technische universität
dortmund

Master's Thesis

Efficient Scans on Modern Hardware

Benjamin Kramer
September 21, 2015

Adviser:
Prof. Dr. rer. nat. Jens Teubner
Dr.-Ing. Christian Mathis

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Datenbanken und
Informationssysteme

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 21. September 2015

Benjamin Kramer

Abstract

In recent years, hardware characteristics, software systems, and application patterns have all changed in a way that emphasizes the importance of scan tasks in modern main-memory database systems. Main memory is available cheaply and can hold significant amounts of data nowadays. In-memory computing has matured and has resulted in key business platforms such as SAP HANA. Expectations for ad-hoc querying defeat the use of pre-built index structures and mandate scans instead.

At the same time, it is still unclear how the latest advances on the hardware side can best be leveraged to support efficient scans. Growing SIMD widths, for instance, appeal with theoretical speed-ups of up 16× or more. But existing strategies for predicate evaluation and scans are not prepared for such large SIMD widths. In fact, even for smaller SIMD sizes it is not clear how, e. g., predicates that involve multiple data types and widths can be realized most efficiently.

This thesis studies and evaluates methods that make use of modern hardware technologies to accelerate scans, in particular *SIMD-Scan*, *BitWeaving* and *ByteSlice*. Starting from *SIMD-Scan*, methods to further reduce memory bandwidth are explored by skipping parts of the input efficiently or by reducing the output size by handling intermediate results more efficiently.

In den vergangenen Jahren haben sich Hard- und Software sowie die Anforderungen an die Software so geändert, dass der Scan – das durchsuchen einer Spalte in einer Datenbank – in In-Memory-Datenbanken immer wichtiger wird. Hauptspeicher ist günstig und kann heute große Mengen an Daten halten. In-Memory-Datenbanken sind verbreitet und ausgereift und werden an Schlüsselstellen in der Wirtschaft verwendet, wie etwa SAP HANA. Die Anforderungen an diese Datenbanken können oft nicht durch vorberechnete Indices erfüllt werden und benötigen anstelle dessen Scans.

Gleichwohl ist es immer noch unklar wie man die neusten Errungenschaften in der Hardware am besten für diese Scans nutzen kann. Breitere SIMD-Register zum Beispiel, könnten theoretisch eine Beschleunigung um das 16-fache erreichen. Aber bestehende Scan-Methoden sind nicht auf solch breite Register vorbereitet. Selbst für kleinere SIMD-Breiten ist es nicht klar wie Prädikate mit mehreren Datentypen am effizientesten realisiert werden können.

Diese Masterarbeit beschäftigt sich mit Methoden, die diese moderne Hardware ausnutzen um Scans zu Beschleunigung, mit einem Fokus auf *SIMD-Scan*, *BitWeaving* und *ByteSlice*. Vom Startpunkt des *SIMD-Scan* werden weitere Möglichkeiten untersucht um die benötigte Speicherbandbreite zu reduzieren, zum Beispiel durch Überspringen von Teilen der Eingabe oder durch Reduzierung der Ausgabemenge durch effizientere Verwaltung von Zwischenergebnissen.

Contents

1 Introduction

The scan is one of the most basic primitives for building a database system. It searches over a column in the database and checks for every entry in the column whether it satisfies a predicate. The result is usually stored in a bit vector of the same size as the column, indicating the status of the predicate for every value, or a list of indexes.

A scan is not a computationally intensive task, but involves a lot of data so its performance is directly related to how fast the data can be loaded. Historically this meant that the scan runtime was equivalent to the speed of the available non-volatile memory, typically hard drives with magnetic rotating disks. In more recent time the storage for the data has shifted from hard drives to main memory as it became cheaper. Main memory is orders of magnitudes faster than hard drives, so the processor became the bottleneck. However, processors also became better – at a much faster pace than main memory – so the processor gave its bottleneck position to main memory.

This thesis focuses on that bottleneck with respect to fast column cans and explores ways to reduce the required bandwidth from main memory without shifting the bottleneck back to the processor.

1.1 Motivation and Goals

The main motivation of this thesis is to explore and evaluate the use of modern hardware technologies to accelerate scans. It builds upon existing research results to reduce memory bandwidth such as *BitWeaving* [1] and *SIMD-Scan* [2] and evaluate their effect on modern hardware platforms and for different workloads.

The goal is to develop strategies to optimize the scan operation to leverage the full capabilities of modern hardware. This includes identifying the critical bottlenecks of existing scanning approaches and coming up with strategies to reduce the load at those critical points.

1.2 Thesis Structure

Following this introduction chapter, Chapter 2 discusses the theoretical background of the problems faced with fast scan algorithm and introduces several improved scan approaches. Chapter 3 tries to quantify the improvements and shows in which cases they work and in which cases they will not. Details on the implementation of the scan algorithms are laid down in Chapter 4 which is directly followed by measurements and

analysis of the results in Chapter 5. The concluding Chapter 6 summarizes the achieved work and benchmark results and points out directions for the future.

2 Background

This chapter gives an overview on the theoretical background of scans and what is needed to make them efficient on modern hardware. Then some possible solutions are introduced and the approaches that are evaluated later in this thesis are described in detail.

2.1 Memory

Since main memory has become large and cheap enough to store very large data sets, many database systems no longer bother reading data from slower but cheaper storage such as hard drives, but keep everything in main memory and perform all queries directly there. This lead to database systems that are much faster than the previous generation of database systems. However, there are performance limitations to this approach: it is a well-known fact that processor performance is growing much faster than memory performance is growing, as illustrated in Figure 2.1.

This performance discrepancy has been growing over time and is unlikely to change in the near future. The effect is mainly due to the used type of memory, so-called dynamic RAM (DRAM), which is extremely dense in terms of on-chip space and can thus be produced very cheaply. However, it relies on charging and discharging of capacitors, which is comparatively slow: reading a single bit from DRAM can take hundreds of processor cycles. Processors use faster static RAM in small caches which store data that was recently accessed. This principle of locality helps performance in programs working on small data sets but for the large data sets typically used in database systems it rarely makes a difference, because caches are too small to hold all of the data.

To overcome expensive one-bit reads, many bits are read from RAM in parallel, hiding the time needed to charge and discharge the capacitors. This essentially means that a block of memory is read in the same amount of time as a single bit is. To further improve the access times, the CPU detects sequential access of a block of memory and will instruct the memory controller to load the next blocks in advance. Because the next block is read in the background the latency is invisible to the user. This so-called prefetching strategy is the only way high bandwidth can be achieved with this kind of dynamic memory. Common methods to accelerate accesses in database systems, such as trees and hash tables, use a memory access pattern that cannot take advantage of prefetching, so a different approach is needed.

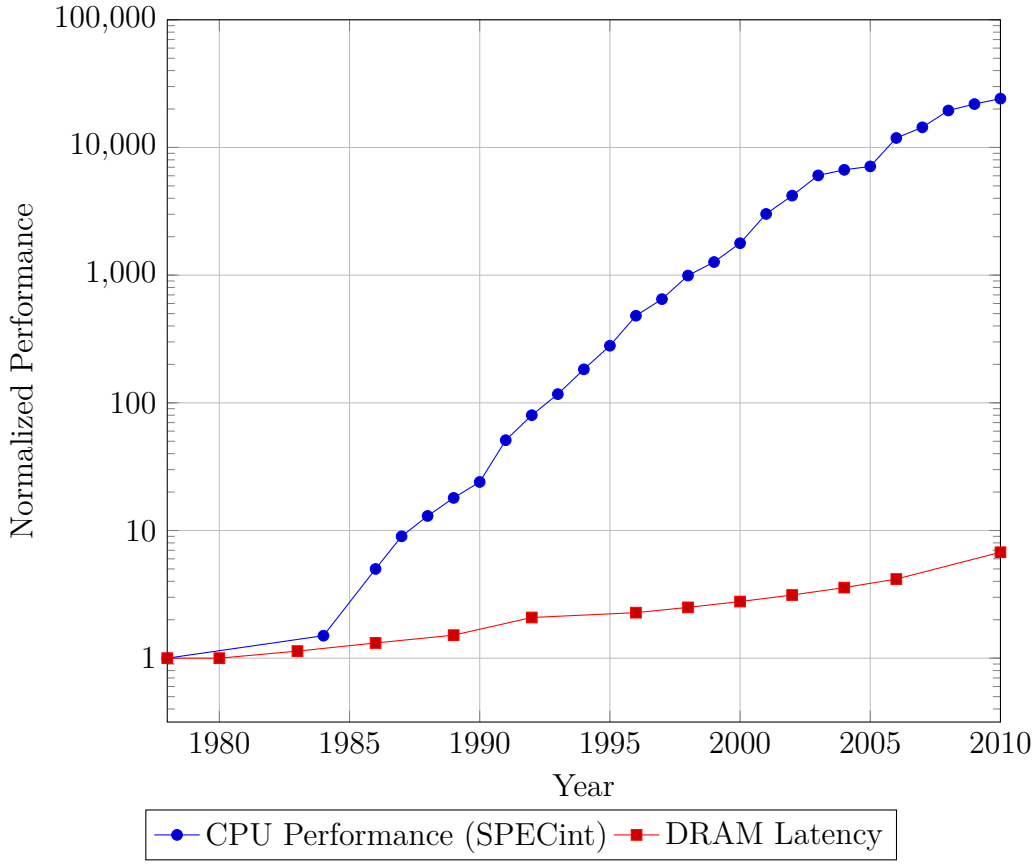

Figure 2.1: CPU and DRAM performance over time [3].

2.2 SIMD

The idea behind SIMD (Single Instruction/Multiple Data) architectures as first classified by Flynn [4], is to use a single instruction stream to process many different values at once. The earliest members of this class of architectures were the vector processors of the 1970's such as the Cray 1 or the Earth Simulator, the fastest supercomputer in the world in the year 2002 [3]. In a vector processor the size of a register is not static but the number of elements in a register can be configured at runtime up to a certain limit. The Cray 1 had a limit of 64 values in a register, each of the values 64 bits wide. When arithmetic instructions are executed with such a vector register, the individual values in the operands can be processed in parallel as far as resources permit. So processing a 64-element vector can take significant less than 64 cycles. Load operations typically gather values that can be scattered throughout memory into a single register and load operations have to disperse the values back into memory. By this approach a CPU with a vector instruction set can perform the same operations while executing dramatically fewer instructions than an architecture that operates on a single value at a time. This

Instruction	Latency	Throughput
Scalar load	(depends on cache)	2 per cycle
Scalar add	1 cycle	4 per cycle
Vector load	(depends on cache)	2 per cycle
Vector add	1 cycle	2 per cycle

Table 2.1: Instruction latencies and throughput on Intel Haswell [5]

frees up instruction decoding resources that can be repurposed to increase throughput elsewhere.

However, classic vector architectures have not become the most prevalent model for modern processor architectures, obvious reasons being:

- Superscalar out-of-order processors already provide parallelism by automatically identifying independent instructions and executing them simultaneously.
- Gathering loads and scattering stores are not a good fit for caches and memory organized in fixed-size blocks where at least a full block is fetched or stored. This would waste a significant amount of memory bandwidth and cache memory.
- The large variable-length registers add additional state to the processor. This state must be managed by the operating system, making the switching between processes more expensive.

In the end, a fairly limited subset of a vector processor's instruction set has found its way into the instructions set of virtually any commodity processor, except those that are geared towards low-performance embedded applications. These so-called multimedia instructions only support fixed-size registers which can usually be overlaid with different data types. For example registers containing 128 bits can be used as either four 32-bit values or eight 16-bit values. This means there are many different instructions compared to the approach taken by vector architectures. Scattering and gathering is usually not present or only in a limited form by loading values separated by a constant gap. Examples for this form of vector instructions are MMX, SSE and AVX on Intel CPUs, AltiVec on PowerPC or NEON on ARM processors.

To understand why SIMD still provides an advantage over superscalar instruction-level parallelism the latency and throughput of normal instructions has to be taken into account. In Table 2.1 the latencies and throughput for loads and additions are shown. Now imagine a loop creating a sum over 32-bit values; requiring a load and an add. All instructions are fully pipelined, so we can load two 32-bit integers and perform two additions per cycle, the full throughput is not achievable, because we cannot load more than two values. Now we change the loop to work on 256-bit vectors, the load throughput does not change with size, so we can load 16 32-bit integers per cycle and still perform 2 full adds, adding all 16 integers. In theory we now increased the throughput eightfold. In practice the performance will be bound by memory latency though, unless the number of arithmetic instructions is increased.

2.3 Scans

One of the most basic primitives to build a database system is the scan. For a given table in the database the scan compares every row against a given predicate and returns which rows match that predicate. This is equivalent to a linear search over a block of memory, the runtime is $\mathcal{O}(n)$, where n is the number of rows.

For a long time, scans were regarded as something to avoid in database systems. Index data structures were developed to accelerate accessing the database. For example, evaluating the predicate $a < x < b$ on a sorted table or binary tree can be done in two $\mathcal{O}(\log n)$ binary searches. On the first glance this should be a lot faster than the naïve $\mathcal{O}(n)$ scan, but a binary search has a number of downsides.

- Keeping sorted data available requires redundancy or expensive data movement during modification operations. The indexes also need to be maintained after every change to the underlying data.
- A binary search executes $\mathcal{O}(\log n)$ branch instructions, every single branch is dependent on the input data and not predictable. Mispredicted branches slow down the scan as the processor has to wait on the result before proceeding with the next value.
- Most importantly, binary searches introduce random access to the data instead of a sequential access pattern. When the data does not fit into cache, every access has to pay the full access time of the DRAM and cannot use the prefetching to hide memory latency described above.

The other major indexing technique, hashing, can avoid branching. It still requires redundancy and uses a random memory access pattern. It is obvious that neither binary search nor hashing can take advantage of fast sequential accesses for databases in DRAM on modern hardware. Of course the actual performance depends among other things on the data distribution and the percentage of values selected during the scan, so there are still use cases where an index is highly preferable.

