large. This implies that the computational bottleneck largely lies within the creation of the result array and reconstruction of the values to be summed.

Not surprisingly, full vectorization makes for another leap in performance, bringing runtime down to around 32 cycles per sample.

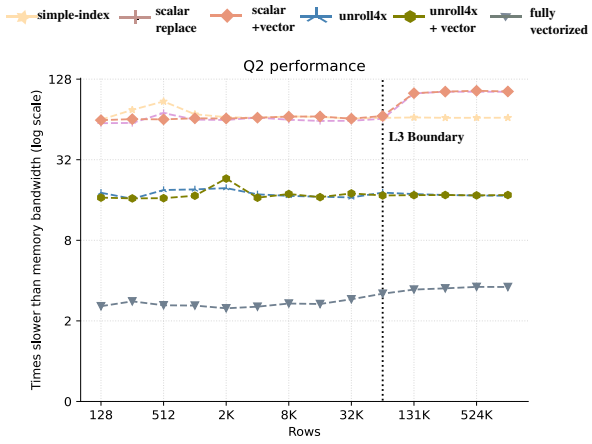

Fig. 7.

Results Query 3.

Figure 8 shows the runtime and performance metric of all optimization steps of query 3. Performance is measured by cycles per join condition computation. From the naive implementation to the final version of query 3 we see an overall speedup of 35x per pair, reducing the cycle count from ∼700 to ∼20. The largest initial increase, around one order of magnitude, is produced by 1-level block nested loop join. We see the AVX implementation and strength reduction further reduces the cycle by half. Profiling sections of the code further displays that the percentage of time spent on reconstruction is around $\sim 23\%$ for 1-level block nested loop join. The final implementation heavily reduces this percentage to $< 1\%$, meaning the bit-parallel nature of the layout incurs non-significant overhead for query 3. This, however, is in part due to the sheer cost of a modulo join and may be a much larger percentage for less expensive join conditions. To conclude the evaluation, the major bottleneck remains evaluating the modulo join condition.

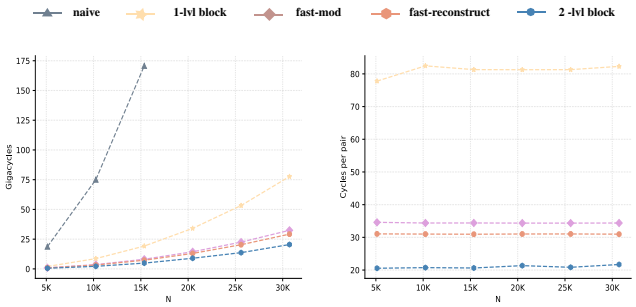

Fig. 8. *Left figure shows runtime and right figure shows cycles for each pair of tuples, with respect to the table sizes. Both tables have 32 columns.*

7. CONCLUSION

We achieve an overall 35 - 40x speedup for each query when compared to their respective straightforward implementations. We reach speeds where early pruning often causes more overhead than gain, losing one purported benefit of the layout. Allowing for a more densely packed representation for a smaller number of columns in our layout may also be considered a slight improvement over MLWeaving, further increasing the utilization of word-width.

We presented optimized implementations of a table scan, an aggregation, and a join query, pushing the performance boundary for a bit-parallel database layout to the limit. We hope these implementations provide a starting point for achieving general-purpose bit-parallel in-memory databases, optimizing for fast table scans and machine learning applications, while incurring minimal overhead for more complex relational queries. We believe these boundaries can be pushed even further in the case of join queries by exploring queries with join conditions that allow for bit-by-bit evaluation. Furthermore, an optimized FPGA could bring speeds that justify the coding effort, and further exploit word-width utilization where software implementations cannot.

8. CONTRIBUTIONS OF TEAM MEMBERS

Sebastian: Bit-wise modulo

Dominik: Query 1, layout layout converter

Micheal: Query 2

Matt: All optimizations regarding query 3 except for bit-wise modulo, all straightforward implementations

9. REFERENCES

[1] Y. Li and J. M. Patel, "Bitweaving: Fast scans for main memory data processing," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), p. 289–300, Association for Computing Machinery, 2013.

[2] L. Lamport, "Multiple byte processing with full-word instructions," *Commun. ACM*, vol. 18, p. 471–475, Aug. 1975.

[3] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang, "Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning (technical report)," *CoRR*, vol. abs/1903.03404, 2019.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 205–218, 2006.

[5] Intel, "Intel intrinsics guide." https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed: 2021-17-05.

[6] A. Fog *et al.*, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.

[7] A. Fog *et al.*, "Vector class." https://github.com/vectorclass/version2. Accessed: 2021-20-04.

[8] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," *SIGPLAN Not.*, vol. 29, p. 61–72, June 1994.

[9] J. Sheldon, W. Lee, B. Greenwald, and S. Amarasinghe, "Strength reduction of integer division and modulo operations," 08 2001.

[10] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.