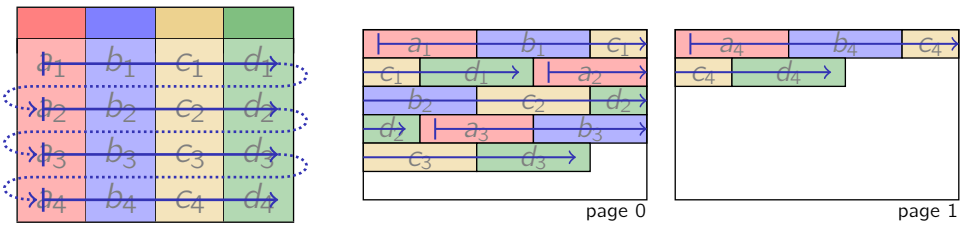

Figure 2.2: Row-wise storage [6]

Traditionally databases store their tables row-wise as seen in Figure 2.2. This means that a row is stored completely, then the next row is stored adjacent to the last row and

so on. When evaluating a predicate on this rows all unused columns have to be skipped. Since the skipped data is likely smaller than the minimum size of a data chunk that can be loaded from DRAM by the CPU (a cacheline) a scan will load a lot of data that is not used. But the far bigger issue with this layout is that it makes use of SIMD much harder, as SIMD works best with homogeneous data. On current processors it is unlikely that maximum performance can be achieved without using SIMD.

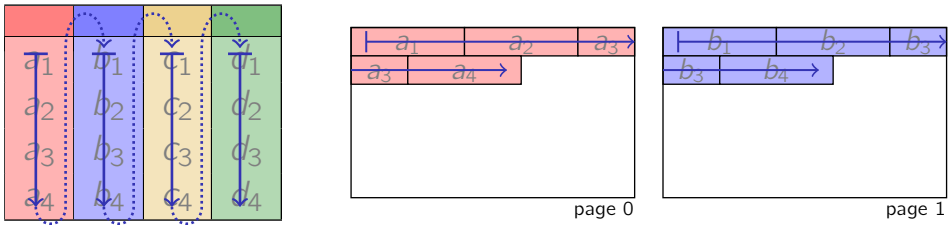

Figure 2.3: Column-wise storage [6]

The alternative to row-wise storage is to decompose the rows into individual columns and store them as shown in Figure 2.3 [7]. This has several advantages over the row-wise format.

- Less data is being fetched from memory during a scan. Columns not used in a scan can be ignored completely.
- SIMD is easily applied as all values in a column have the same format.
- More potential for compression as the elements in one column are often similar to each other.

The major downside of this format is fetching a row from a column store is more complex than fetching one from a row store. After the row has been selected, all columns have to be joined, this incurs at least a potentially random access for every column. The column store trades scan performance for a slower access of rows. Since the goal is to make scans as fast as possible this thesis will focus on column stores.

2.4 SIMD on Modern Hardware

The focus of this section lies on the current 4th generation Intel Core architecture. The basic observations are also applicable to any other current architecture or SIMD instruction set, but the performance characteristics may differ widely.

Intel has a long history supplying multimedia instruction sets:

MMX The Multi Media eXtension made the start in 1997, supplying integer instructions on 64 bit registers shared with the existing floating point unit.

SSE The Streaming SIMD Extension added a fully independent 128-bit register set in 1999. SSE only supported floating point operations, SSE 2 added the integer counterparts in 2001. SSE was extended with many different instructions often catered towards operations commonly found in video codecs.

AVX The Advanced Vector eXtensions extend the registers introduced in SSE to 256 bits. As with SSE, AVX provided only floating point operations and AVX 2 added integer instructions. AVX also provides new encodings for all existing SSE instructions using a three-operand form instead of the more restricted two-operand form used by SSE. AVX instructions generally behave as if working on two independent 128-bit registers instead of one large 256-bit register, this makes operations that combine bits from the lower 128 bits with bits from the upper 128 bits of the register more complex.

AVX-512 AVX-512 is a new instruction set extending the AVX registers to 512 bits. While it contains interesting new operations that could be very useful for implementing scans it was not available in general purpose CPUs at the time this thesis was written.

Since AVX 2 is currently the latest available set of SIMD instructions on Intel hardware it will be the focus of this thesis. It provides 16 256-bit registers and integer operations on 32 8-bit, 16 16-bit, 8 32-bit or 4 64-bit values at a time.

As a preliminary it is vital to understand how caches are organized on a current Intel CPU. Since for a scan data will almost always be loaded freshly from DRAM, cache levels are ignored here. The cache on any recent Intel CPU is partitioned into cache lines of 64 bytes. This means that when reading data from memory it does not matter if 1 byte or 32 bytes – which is the maximum load size in AVX 2 – are read at a time and the whole cache line will be fetched from DRAM.

A scan over an array of integers of size 8, 16, 32 or 64 bits can be easily implemented in SSE or AVX as illustrated in Figure 2.4. Here a greater than predicate is executed, the `pcmpgt` instruction takes two vectors and sets the corresponding element in a result vector to either all ones if the element in the first vector is greater than the element in the second vector. Otherwise it is set to zero. To generate a bit vector from this result the `movmsk` instruction is used, which takes the most significant bit of every element in a vector and sets the bit with the same number in a simple integer. In the example the result of `movmsk` is 00111101 in binary or $3d$ in hexadecimal.

2.5 *SIMD-Scan*

SIMD-Scan [2] takes the basic idea shown in Figure 2.4 and extends it to arbitrary bit widths. The methodology is comparable to a normal compression format where data is first decompressed in one step and then processed in another. The compression format stores a set of fixed-size integers without any padding. For decompression those individual integers have to distributed over the lanes of a SIMD vector. The main achievement of

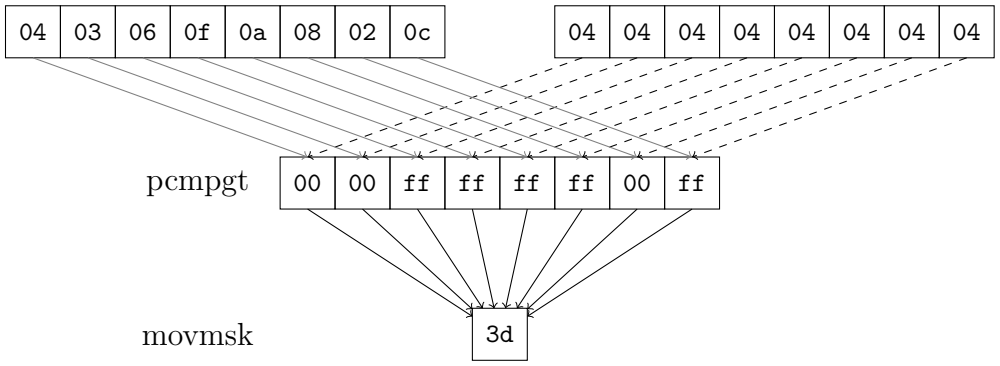

Figure 2.4: Comparing two vectors and generating a bit vector in SSE

SIMD-Scan is an efficient decompression method for all of the 32 bit cases from one bit to 32 bits. Due to limitations in Intel's vector extensions, the performance of the bit case can vary though, with cases 8, 16 and 32 bit being trivial on SSE hardware as decompression is equivalent to a simple copy here. Other bit widths require varying amounts of arithmetic and logic operations to decompress. In practical implementations of *SIMD-Scan* such as in the in-memory database SAP HANA every bit case is implemented manually to get an optimal sequence of instructions in a tight loop.

The basic steps performed when evaluating a range predicate $\min < x < \max$ with *SIMD-Scan* are shown in Figure 2.5. The example uses a bit width of 20 bits per word and a 256-bit vector register holding eight 32-bit values. In a first step eight values (160 bits) are loaded from the table and permuted over the eight vector lanes. Because the permutation operation only works on byte boundaries the result of this step is not aligned on the 32-bit boundaries. The next step would now be to shift the individual values to the boundaries resulting in an aligned vector, but here a different approach is taken. Since only comparisons are performed instead of shifting the values loaded from the table, the reference constants min and max are shifted before entering the main loop avoiding some work. Then the permuted values are compared to the permuted min and max constants, yielding a bit vector similar to the process shown in Figure 2.4. Because a range predicate is evaluated, the two comparison results are combined with an AND operation and turned into a bit vector using the `movemask` instruction. This bit vector result is then written into the output buffer.

SIMD-Scan has greatly benefited from new instruction set extensions such as AVX 2 [8] and is likely to benefit from AVX 512 too, as more general shuffle and compare instructions have been added, reducing the amount of instructions needed for some bit widths.

As will be shown in Chapter 5, the *SIMD-Scan* is bound by memory bandwidth alone for almost any bit width when using AVX 2 on a modern CPU. On Intel hardware this is a fairly new behavior that was not observed using plain SSE [8] and required a lot of manual optimization. This also means that *SIMD-Scan* already has the best possible

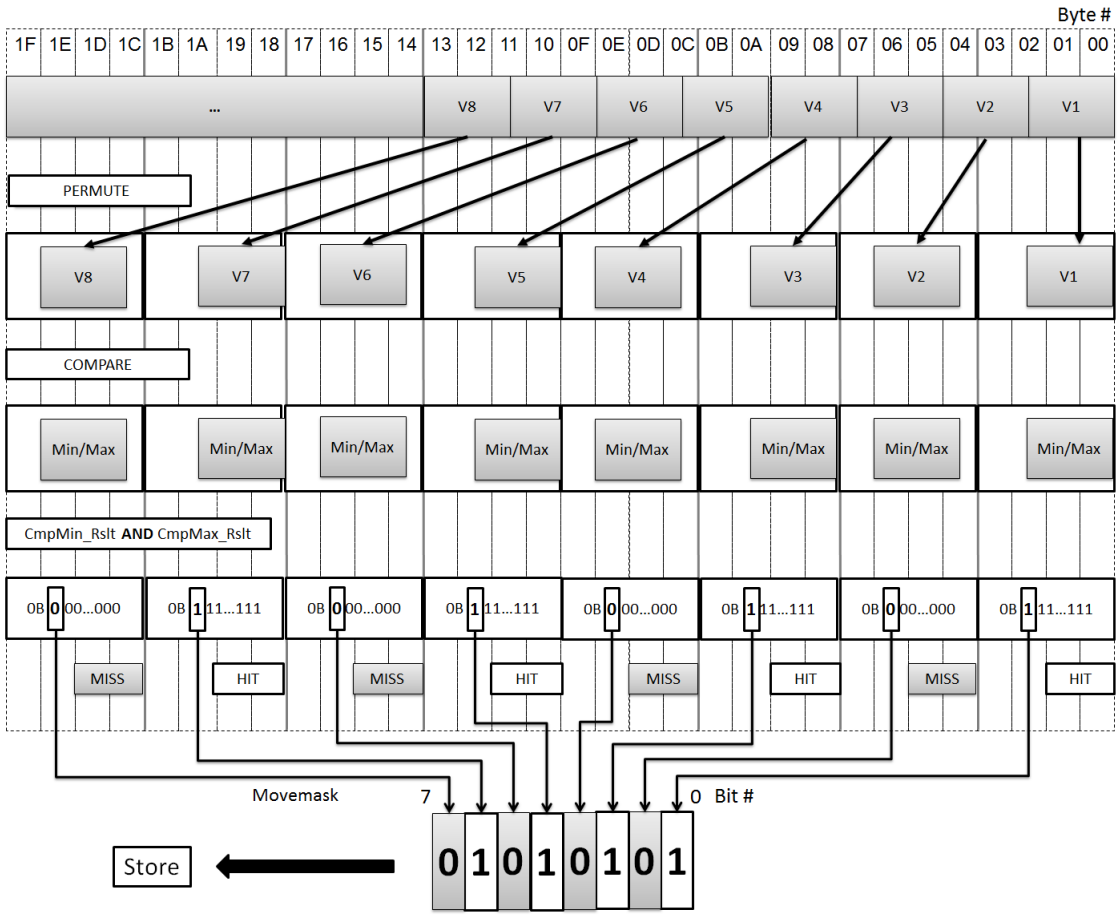

Figure 2.5: Vector comparison with *SIMD-Scan* [8]

performance when all bits of a column are read. There are three possible strategies to improve this.

- Using a more complex compression scheme such as PFOR [9] which tries to reduce the overall bit width by putting large numbers in a separate memory area and providing a reference to this table in the column. This still allows random access but requires a decoding step and trades memory bandwidth for performance when scanning data containing many large numbers.
- Using a variable-length encoding such as Simple-8 [10] or varint [11] which store significantly fewer bits if an entry in the column is small. This complicates random access and makes fully utilizing SIMD harder, because SIMD instructions are usually only provided for fixed-size integers.
- Trying to abort the scan before all bits are read. This is possible by modifying the storage format and performing the comparison of a word on smaller chunks than the full word size.

The following sections focus on exiting the scan early, because this is most likely to have the lowest overhead of the three methods.

2.6 *BitWeaving*

BitWeaving [1] is a new framework to further accelerate scans by rearranging the bits scanned in memory for better performance. It comes in two variants, a horizontal version, *BitWeaving/H*, and a vertical version, *BitWeaving/V*.

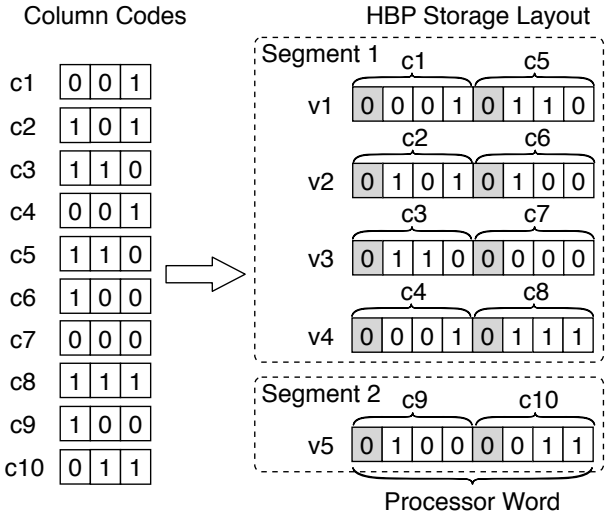

Figure 2.6: Storage layout of a horizontal bit packed format [1]

The horizontal form is based on "SIMD within a register" (SWAR) as first described by Leslie Lamport in 1975 [12]. The data is stored contiguously, similar to the format *SIMD-Scan* uses, except that a stop bit is added between every element, thus adding memory overhead. An example is shown in Figure 2.6. To perform a scan on this data it is not decompressed, but the arithmetic primitives used to do a comparison are decomposed and directly applied to the compressed data. This approach is different from *SIMD-Scan*.

As *BitWeaving/H* provides no significant advantage over *SIMD-Scan*, but adds additional memory overhead it was not taken into consideration. Knuth suggests in [12] that the additional bit can be avoided by adding more arithmetic operations, it is however unlikely that this will make *BitWeaving/H* faster than *SIMD-Scan*, the contrary is more likely. *BitWeaving/H* also has to read all the data for a scan.

Not reading all the data is the primary advantage of the other flavor of *BitWeaving*, *BitWeaving/V*. This format uses a vertical format first suggested in [13]. The individual bits of a word in the column are no longer stored together, but distributed over many words. In memory this means that the most significant bits of all words are stored first,

followed by the second-most significant bits and so on as seen in Figure 2.7. Scans are still implemented using a SWAR method on a bit-wise level. The difference is now that for a comparison operation the scan of a word is aborted after a mismatch occurs for the first time. Any additional bits do not have to be read anymore. This behavior is called early pruning or early exiting.

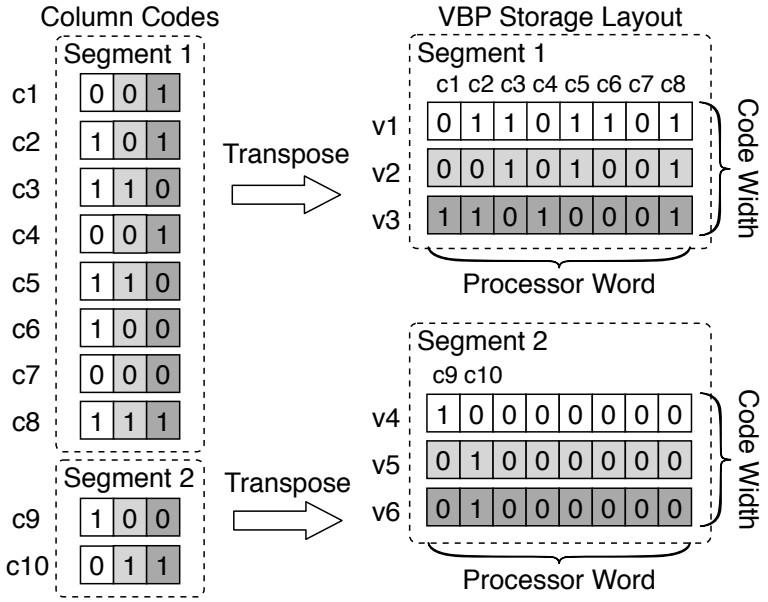

Figure 2.7: Storage layout of a vertical bit packed format [1]

To derive a performance advantage out of the memory that is not read anymore the column is partitioned into a fixed amount of groups. This can be imagined as first storing the four most significant of all words first, in the vertical format. This forms the first partition or group. Then the fifth to eighth significant bits are stored in the next group and so on. Early pruning is performed after a full group is processed, so the processor can take advantage of the sequential access without any interruption. This also reduces the likeliness of mispredicted branches as the total number of early pruning branches is reduced.

The major downside of this vertical format is the same that was seen when switching from row-wise to column-wise storage. Again, this is a trade-off between scan performance and the amount of work necessary to extract a row from the storage. As the bits of a single value are now distributed over many memory locations, reading it becomes a lot slower.

2.7 *ByteSlice*

ByteSlice [14] can be seen as a hybrid between vertical and horizontal bit packing formats. It is very similar to the *BitWeaving/V* format, but does not distribute individual bits over multiple locations. It only distributes bytes. Intel's SIMD extensions have very efficient instructions for byte-level comparisons and permutations, so this avoids complexity in the scan loop.

This byte-level vertical format has several advantages over the bit-level vertical format. Packing and unpacking *ByteSlice*d data only has to touch one eighth of the memory locations *BitWeaving/V* has to access. Furthermore, moving bytes in a vector register for packing and unpacking can be done using shuffle instructions in most SIMD instruction sets. Bit-level shuffling is usually not available. This allows very fast access of individual values without the high overhead in the *BitWeaving/V* format while still providing early pruning opportunities.

When comparing early pruning, it might seem like *BitWeaving/V* provides better pruning opportunities as aborting is possible after every bit in theory. *ByteSlice* can only prune after quantities of eight bits have been processed. This is only partially true though, because *ByteSlice* contains fewer values per vector word, so early pruning is more likely to occur, a single value in a vector word in either format can prevent early exiting. The grouping in *BitWeaving/V* also reduces the early pruning opportunities, but this can also be applied to *ByteSlice*. However, *ByteSlice* group sizes are always a multiple of eight.

The storage of bytes also comes at a cost of flexibility. *BitWeaving/V* can easily store any bit width with no memory lost to overhead bits. *ByteSlice* is limited to multiples of eight, all other bit widths add more complexity. One solution is to pad to the next multiple of eight with zeros. For example a column of eleven bit values is padded to 16-bit values, adding three bits of overhead as shown in Figure 2.8. In the worst case this can add up to seven bits of overhead per value. This is clearly a disadvantage when the scan is limited by memory bandwidth. Another solution would be to store as many bytes as possible and switch to the *BitWeaving/V* format for the remaining bits. For eleven bit values this would be one byte in *ByteSlice* and three bit in *BitWeaving/V*. This also has zero memory overhead, but increases the complexity of the scan by splitting it into two stages.

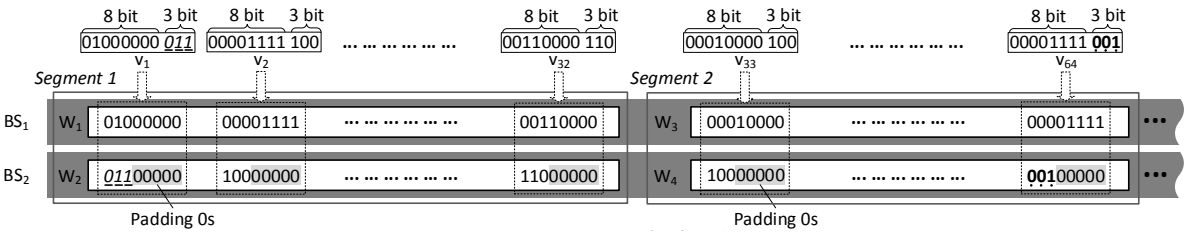

Figure 2.8: Storage layout eleven bit values in *ByteSlice* using zero padding [14]

2.8 Summary

In this chapter we have seen a basic overview of memory and SIMD instructions on modern hardware and methods to perform scans on in-memory column stores. *SIMD-Scan* is one method for scans that works very well on modern hardware, being only limited by the bandwidth from memory to the CPU core. *BitWeaving/V* and *ByteSlice* are two techniques that look very promising as they are likely just as bound by memory bandwidth as *SIMD-Scan* and their scans are not more complex in terms of instructions executed. However, these methods can extract additional advantages by using a more sophisticated memory layout to avoid reading all bits from memory and thus saving precious memory bandwidth.

BitWeaving/V and *ByteSlice* are very similar in their basic idea and behavior, but differ in details. This affects the number of branches executed, the probability of early pruning and the complexity of bringing data into the scan representation and vice-versa. This justifies a separate analysis of both approaches.

In the rest of the thesis we will evaluate how this early exiting performs when doing a scan both in theory and with practical benchmarks. In contrast to *SIMD-Scan* performance analysis now contains a probabilistic component depending on the input data.

3 Analysis

In this chapter the scan algorithms are examined with regard to possible reduction of the required memory bandwidth. This is done under the assumption that *SIMD-Scan*, *BitWeaving/V* and *ByteSlice* operate at or close to the bandwidth limitations of a modern processor. As such, *SIMD-Scan* represents the upper bound of memory bandwidth, because *SIMD-Scan* and *BitWeaving/V* use in-memory representations of the same size, just in different layouts. *ByteSlice* is comparable for all bit cases evenly divisible by eight. The early pruning behavior of *BitWeaving/V* and *ByteSlice* is analyzed for possible reductions in memory bandwidth. For all three approaches the impact of write size reduction by scanning a column in fixed-size blocks instead of doing a full column scan when evaluating a query over multiple columns is examined. The block-wise processing lowers the amount of temporary results materialized in memory.

3.1 Early Pruning Probabilities

If a scan algorithm is limited by the available memory bandwidth there is only one way to improve its performance further: reducing the amount of memory touched by it. As the input data for *BitWeaving/V* is already packed with no padding, the only way to reduce the number of bytes read during a scan without changing the input data is aborting the scan of a given code word early. The effectiveness of this approach relies on three factors:

- the predicate,
- the selectivity,
- and the distribution of the input data.

Compared to the *SIMD-Scan* approach, this adds a probabilistic dimension to the runtime analysis of a scan. The simplest predicate to analyze is the equality predicate. This predicate looks for values that are exactly the same as a reference. In *BitWeaving/V* this is solved by looking at a value from the column and the reference bit by bit, starting with the most significant, as shown for a single value in Algorithm 3.1.1. In reality *BitWeaving/V* compares many values in parallel.

Establishing the Best and Worst Cases for Early Pruning

It is now fairly simple to establish the best and worst cases for this scan. In the best case all values a from the column differ in the most significant bit with the reference

Algorithm 3.1.1 Algorithm to check whether two bit vectors of size n are equal

```
1: procedure IsEqual(a, b)
2:     for i ← n − 1 to 0 do
3:         if a_i ≠ b_i then
4:             return false
5:     return true
```

value b, for example, if b is negative, but all as are positive values. In this case only the most significant bit has to be read and all other bits are ignored. The loop only executes a single iteration and the selectivity is 0 %. In the worst case all values a are equal to b. In this case the loop will always execute n iterations and the selectivity is a 100 %. No memory reads can be skipped, early pruning is entirely defeated. However, if the overhead of the pruning scan is low enough this worst case will have the same performance characteristics as the *SIMD-Scan*, it reads exactly the same amount of data.

Now of course *BitWeaving/V* does not look at bits individually, but will start by reading a number of most significant bits at the same time and perform the comparison in parallel. For example an implementation targeting the 256-bit registers in Intel's AVX 2 instruction set extension will read bits from 256 values at the same time, perform the comparison and advance to the next 256 bits of the same values. This however means that if at least one of those 256 values is equal to the reference value, the worst case is automatically triggered for 256 values. For uniformly distributed data the expected value E for a selectivity S where early pruning still has an effect and not all bits must be read is as low as

$$E[S] = \frac{1}{256} \approx 0.4\,\%$$

In other words, the worst case can be achieved by placing one equal value in every 256 values.

For *ByteSlice* the worst case is always eight times better as fewer values are being processed in parallel. For an AVX 2 implementation only 32 eight-bit slices are compared at the same time, pushing the minimum selectivity needed for the worst case up to

$$E[S] = \frac{1}{32} \approx 3.1\,\%$$

Putting It on Real Hardware

On current server-class hardware *BitWeaving/V* can easily hit memory bandwidth limitations. This has an effect on the early pruning probabilities too, as memory is loaded in cache line-sized chunks. Only if an entire cache line can be skipped, a reduction in memory bandwidth (and thus scan performance) can be achieved. On the tested Intel hardware a cache line consists of 512 bits, meaning that the theoretical worst case is off by a factor of two. For a uniform distribution of equal values the worst case selectivity for *BitWeaving/V* becomes

$$E[S] = \frac{1}{512} \approx 0.2\,\%$$

as every cache line contains bits of 512 values. For *ByteSlice* which only uses bytes and stores bits of 64 value per cache line it goes down to

$$E[S] = \frac{1}{64} \approx 1.6\,\%$$

As long as the cache line size is larger than the register size, the register size has no effect on the early pruning probability. As soon as the computation is fast enough, such the scan is only limited by memory bandwidth, further increases or decreases of the register size will not have any effect on scan performance.

Calculating the Early Pruning Probability For A Given Bit

Now that we have established the worst and best cases we can look at the average cases. For uniformly distributed data where any bit can be set with a probability of 50 %, the probability that a given value is equal with the reference value up to bit b is $\left(\frac{1}{2}\right)^b$, so the full probability that we skip a cache line when reading bit b is

$$P(b) = \left(1 - \left(\frac{1}{2}\right)^b\right)^{512}$$

for *BitWeaving/V* [1] and

$$P(b) = \left(1 - \left(\frac{1}{2}\right)^b\right)^{64}$$

for *ByteSlice* [14]. Both rapidly approach 100 % for high bit widths, with *ByteSlice* having a huge advantage over *BitWeaving/V* at position eight and still being better at position sixteen as seen in Figure 3.1.

For both *BitWeaving/V* and *ByteSlice* there can be a significant performance penalty due to branch mispredictions in the early pruning code if pruning triggers irregularly. Attempting to prune too early will be unsuccessful, or hit an area where the probability is too low, so branch mispredictions will eliminate any performance advantage from the saved memory bandwidth. In the range of bit widths between 1 and 32 bits, the most mispredictions will occur for code widths between 8 and 16 bits as the probabilities are not close to 0 % or 100 %. This can severely impact the runtime of the scan. To reduce this effect, *BitWeaving/V* partitions the bits into groups and only after a full group has been processed the pruning branch is executed. *ByteSlice* has an intrinsic group size of eight, so no additional handling is needed, but it is also less flexible. The intrinsic group size does not have an effect on early pruning probabilities – the pruning decision is only postponed – but reduces the amount of memory pruned away.

Worst and Best Cases for Other Predicates

For the other predicates $<, \leq, >, \geq$ and BETWEEN the worst and best cases are basically the same as for equality and inequality predicates, they are triggered by exactly the same

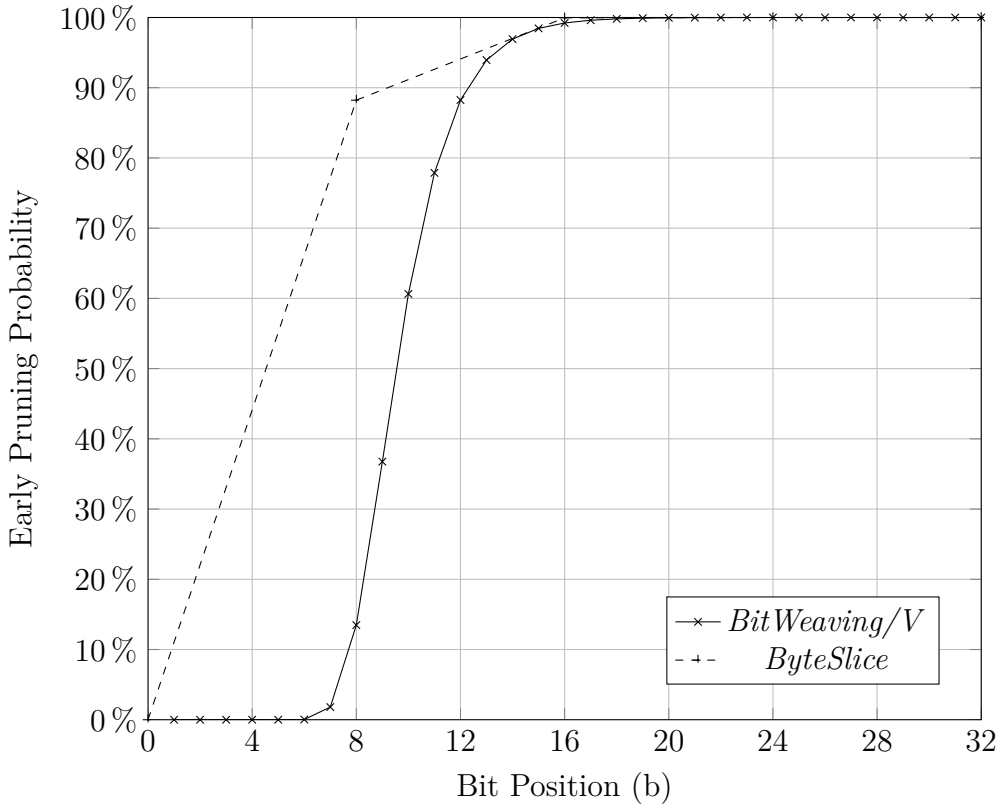

Figure 3.1: Early Pruning Probabilities for *BitWeaving/V* and *ByteSlice*

data. However, there is no direct link between the worst case and selectivity anymore. For $<, \leq, >, \geq$ the worst case is hit when all data is equal to the reference constant as seen in Algorithm 3.1.2. For BETWEEN there is a slight difference as both boundaries have to be taken into account as the computation is only aborted when neither has a current bit equal to the value processed (Algorithm 3.1.3).

Pruning Between Columns

Another application of early pruning is discarding elements early when evaluating a predicate over multiple columns. For example in Figure 3.2, a conjunctive predicate over three columns is evaluated, completely gray colored rows were skipped due to row the

Algorithm 3.1.2 Algorithm to check in bit vector of size n, whether a is less than b

```
1: procedure IsLess(a, b)
2:     for i ← n − 1 to 0 do
3:         if a_i ≠ b_i then
4:             return a_i < b_i
5:     return false
```

Algorithm 3.1.3 Algorithm to check whether bit vector of size n a is between b and c

```
1: procedure IsBetween(a, b, c)
2:     isGreater ← false
3:     isLess ← false
4:     for i ← n − 1 to 0 do
5:         isGreater ← isGreater or a_i > b_i
6:         isLess ← isLess or a_i < c_i
7:         if a_i ≠ b_i and a_i ≠ c_i then
8:             return isGreater and isLess
9:     return false
```

already being discarded in the previous column. The regular *BitWeaving/V*-style early pruning is also applied. This can be extremely effective for conjunctive predicates with very low selectivity or disjunctive predicates with very high selectivity as large parts of the later columns can be pruned away. The order in which the columns are scanned is now also important, a query optimizer could decide on the evaluation order based on its selectivity information.

Intra-column pruning also applies to scan algorithms that do not generally support early pruning, such as *SIMD-Scan*.

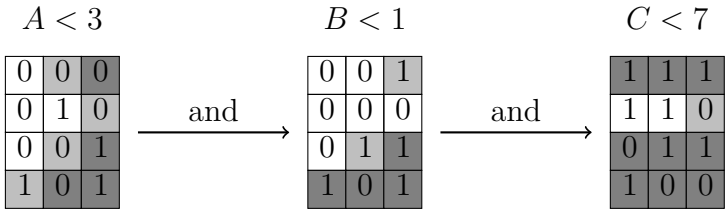

Figure 3.2: Evaluation of the predicate $A < 3$ and $B < 1$ and $C < 7$ on three columns

3.2 Block-Wise Scanning

The output of a scan as performed by *BitWeaving/V* and *ByteSlice* is a bit vector with one bit for every value in the input. Depending on the result of the predicate, the bit is either set or not set. Traditionally a column is scanned in its entirety, the result bit vector is written to memory and the scan proceeds to the next column. This means that an extra bit per value has to be written to memory for the intermediate results which is not needed for the end result. To reduce bandwidth, the columns are now partitioned into fixed-size blocks. After a block is scanned, the bit vector is kept in cache and the algorithm proceeds scanning the next column. This is repeated for all columns, the final result is written and the scan advances to the next set of blocks.

The bus between the processor and main memory is not capable of full duplex operation, so any reduction in write bandwidth counts directly towards the total bandwidth. The

reduction is particularly significant for small bandwidths, for example for a four-bit scan of two columns the total bandwidth would be ten bits, eight read and two written. Block-wise scanning reduces this to nine bits, a 10 % reduction. Figure 3.3 shows reductions for scans over four and sixteen columns, for simplicity all columns are of the same bit width. The savings are calculated by dividing the saved bits by the total bit bandwidth without block-wise scanning, $\frac{c-1}{c(b+1)}$ where c is the number of columns and b is the bit width.

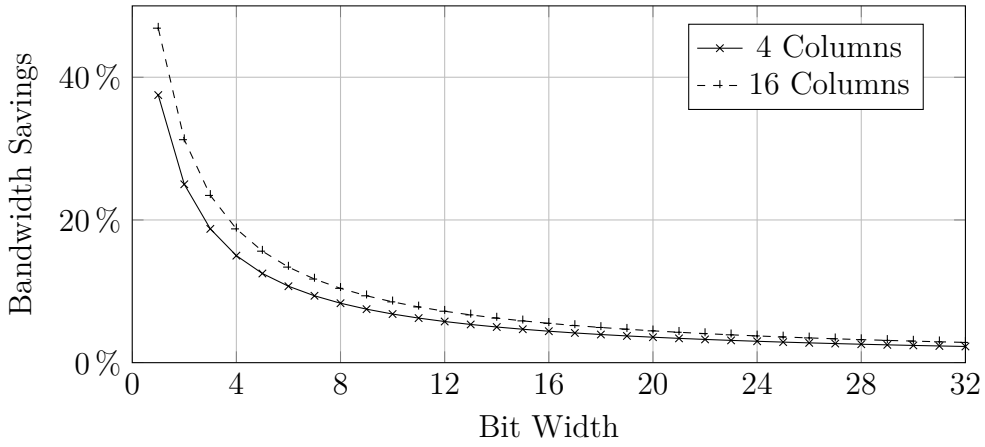

Figure 3.3: Potential bandwidth savings through block-wise scanning

The potential downside for large column counts is that it can destroy the regular consecutive access pattern of the memory. Modern processors are optimized to recognize this pattern and pre-fetch sequential accesses. Since this is the reason why column scans are fast, a significant performance penalty will be seen if pre-fetching stops working due to changed access patterns.

The block-wise technique works for all kinds of scan algorithms, including *SIMD-Scan*, *BitWeaving/V* and *ByteSlice*.

Since *BitWeaving/V* with AVX 2 always processes 256 elements at a time, it would be reasonable to pick 256 as the block size and keep the intermediate values in registers at all time. Since the *BitWeaving/V* implementation is limited by DRAM bandwidth and intermediate values will stay in cache unless the block size is very large this is unlikely to have a performance impact in comparison to other sizes though.

4 Implementation

In this chapter a benchmarking framework is introduced that allows comparing the three implementations of a scan

- *SIMD-Scan* (SAP proprietary) using AVX 2
- *BitWeaving/V* custom implementation using AVX 2
- *ByteSlice* custom implementation using AVX 2

4.1 Benchmark Design

The benchmark code is based on the code used to drive the benchmark in [8]. It contains the infrastructure to execute scans using SAP's *SIMD-Scan* and provides integration with Intel's *Performance Counter Monitor* (PCM) [15] to gather data from the various performance counters in the CPU core. It also contains code to run and validate a range predicate on a chunk of memory and to store the result as a bit vector, that code could be reused partially. A random number generator is also present.

It was upgraded to use the latest available version of *SIMD-Scan* from HANA which contains additional optimizations over those described in the paper. Support for benchmarking *BitWeaving/V* and *ByteSlice* was added, including a new benchmarking mode for unpacking data from the vertical layout. The benchmarking modes in detail are

range scan to bit vector A column of uniformly distributed random 32-bit integers is generated and brought into the representation required for the scan algorithm under a given bit width. A range predicate of the form $0 \leq x < max$ is created, where max is chosen based on the selectivity, e. g., a selectivity of 50 % with eight-bit values would result in the predicate $0 \leq x < 128$. The scan is executed once and compared with a scalar version of the algorithm to ensure correctness, both algorithms produce a bit vector result. Then the scan is performed three times under performance counter supervision and the result is reported.

This is repeated for a range of selectivities from 100 % to 0 % and for all bit widths from 1 to 32 for *SIMD-Scan* and *BitWeaving/V*. For *ByteSlice* only 8, 16, 24 and 32 bits are used. The size of a column was fixed at $2^{25} = 33\,554\,432$ items.

equality scan to bit vector This mode is very similar to the range scan but uses a predicate, of the form $x = c$, where c is a random constant. Since the selectivity cannot be enforced via the predicate the input data is altered to contain c at

random positions under a uniform distribution. The number of *c*s in the result is chosen based on the selectivity. Otherwise the scan is carried out using the same methods as the range scan.

multi-column scan To measure the impact of block-wise scanning, a multi-column mode was added that partitions the randomly generated input data chunk into multiple columns. On those columns, the predicates described above are used. Columns can be scanned in its entirety (block size is infinite) or with smaller block sizes. The results are combined with a logical *and* operation.

It turned out that 2^{25} items were insufficient to completely remove caching effects, so the number of items was later raised to 2^{30} items for all columns combined.

Since a logical *and* is used to combine the results, early pruning was not used in between columns for this benchmark. Pruning between columns with an *and* predicate becomes very sensitive to the selectivities of the individual columns as they multiply from column to column, distorting the results per column.

packing This benchmarking mode generates a chunk of 32-bit integers which are then brought into the format used by the scan algorithm. *SIMD-Scan* packing (and unpacking) is performed using the FastPFOR library by D. Lemire et al [16], *BitWeaving/V* and *ByteSlice* use custom packing routines. The runtime of packing is measured with PCM.

Packing is executed for all bit widths from 1 to 32. For *ByteSlice* zero padding is added for cases not evenly divisible by eight. Not the entire column is unpacked but only slices. The benchmark tests multiple slice sizes from 256 items to 2^{25} items.

unpacking Same as packing, but in this case integers are unpacked from the scan-specific representation into an array of 32-bit integers. Otherwise the configuration is identical to the packing benchmark.

4.2 AVX 2 *BitWeaving/V* Implementation

To have a fair comparison, a clean-room implementation of the *BitWeaving/V* scan algorithm was implemented in C++ using AVX 2 compiler intrinsics. Implementing the scan itself was rather straight forward and only minor deviations from the suggested implementation in the paper were made. The most significant change is that the values that replicated bits of the reference constants over a full vector were computed on demand using Algorithm 4.2.1. This did not change the runtime of the scan but potentially saves resources in the already congested load path.

Putting it all together will result in the scan loop in Algorithm 4.2.2. It processes 256 bit at the same time with only a relatively small number of arithmetic instructions all of which are directly available on AVX 2. A logical not ($\sim$) is not available in AVX 2 but a combined `andnot` instruction can be used instead. This also reduces the total number of

Algorithm 4.2.1 Duplicate bit number b from a 32-bit integer over a full 256-bit vector

```
1: procedure SPLATBIT(uint32_t w, int b)
2:     w ← w << (31 − b)          ▷ move bit b into the most significant position
3:     w ← w >> 31                                          ▷ arithmetic shift right
4:     return _mm256_set1_epi32(w)                ▷ broadcast w over vector width
```

instructions needed. Early pruning can be conveniently implemented using a logical or and the `ptest` instruction in AVX 2, which compares a full 256-bit vector with zero and allows branching on the result.

Algorithm 4.2.2 Evaluating $lower \leq x < upper$ in *BitWeaving/V* for AVX 2

```
1: procedure BITWEAVINGSCAN(uint32_t upper, uint32_t lower, ___m256i *out, int
   width, int num, int numGroups, int groupSize)
2:     for i = 0 to num/256 do
3:         Mlt ← 0
4:         Mgt ← 0
5:         Meq1 ← AllOnes
6:         Meq2 ← AllOnes
7:         for j = 0 to numGroups do
8:             for k = (j + 1) ∗ groupSize downto j ∗ groupSize do
9:                 if Meq1 = 0 or Meq2 = 0 then
10:                    break                                    ▷ early pruning
11:                Bits ← load one 256-bit word
12:                splatLower ← splatBit(lower, k)
13:                splatUpper ← splatBit(upper, k)
14:                Mgt ← Mgt or (splatLower and ∼(Meq1 and Bits))
15:                Mlt ← Mlt or (Bits and ∼(Meq2 and splatUpper))
16:                Meq1 ← (Bits xor splatLower) and ∼(Meq1)
17:                Meq2 ← (Bits xor splatUpper) and ∼(Meq2)
18:        out[i] ← Meq1 or (Mlt and Mgt)
```

The most complex part of the implementation was designing a fast way to unpack and pack the input data into the format required by the *BitWeaving/V* scan. For packing the `movmskps` instruction was used combined with vector shifts. This processes eight 32-bit integers at the same time but is still slow as a lot of memory is touched. Since there is no reverse operation of `movmskps` in the AVX 2 instruction set, unpacking was implemented by replicating the bits of a single value over a full vector and using shifts and masking. The basic idea is shown in Algorithm 4.2.3.

Packing and unpacking is generally very slow for *BitWeaving/V*, because unpacking a single value has to touch as many cache lines as there are bits in the value. This is a significant disadvantage in comparison with the other methods.While the algorithm looks extremely simple, it is much slower than the scanning algorithm. The parallelism

here is very limited, because the innermost loop only processes 8 bits at a time which is extremely low for SIMD code. This code is already fairly optimized and there is not much improvement possible with the currently available instruction sets. AVX 2 contains limited support for gathering and scattering bits using the `pdep` and `pextr` instructions, but those work on 64-bit words instead of 256-bit vectors and did not improve the runtime of packing. A full 256-bit bit-wise version of those instructions could improve the situation.

Algorithm 4.2.3 Simplified *BitWeaving/V* pack algorithm, from 32-bit integers

```
1: procedure packBitWeaving(uint32_t *in, uint32_t *out, int width, int num)
2:     for i = 0 to num/256 do
3:         for j = 0 to 32 do
4:             Bits ← load one 256-bit word from in
5:             for k = 0 to width do
6:                 Mask ← movemask(Bits << (32 − j))          ▷ Extract 8 bits
7:                 Store Mask to the right position in out
```

4.3 AVX 2 *ByteSlice* Implementation

In contrast to the *BitWeaving/V* scan the *ByteSlice* turned out to be trickier to implement due to limitations in the AVX 2 instruction set that were not described in the *ByteSlice* paper. The implementation relies on unsigned integer comparisons, but AVX 2 only supplies signed versions. To turn the signed versions into unsigned compares, all sign bits are flipped as illustrated in Algorithm 4.3.1. This adds more instructions to the main scan loop than earlier anticipated.

Algorithm 4.3.1 Extract a reference byte and do an unsigned comparison with a vector

```
1: procedure unsignedCompare(__m256i bits, uint32_t reference, int bytenumber)
2:     bitsFixed ← _mm256_xor_si256(bits, _mm256_set1_epi8(0x80));
3:     referenceFixed ← reference ^ 0x80808080
4:     referenceVector ← splatByte(referenceFixed, bytenumber)
5:     return _mm256_cmpgt_epi8(bitsFixed, referenceVector)
```

The byte extraction and duplication is also implemented differently from the paper. It uses the same method as the *BitWeaving/V* implementation, but adapted to bytes instead of bits (Algorithm 4.3.2).

Putting it all together we end up with the full scan in Algorithm 4.3.3. The only addition is `equalCompare` which is the same as `unsignedCompare` with the *greater than* comparison replaced by *equality*. By inlining the compare functions into the scan loop, some redundant work can be eliminated, but this predicate is significantly more complex than the same predicate implemented in *BitWeaving/V*. This is primarily due to the lack of unsigned vector compares in AVX 2, which have to be emulated with multiple

Algorithm 4.3.2 Duplicate byte number b from a 32-bit integer over a full 256-bit vector

```
1: procedure SPLATBYTE(uint32_t w, int b)
2:     w ← w >> (b * 8)                    ▷ move byte b into the least significant position
3:     return _mm256_set1_epi8((uint8_t)w)    ▷ broadcast w over vector width
```

instructions. As the instruction budget to stay limited by memory bandwith is very tight, this may be a disadvantage over other scan algorithms. The next version of AVX, AVX-512 will include unsigned comparisons, eliminating this issue altogether. Just as with *BitWeaving/V*, early pruning can be conveniently implemented using a single *or* instruction followed by AVX 2's `ptest`. This scan loop also processes 256 bits at a time but only 32 values concurrently. As the loop is not as nested as *BitWeaving/V*, the early pruning branch is executed more often. Operation `movemask` compresses the result vector into a 32-bit value that is written to the output buffer.

Algorithm 4.3.3 Evaluating $lower \leq x < upper$ in *ByteSlice* for AVX 2

```
1: procedure BYTESLICESCAN(uint32_t upper, uint32_t lower, uint32_t *out, int
   width, int num)
2:     for i = 0 to num/32 do
3:         Mlt ← 0
4:         Mgt ← 0
5:         Meq1 ← AllOnes
6:         Meq2 ← AllOnes
7:         for j = (width+7)/8 − 1 downto 0 do
8:             if Meq1 = 0 or Meq2 = 0 then
9:                 break                                   ▷ early pruning
10:            Bits ← load one 256-bit word
11:            gt ← unsignedCompare(Bits, lower, j)
12:            lt ← unsignedCompare(upper, Bits, j)
13:            eq1 ← equalCompare(Bits, lower, j)
14:            eq2 ← equalCompare(Bits, upper, j)
15:            Mgt ← Mgt or (gt and Meq1)
16:            Mlt ← Mlt or (lt and Meq2)
17:            Meq1 ← Meq1 and eq1
18:            Meq2 ← Meq2 and eq2
19:        out[i] ← movemask((Mlt and Mgt) or Meq1)
```

Packing and unpacking into the *ByteSlice* representation can be done extremely quickly on AVX 2. The basic idea for packing is to load a full 256-bit vector containing eight 32 bit integers. Then use vector shuffle instructions to rearrange the bytes of those integers to be clustered together. That means all most significant bytes are in the first 64 bits of the vector. Then up to four 64 bit integers are extracted from that vector and stored at the corresponding positions in the output buffer. This only touches five memory

locations, reading from one and storing to four at the maximum bit width of 32 bits. All accesses are sequential, so it should perform very well on modern hardware.

The code in Algorithm 4.3.4 shows the basic packing method. A 256 bit vector is loaded and vector shuffles are used to group the vector elements by their byte number. The first 64 bit integer of the vector then contains the most significant bytes of the eight 32 bit integers that were loaded from memory. The next 64 word contains the second-most significant bytes and so on. This can be achieved with four shuffle instructions on AVX 2 and possibly less on AVX-512, as it contains more general shuffle instructions. The up to four 64 bit values from the vector are then individually stored to the right positions in the output. This algorithm still is not perfect SIMD as it only stores 64 bit quantities. The arithmetic is performed on a full 256-bit word though which is much faster than the eight bits of *BitWeaving/V* packing.

For *ByteSlice* unpacking and packing a single value only touches eight times fewer cache lines than *BitWeaving/V*. This makes the transformation process much faster.

Algorithm 4.3.4 Simplified *ByteSlice* pack algorithm, from 32 bit integers

```
1: procedure packByteSlice(uint32_t *in, uint32_t *out, int width, int num)
2:     for i = 0 to num/8 do
3:         Bits ← load one 256-bit word from in
4:         Bits ← shuffle(bits)
5:         for j = (width+7)/8 downto 0 do
6:             toStore ← extract 64 bits from Bits at position j
7:             Store toStore at the right position in out
```

5 Evaluation

After laying down the theoretical analysis of the scan algorithms and providing an efficient implementation, it is time to verify whether the expectations hold up in reality. This chapter does so with a focus on six basic questions.

1. Can the early pruning in *BitWeaving/V* beat the performance of *SIMD-Scan*, despite the branching overhead?

2. How large is this branching overhead?

3. Is the early pruning probability difference of *BitWeaving/V* and *ByteSlice* visible in practice?

4. What is the impact of predicate selectivity on early pruning?

5. How much time does it take to bring data in the formats required by the scanning algorithms?

6. Are there visible improvements due to the bandwidth reduction of block-wise scanning?

This chapter will try to find answers to those questions and confirm those answers using benchmark results.

5.1 Benchmark Setup

All benchmarks were performed on a dual-CPU Intel Xeon E5-2697 v3 machine. This Haswell setup had 56 physical and logical cores but all benchmarks only measure single-core performance. The CPU has a last-level cache size of 35 megabytes. Memory was supplied by eight banks with eight gigabytes of DDR4 RAM each, running at 1067 MHz. On the software side Ubuntu 14.04 was used and GCC 4.8 compiled the benchmark code, except for *SIMD-Scan* which was using a precompiled object file compiled with ICC.

All runtimes were recorded using C++11's `std::chrono::high_resolution_clock`, memory bandwidth to and from the DRAM controller was measured from the CPU performance counters using Intel PCM.

5.2 Comparing *SIMD-Scan* to *BitWeaving/V*

The first and most important evaluation result will be a direct comparison between *SIMD-Scan* and *BitWeaving/V*, as seen in Figure 5.1. In this configuration early pruning was performed after reading 4 bits as suggested in [1].

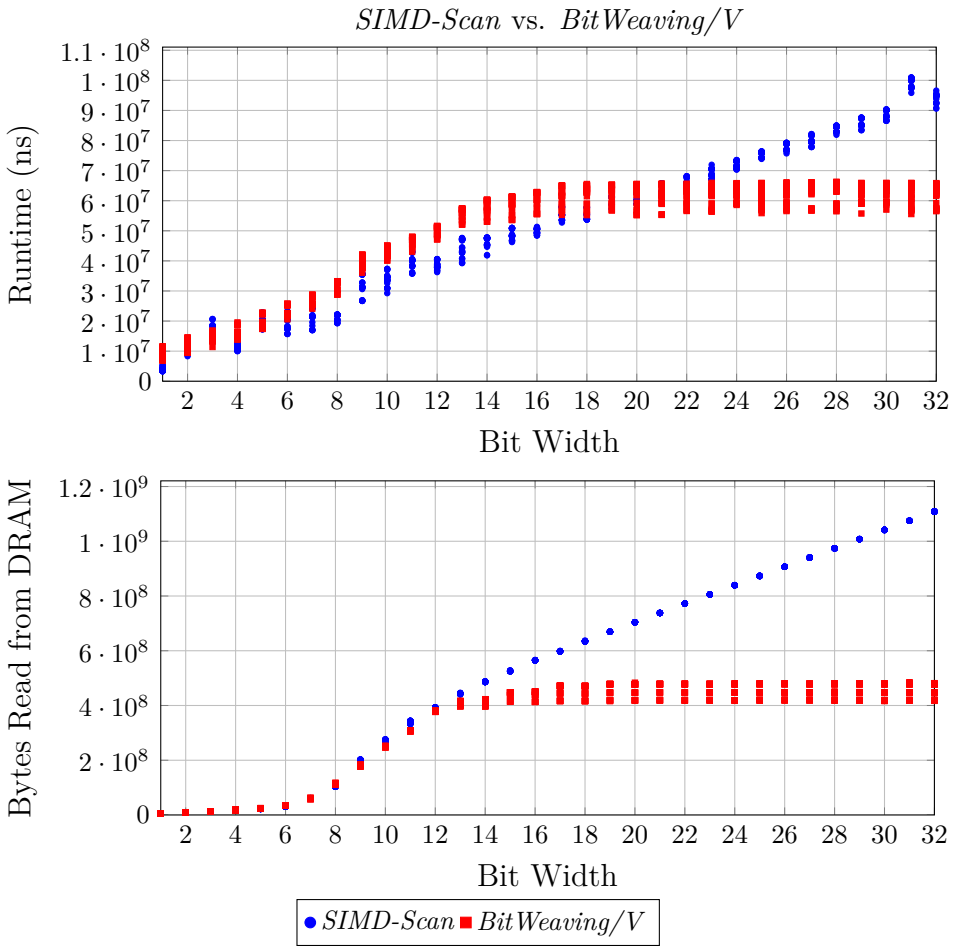

Figure 5.1: Runtime of a range predicate with *SIMD-Scan* and *BitWeaving/V*, sum of eight runs over 2^{25} random integers. *BitWeaving/V* group size is four.

As described earlier the benchmark was run with multiple selectivities, which are represented as a stack of markers in this chart. Since selectivity has virtually no impact on the runtime of *SIMD-Scan*, the variance between the selectivities can be interpreted as an indicator for the noise level. The read bandwidth used by *SIMD-Scan* is approximately 11 gigabytes per second and scales linearly over all bit widths with only a few exceptions. 11 gigabytes per second matches the bandwidth available to a single core on the Haswell processor used. This was verified with a separate benchmark only touching every cache

line once. Total bandwidth is significantly higher but can only made use of by utilizing multiple cores. The graph itself can be analyzed as four different parts.

- Bit widths $0 - 8$: due to the large amount of last-level cache available, not all of the memory is read from DRAM but was kept in the cache over multiple iterations. This skews the result, however there is virtually no difference in scan performance between *SIMD-Scan* and *BitWeaving/V*. Early pruning can theoretically abort the scan after four or eight bits but the probability calculated in Chapter 3 is very low, so no effect is observed.

- Bit widths $8-12$: as calculated earlier and visible in the memory graph early pruning is still not effective here. The overhead of the additional branch is slowing down *BitWeaving/V* in comparison to *SIMD-Scan* which does not have any branching in its scan loop. *SIMD-Scan* is the better scan algorithm for these common bit widths.

- Bit widths $12 - 20$: Early pruning starts to become more effective and less memory is loaded from DRAM. The early pruning overhead is clearly visible and yields a slower runtime than *SIMD-Scan*, so its still preferable over *BitWeaving/V*.

- Bit width $20 - 32$: Early pruning is very effective and beats the runtime of *SIMD-Scan* with a growing margin. The performance curve is entirely flat and will stay the same if the bit width is extended further. *BitWeaving/V* is the preferable algorithm for this kind of scan.

All in all, this benchmark shows that *BitWeaving/V* can beat *SIMD-Scan*, but it is not always a clear win. It provides no advantage over *SIMD-Scan* at low bit widths and is slower at medium bit widths. Only at relatively large bit widths early pruning can make up its own overhead and provide an advantage that grows larger as the bit width gets larger. This contradicts the observations of the *BitWeaving/V* paper [1] where *BitWeaving/V* is significantly faster than all other methods. It is very likely that a faulty implementation of *SIMD-Scan* was used for those benchmarks as it is very competitive.

5.3 Reducing the Early Pruning Overhead

As seen in the last graphs there is a significant overhead in the area around 16 bits when comparing *BitWeaving/V* with *SIMD-Scan*. This overhead comes from the additional branching that has to be performed at every early pruning step. To reduce the amount of branching, one possible tweak is to reduce the number of bits that are processed without early pruning. This diminishes the effectiveness of early pruning though. For the benchmark in Figure 5.2 early pruning was performed after every 4, 8 or 16 bits processed. The selectivity for this benchmark was fixed at 50 %.

As expected, smaller group sizes work better with regard to the number of bytes read from DRAM as early pruning is a lot more effective. The impact on performance is

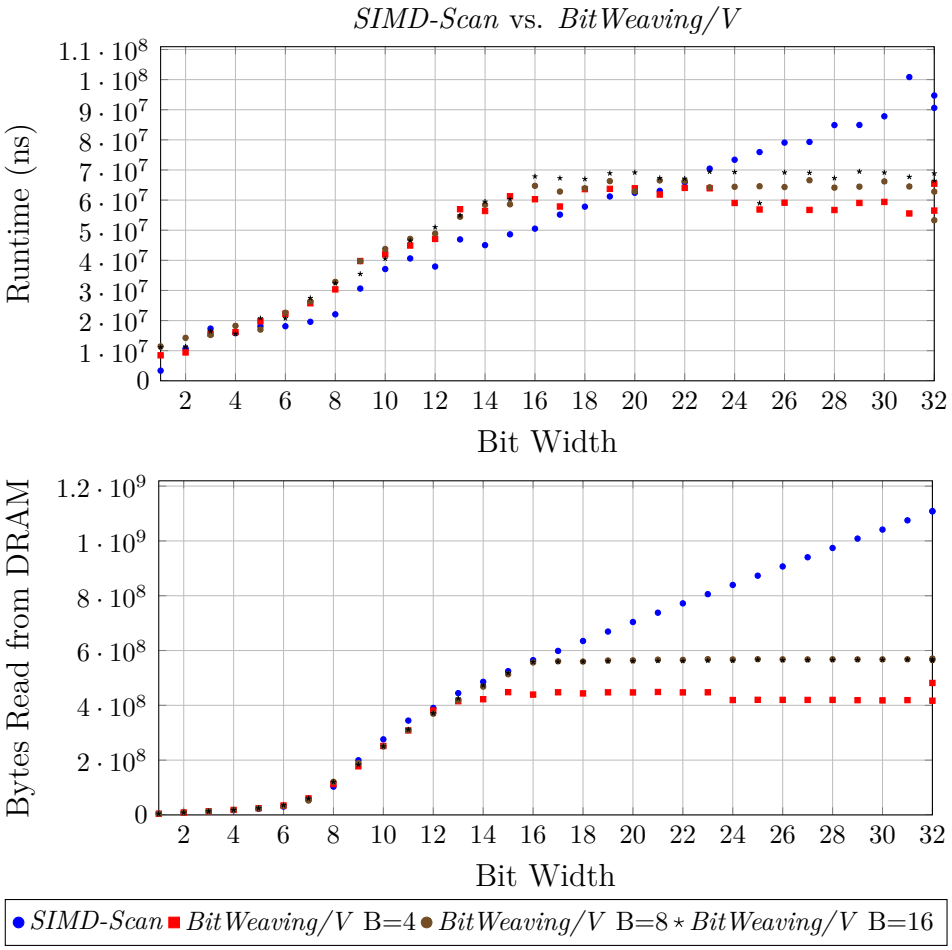

Figure 5.2: Runtime of a range predicate with *SIMD-Scan* and *BitWeaving/V*, sum of eight runs over 2^{25} random integers. Varying group sizes.

very small though, the expected improvement due to fewer mispredicted branches is not visible. Nullifying the overhead does not seem possible.

As the difference is very small, a block size of eight was chosen for the following experiments, as it compares better to *ByteSlice* which has an intrinsic group size of eight due to processing values at the byte level.

5.4 Comparing *BitWeaving/V* to *ByteSlice*

Figure 5.3 shows the same range predicate used for the comparison between *SIMD-Scan* and *BitWeaving/V* on *ByteSlice* and *BitWeaving/V*. Only bit widths supported by *ByteSlice* without padding are shown as *ByteSlice* performance for the bit widths in between is identical to the next higher bit width without padding. It can be clearly seen that *ByteSlice* is not memory bound in this benchmark. Even though the pruning works

better for case 24 the runtime performance is much worse. The gap can be reduced by using a simpler predicate, for example in Figure 5.4 an equality predicate was used. The equality predicate is a small subset of the range predicate, so the amount of arithmetic performed during the scan is significantly lower. While much narrower, even this was not enough to eliminate the performance advantage of *BitWeaving/V* over *ByteSlice*. A possible explanation for this behavior is the increased number of branch instructions executed by *ByteSlice*. While *BitWeaving/V* branches after every n 256-bit words read, where n is the group size, *ByteSlice* checks for early exit after every 256-bit word. This makes pruning more effective at the cost of more branching. It would be possible to add groups to *ByteSlice* but since it works on bytes, the group size must be a multiple of eight, severely limiting early pruning opportunities, so this method was not tried.

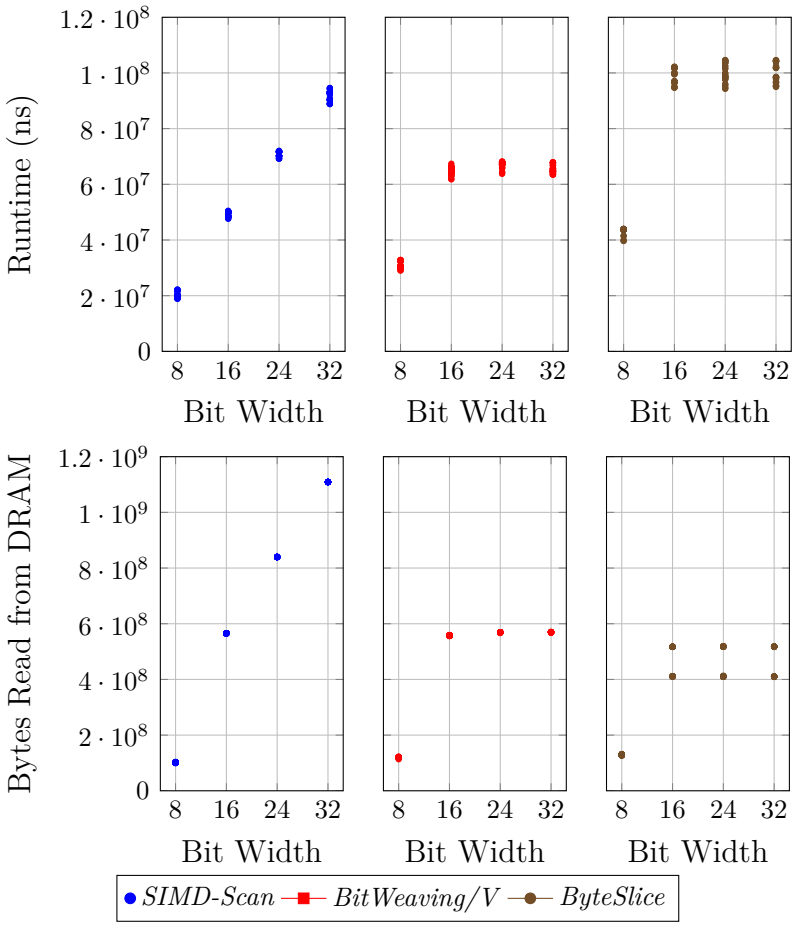

Figure 5.3: Runtime of a range predicate with *BitWeaving/V* and *ByteSlice*, sum of eight runs over 2^{25} random integers. *BitWeaving/V* group size is eight.

This benchmark contradicts the benchmarks seen in the *ByteSlice* paper [14] where *ByteSlice* is almost always faster than the vertical bit-packed format and *SIMD-Scan*. It is likely that the same faulty implementation of *SIMD-Scan* was used for this benchmark

that was also used in the *BitWeaving/V* paper and a very simple predicate was used for *ByteSlice*.

It is also possible that this gap will be closed for range predicates by the next CPU generation or more optimized predicate evaluation code. *ByteSlice* has other advantages, namely a higher pruning probability and very efficient packing and unpacking, which would make it highly preferable over *BitWeaving/V* if performance parity can be reached.

5.5 Comparing Predicates and Selectivities

So far only a range predicate was tested. For random data in a uniform distribution this has the very nice property of hitting the worst case very rarely. As a reminder, for the used predicate of the form $x \leq y < z$ the worst case is triggered when y is either equal to x or z. So for the next benchmark this worst case will be investigated. For reasons of simplicity an equality predicate was used, but the observation also applies to any inequality or range predicate. Only 8, 16, 24 and 32 bits are evaluated, because others are not directly supported by *ByteSlice*. The results in runtime and DRAM reads are plotted in Figure 5.4.

- *SIMD-Scan* is the base line. *BitWeaving/V* and *ByteSlice* read at most as much data as *SIMD-Scan*. The performance represents the maximum amount of data one CPU core can process at the given time, so any other algorithm can not be faster unless it reads less data.

- When the same amount of data is read, the performance of *BitWeaving/V* tends to be almost identical to *SIMD-Scan*. On the one hand, the equality predicate requires very little arithmetic work compared to the range predicate shown earlier. On the other hand early pruning does not work well, so the early exit branch can be easily predicted by the hardware.

- Early pruning is more effective for *ByteSlice* than for *BitWeaving/V*, but performance tends to be slightly worse. The implementation of *ByteSlice* is not entirely memory bound like *SIMD-Scan* and *BitWeaving/V* are.

One important fact that is not represented in this graph is that for *BitWeaving/V* and *ByteSlice* about half of the points are in the topmost position. This is shown in detail in Figure 5.5 which shows the results against the selectivity on a logarithmic scale. The results are only shown for a bit width of 32 as this shows early pruning in action for both *BitWeaving/V* and *ByteSlice*.

The graph clearly illustrates the early pruning patterns calculated in Chapter 3. The bit width of 32 bits is also rather large so early pruning can work decently here and the branch overhead does not penalize *BitWeaving/V* in comparison to *SIMD-Scan*. For *ByteSlice* the situation looks a bit different though, as there are obvious deficiencies in performance here.

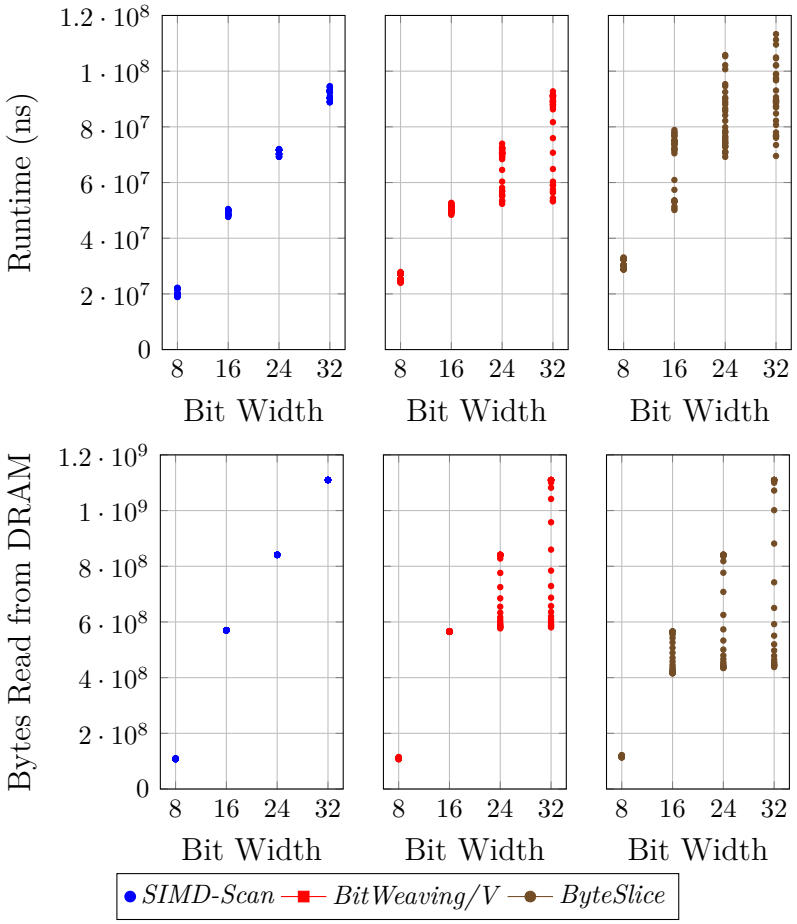

Figure 5.4: Runtime of an equality predicate with *SIMD-Scan*, *BitWeaving/V* and *ByteSlice*, sum of eight runs over 2^{25} random integers. 32 different selectivities between 100 % and 0 %.

5.6 Packing and Unpacking

Packing the data into the representation required by the scan algorithm and unpacking them for consumption are two operations that are often overlooked when measuring scan performance. While packing performance is usually done once for a column which is scanned many times, unpacking is necessary to materialize the results. This can be circumvented by keeping the data available in multiple formats, one optimized for scanning and another optimized for unpacking, but this obviously requires twice the amount of memory. Figure 5.6 shows packing of 32-bit values into the scan representation of various sizes and the reverse, unpacking the scan representation back into 32-bit values. For *SIMD-Scan* at bit width 32 both operations are equivalent to `memcpy`, performing a verbatim copy and not modifying the data at all.

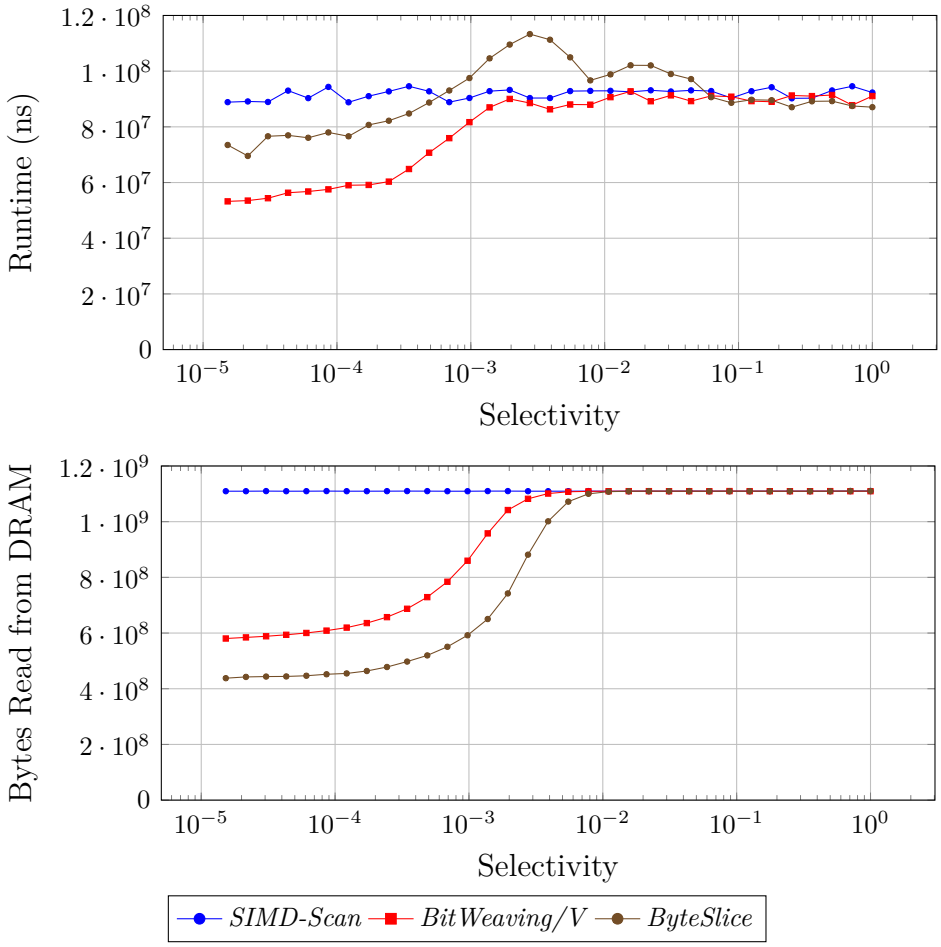

Figure 5.5: Different selectivities for *SIMD-Scan*, *BitWeaving/V* and *ByteSlice* for code width 32.

The most obvious issue is that *BitWeaving/V* is orders of magnitudes slower than the other two representations. This is due to the amount of arithmetic needed to extract and insert individual bits while manipulating full bytes or more can be done much more efficiently with shifts and vector shuffles. It should also be noted that packing is more expensive than unpacking in this scenario.

For *SIMD-Scan* and *ByteSlice* runtime is extremely close to the speed of memory as seen in the 32-bit case of packing. The difference in performance between the two is mostly due to the *ByteSlice* packing algorithm making use of SIMD while the others were implemented using scalar 32 or 64 bit integers. There is still more potential for optimization here.

In conclusion it is certainly feasible to use the in-memory representation of *SIMD-Scan* or *ByteSlice* as the storage format for both scanning and extraction of values, unpacking from *BitWeaving/V* is prohibitively expensive, so its usage is restricted to being an index.

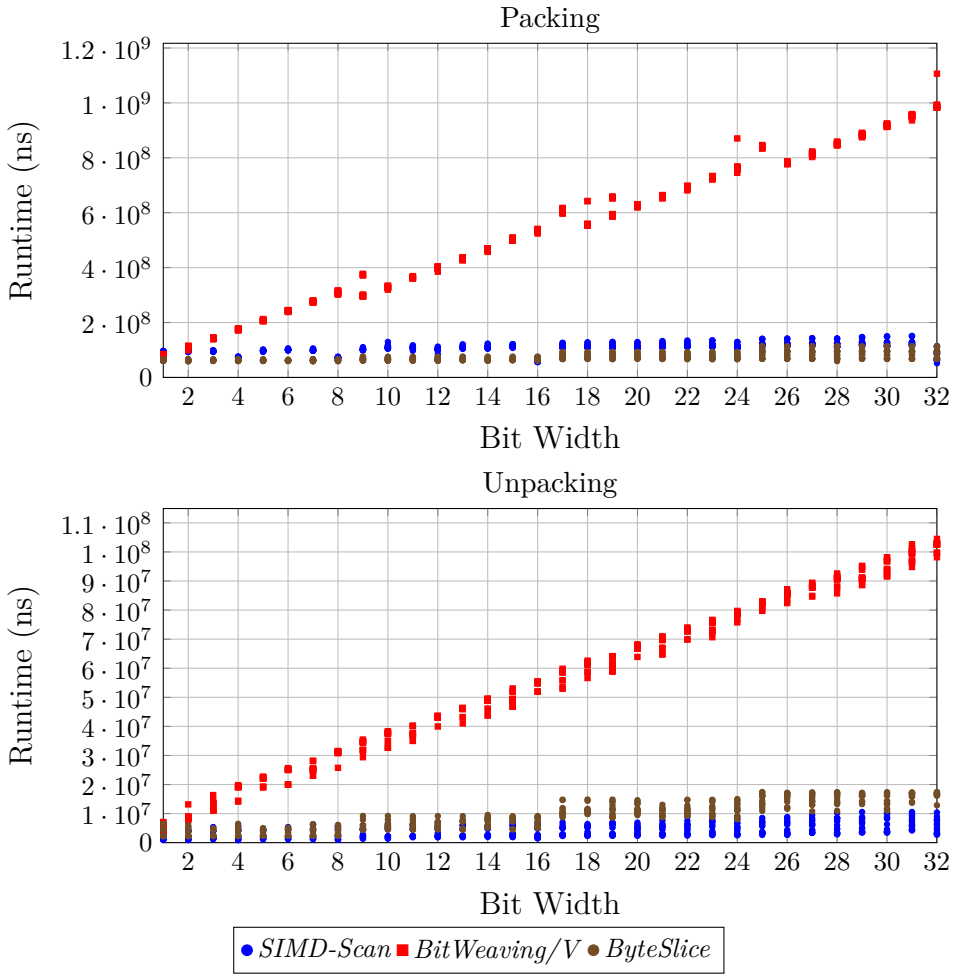

Figure 5.6: Packing 32-bit values into *SIMD-Scan*, *BitWeaving/V* and *ByteSlice* representations and the reverse. 256 values were transformed at a time.

5.7 Adding Block-Wise Scanning

The last part of this evaluation is measuring the impact of block-wise scanning on the used memory bandwidth, and thus performance. Five block sizes were evaluated, only *BitWeaving/V* results are shown here as *SIMD-Scan* and *ByteSlice* show exactly the same behavior.

- Block size 1. This is equivalent to a row-wise scan. One element is read from the first column, compared and then the next column is processed before eventually advancing to the next row. For this test a scalar loop was used as the other methods have minimum block sizes, because of their SIMD usage.

- Block size 256. This is the native width of *BitWeaving/V* using AVX 2.

- Block size 1024. Picked to see if using the native width has an positive or negative impact.
- Block size $2^{17} = 131\,072$. This was used to emulate a database system that divides its data into chunks of 100 000 elements. Rounded up to the next power of 2.
- Block size ∞. This is equivalent to scanning the entire column before advancing to the next column.

Except for the infinite block size all intermediate bit vectors always fit in cache and since the bit vectors are permanently overwritten it is very unlikely for them to be evicted from cache.

First a word on block size 1. Tests showed that this method was not competitive at all with the other sizes, because it inhibits using SIMD instructions. Testing showed a slow down of more than 10×. Using this method is not recommended for any type of scan.

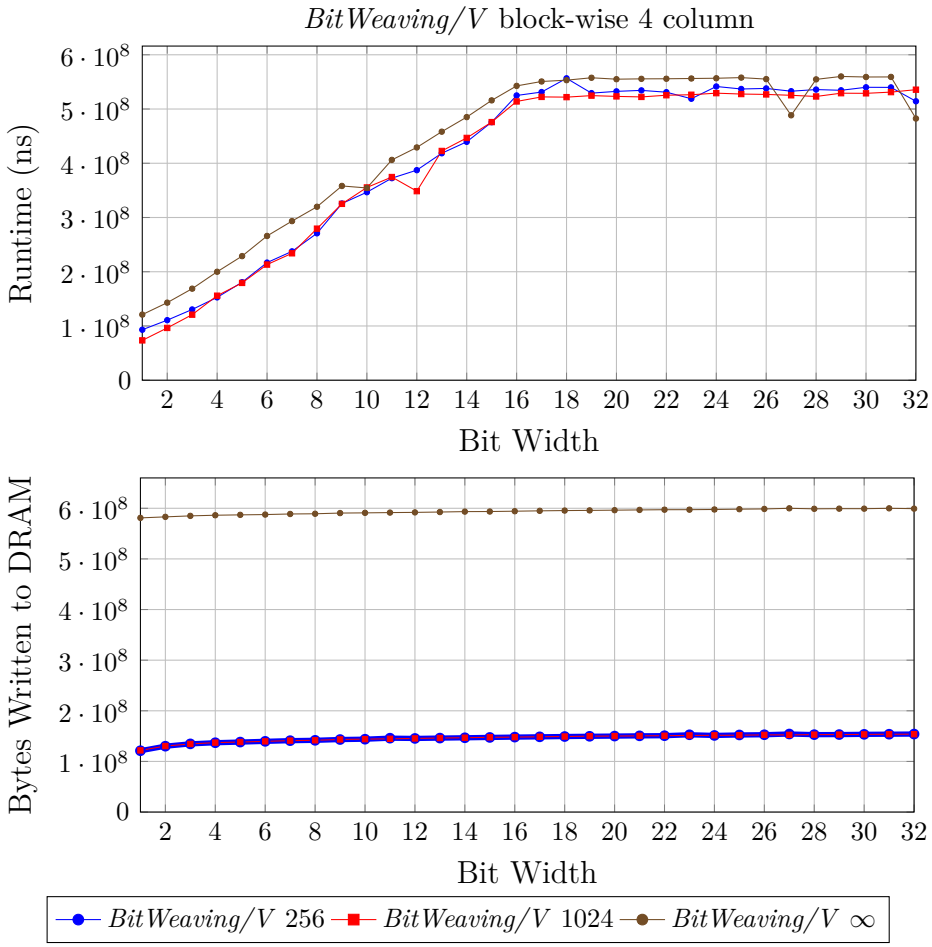

Figure 5.7: Comparing different block sizes for four columns with *BitWeaving/V*

The graph in Figure 5.7 compares block sizes 256 and 1024 with an infinite block size. There is a small performance benefit of this method, but it is not very pronounced, as calculated in Chapter 3. What does not match the analysis is that the bit width has virtually no effect on the performance difference even though the percentage of saved memory bandwidth is much higher for small bit widths. This effect is most likely due to cache effects.

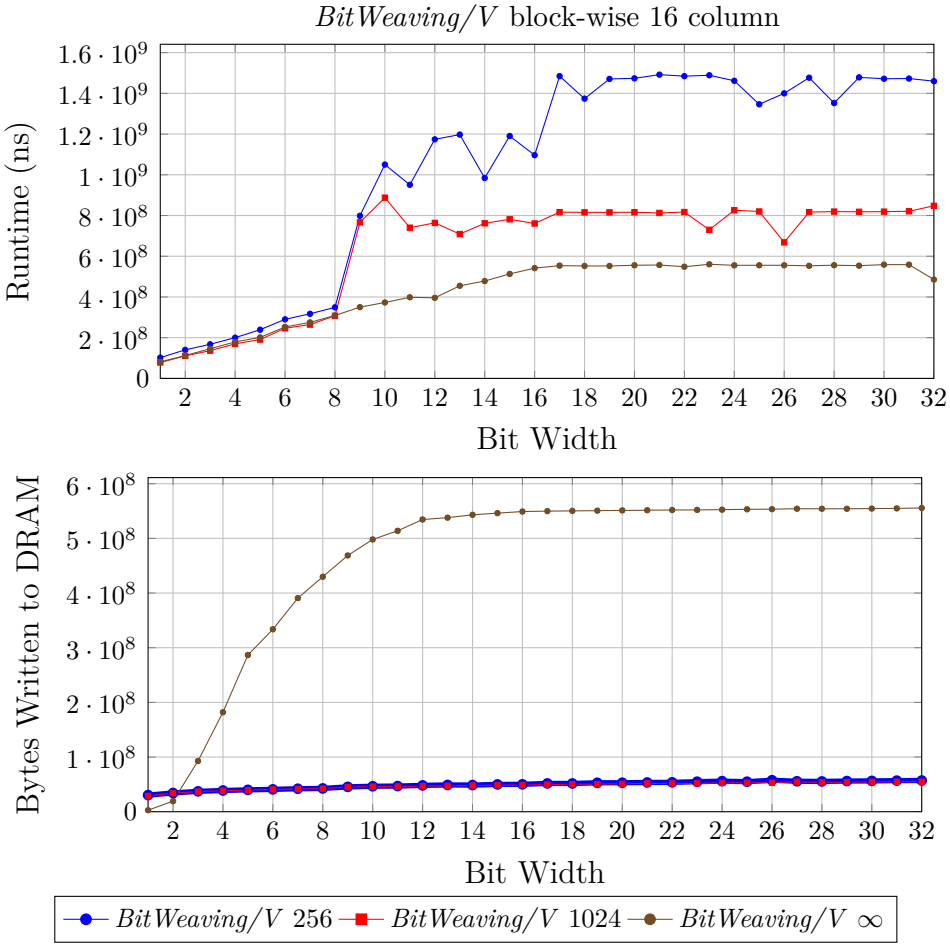

Figure 5.8: Comparing different block sizes for sixteen columns with *BitWeaving/V*

Figure 5.8 shows the same scan, but this time the data was distributed over 16 columns. This has a catastrophic effect on scan performance, because the memory prefetching of the used Haswell CPU cannot cope with this many streams, effectively removing the performance advantage of sequential access. The amount of written data is much lower for this test case though. In both tests the amount of input data was the same overall, but more of the results were intermediate ones compared to the four column case. The intermediate results fit in cache for small bit widths here leading to a curved graph for the infinite block size, this will not happen for larger data sets.

Once the prefetcher ceased to work, there is also a visible advantage of using larger block sizes. In this case 1024 performed significantly better than 256 as there is some sequential access left that the CPU can recognize. This lead to the next test that uses a vastly larger block size of 2^{17} which is tested in Figure 5.9 with a larger data set.

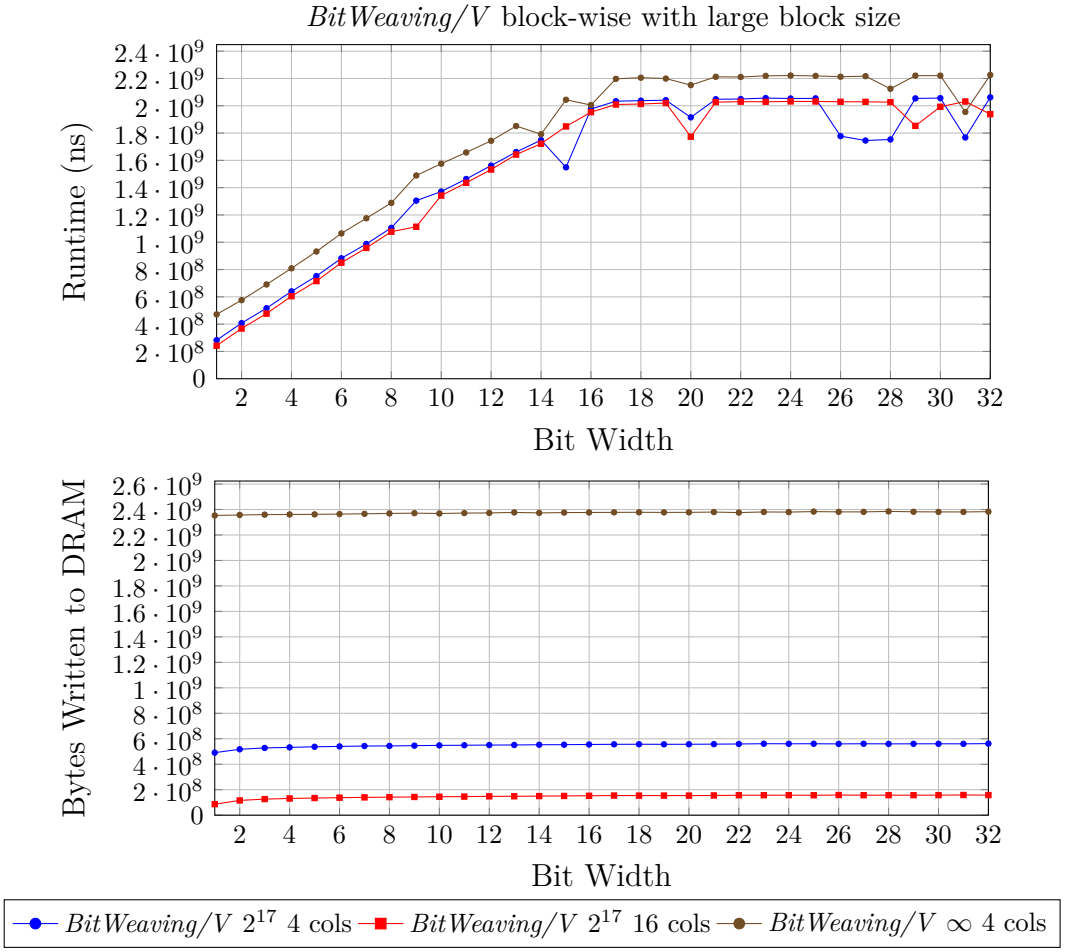

Figure 5.9: Comparing *BitWeaving/V* at block size 2^{17}

With a block size this large the prefetcher issue of smaller block sizes is completely avoided and we gain a small performance benefit by avoiding to write intermediate results into main memory. The result for one block only weighs $\frac{2^{17}}{8} = 2^{14}$ bytes which easily fits into any cache on the processor.

5.8 Summary

Coming back to the questions posed in the introduction of this chapter, all of them can now be answered using the gathered benchmark data and the analysis that was done on that data.

1. *Can the early pruning in* BitWeaving/V *beat the performance of* SIMD-Scan*, despite the branching overhead?*
   Yes, but not for all bit widths. On the tested machine the branching overhead is largest around a bit width of 16 and amortizes if the bit width gets larger. For bit widths around 32 bits *BitWeaving/V* has a significant advantage over *SIMD-Scan*.

2. *How large is this branching overhead?*
   Large enough to make *BitWeaving/V* slower than *SIMD-Scan* for some bit widths.

3. *Is the early pruning probability difference of* BitWeaving/V *and* ByteSlice *visible in practice?*
   It is visible when measuring the amount of memory read from DRAM, but in this implementation it was not faster than *BitWeaving/V*. *ByteSlice* has other advantages over *BitWeaving/V* and it is not unlikely that *ByteSlice* can gain ground on *BitWeaving/V* on the next generation of Intel hardware or with a better tuned implementation.

4. *What is the impact of predicate selectivity on early pruning?*
   For uniform random data there is virtually no impact for predicates containing less-than or greater-than operations. For equality the selectivity directly impacts early pruning and even at low selectivities early pruning is often defeated entirely.

5. *How much time does it take to bring data in the formats required by the scanning algorithms?*
   Both *SIMD-Scan* and *ByteSlice* really shine in this area. *BitWeaving/V* is orders of magnitudes slower, making it infeasible as the primary representation; making it only usable as an index.

6. *Are there visible improvements due to the bandwidth reduction of block-wise scanning?*
   Yes. There is a small but visible effect. Making the block size too small can regress performance though, so a large block size is better as long as the intermediate bit vectors comfortably fit in cache.

6 Conclusions and Future Work

In this thesis multiple ways of performing full table scans in a column store were analyzed and evaluated. The baseline is *SIMD-Scan*, which works on tightly packed data and, as the name says, makes use of SIMD. The key observation was that *SIMD-Scan* is only bound by memory bandwidth on a single core on a current Intel Haswell server and by making full use of AVX 2. This made *SIMD-Scan* the baseline algorithm and possible improvements were looked at. To qualify the new scan algorithm has to have a very tight inner loop as the available memory bandwidth only allows for a small amount of instructions to be executed per word loaded from memory. This also makes the use of wider SIMD instructions very important as it reduces the number of instructions executed per byte read from memory. After many options of possible compression techniques were examined *BitWeaving/V* and *ByteSlice* were chosen as they use an extremely simple compression scheme which uses exactly the same amount of bits as *SIMD-Scan*. The same data size made comparing the different approaches feasible.

The huge advantage of *BitWeaving/V* and *ByteSlice* over *SIMD-Scan* is the ability to abort processing of an element before all bits of that element are loaded. Combined with a memory layout that lets the processor avoid loading unnecessary bits this can yield a significant reduction in memory bandwidth used. *BitWeaving/V* uses a vertical bit layout where bits of different code words are stored together depending on their position in the input word. *ByteSlice* is very similar but uses a vertical byte layout, where individual bytes keep their inner layout but are placed into different memory locations. Nevertheless, early exiting adds a probabilistic component to the analysis of the runtime of the scan, depending on the input data, which makes the scan less predictable.

While analyzing the behavior of *BitWeaving/V* another extremely important observation was made. Since a modern processor always loads data from RAM in constant-sized cache lines, only skipping a cache line in its entirety can avoid loading data from main memory. In theory, the probability of early exiting depends on the number of elements processed at the same time, and thus the register size. A single element can trigger the worst case of loading all bits and the probability that such an element is in the register grows with register size. However, since early exiting only avoids memory traffic, the probability is always fixed at the cache line size regardless of register size. This of course only applies as long as the algorithm is only limited by memory bandwidth, and not by other CPU resources such as instruction throughput.

In benchmarks the early pruning algorithms can easily beat *SIMD-Scan* on BETWEEN predicates with bit widths larger than 20 bits per word and uniform random data. For small bit widths up to about 10 bits there is no difference in scan runtime and in between the pruning scans are actually slower than *SIMD-Scan*. This is due to the

branching overhead incurred by pruning early and makes it much harder to make a clear recommendation.

While inequality predicates such as *less than* and *greater than* show little to none sensitivity on the selectivity, equality predicate performance is directly bound to the selectivity. The selectivities where early pruning becomes significant for those predicates lie at 0.2 % for *BitWeaving/V* and 1.6 % for *ByteSlice* which is very low. For this use case, pruning algorithms have very little advantage over *SIMD-Scan*.

In early pruning also lies the major difference in scan performance between *BitWeaving/V* and *ByteSlice*. *ByteSlice* stores fewer elements per cache line, so the early pruning probability is higher. On the other hand, it also executes more branches per word read from memory, so the branching overhead is higher. In this thesis *ByteSlice* could not beat *BitWeaving/V* in terms of scan performance. The effect is more pronounced for complex predicates such as BETWEEN than for simpler predicates such as equivalence, mostly due to AVX 2 missing important instructions for the operations performed by *ByteSlice* increasing the total number of instructions executed per word.

Another extremely important thing to keep in mind when choosing a scan algorithm is the time needed to bring data into the representation needed for the scan and also the performance of the reverse operation which is needed to materialize the results. Testing shows that *BitWeaving/V* is prohibitively expensive in this regard, transforming the data takes orders of magnitudes longer than a scan for this format. This is due to the gathering and scattering of bits over many different words, AVX 2 provides no fast way to do this. For *SIMD-Scan* and *ByteSlice* though, the packing and unpacking operations are very fast and close to a simple copying operation. This makes it feasible to use the scan representation as the only representation and going without additional indexes.

The last improvement evaluated is block-wise scanning, where multiple columns are scanned and the switching between the columns occurs after a specific block size. This reduces the amount of intermediate results that need to be written to memory. The gain from this is not very large and can trigger effects in the processor that can massively slow down the scan as the processor fails to recognize the memory access pattern as a sequential scan.

The result of the benchmarks is that small block sizes (less than a couple of thousand elements per block) perform a few percent better than a full column scan for less than eight columns. For more columns the processor fails to recognize the scan and performance becomes extremely bad. This effect goes away for larger block sizes as the tested block size of 2^{17}. The performance advantage stays though. All in all, block-wise scanning is only recommended if it can be easily implemented in the data base system as the gains are rather small.

In conclusion, *BitWeaving/V* and *ByteSlice* are viable improvements over *SIMD-Scan* in some cases. The main advantage is visible for range predicates and large bit widths over 20 bits. For all other cases results are more mixed and can be the same or significantly worse. If a database system wants to make use of those algorithms it should only be a secondary option and be picked based on the work load.

6.1 Future Work

The most immediate open question is whether *ByteSlice* can be made as fast as *BitWeaving/V* on current or future hardware. The vertical byte format is vastly superior to the vertical bit format in terms of packing and unpacking performance.

On the scan side, evaluating more compression formats is a very wide topic. There are many different compression formats [17] that can potentially run fast enough to be only limited by memory bandwidth. There are also formats such as Simple-8 [10] that may benefit from SIMD instructions but actually implementing it on top of existing instruction sets like AVX 2 is difficult. Another possibility is changing the order of values in the column [18]. This has the disadvantage of requiring meta data to reconstruct the original order but can facilitate better compression.

Another potentially large field for future work is how to deal with different scan algorithms in a query optimizer. This ranges from reordering the columns in a query by expected selectivity based on historical data to increase early pruning probabilities to the exact point when to choose a format like *BitWeaving/V* over *SIMD-Scan* and vice-versa. A fast *ByteSlice* would make this a little simpler as the fast packing and unpacking makes switching between representations feasible.

If a vertical format is chosen as a primary representation there are also possibilities to perform aggregate operations like summation directly on the data. The same applies to joins. This would remove the need for unpacking operations for queries that do not actually materialize results from the table.

Last but not least this thesis only looked at single-core performance. To use the memory bandwidth to its full potential multiple cores are needed as a single core cannot saturate the bus entirely. For example the Haswell CPU used in the benchmarks peaks at around 11 gigabytes per second on a single core, while the specification suggests a peak bandwidth of 68 gigabytes per second so the processor is still not at its full potential.

Bibliography

[1] Yinan Li and Jignesh 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, New York, USA: ACM, 2013, pp. 289–300. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465322. URL: http://doi.acm.org/10.1145/2463676.2465322.

[2] 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: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 385–394. ISSN: 2150-8097. DOI: 10.14778/1687627.1687671. URL: http://dx.doi.org/10.14778/1687627.1687671.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[4] M. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (Dec. 1966), pp. 1901–1909. ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273.

[5] Agner Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. 2014. URL: http://agner.org/optimize/instruction_tables.pdf.

[6] Jens Teubner. *Data Processing on Modern Hardware*. University Lecture. 2014.

[7] George P Copeland and Setrag N Khoshafian. "A decomposition storage model". In: *ACM SIGMOD Record*. Vol. 14. 4. ACM. 1985, pp. 268–279.

[8] 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, Riva del Garda, Trento, Italy, August 26, 2013*. 2013, pp. 1–12. URL: http://www.adms-conf.org/2013/muller_adms13.pdf.

[9] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of the 22Nd International Conference on Data Engineering*. ICDE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 59–. ISBN: 0-7695-2570-9. DOI: 10.1109/ICDE.2006.150. URL: http://dx.doi.org/10.1109/ICDE.2006.150.

[10] Vo Ngoc Anh and Alistair Moffat. "Index Compression Using 64-bit Words". In: *Softw. Pract. Exper.* 40.2 (Feb. 2010), pp. 131–147. ISSN: 0038-0644. DOI: 10.1002/spe.v40:2. URL: http://dx.doi.org/10.1002/spe.v40:2.

[11] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. "SIMD-based Decoding of Posting Lists". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM '11. Glasgow, Scotland, UK: ACM, 2011, pp. 317–326. ISBN: 978-1-4503-0717-8. DOI: 10.1145/2063576.2063627. URL: http://doi.acm.org/10.1145/2063576.2063627.

[12] Leslie Lamport. "Multiple Byte Processing with Full-word Instructions". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 471–475. ISSN: 0001-0782. DOI: 10.1145/360933.360994. URL: http://doi.acm.org/10.1145/360933.360994.

[13] Patrick O'Neil and Dallan Quass. "Improved Query Performance with Variant Indexes". In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 38–49. ISBN: 0-89791-911-4. DOI: 10.1145/253260.253268. URL: http://doi.acm.org/10.1145/253260.253268.

[14] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. "ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 31–46. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2747642. URL: http://doi.acm.org/10.1145/2723372.2747642.

[15] Roman Dementiev et al. *Intel® Performance Counter Monitor - A better way to measure CPU utilization*. 2015. URL: http://www.intel.com/software/pcm.

[16] Daniel Lemire, Leonid Boytsov, Owen Kaser, Maxime Caron, and Louis Dionne. *The FastPFOR C++ library: Fast integer compression*. 2012. URL: https://github.com/lemire/FastPFOR.

[17] Daniel Lemire and Leonid Boytsov. "Decoding billions of integers per second through vectorization". In: *Software: Practice and Experience* 45.1 (2015), pp. 1–29.

[18] Volker Markl, M Bauer, and Rudolf Bayer. "Variable UB-Trees: an efficient way to accelerate OLAP queries". In: Bayerisches Forschungszentrum.

List of Figures

List of Tables