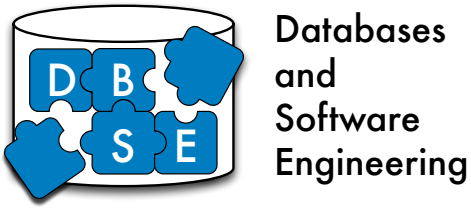
University of Magdeburg

School of Computer Science



Bachelor Thesis

ColumnWeaving: Extending Bitweaving/V for multi-column indexing

Author:

Marten Wallewein-Eising

28th May 2019

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. David Broneske

Department of Databases and Software Engineering

Dr. Veit Köppen

University Library

Wallewein-Eising, Marten:
ColumnWeaving: Extending Bitweaving/V for multi-column indexing
Bachelor Thesis, University of Magdeburg, 2019.

Abstract

Along with rapidly growing database sizes, also the requirements on modern database systems grow. For analyzing large datasets, index structures are fundamentally important, whereas the focus moves clearly to main-memory index structures, since the bottleneck from loading data from the hard-disk drive moves to loading data from main memory into the CPU cache. As one state-of-the-art main-memory index structure, we focus on Bitweaving, a column-wise storage based primary index structure that runs scans at "bare metal" speed and offers possibilities to reduce the amount of processed data, one of them called *Early Pruning*. This approach comes with one limitation in the implementation used by Bitweaving: The performance improvement using early pruning depends on the selectivity of each individual predicate, which in many cases is comparatively high, whereas considering multiple predicates together significantly reduces the selectivity.

In this work, we present ColumnWeaving, an adaption for Bitweaving/V that supports multi-column indexing and aims to benefit from the reduced selectivity of considering multiple predicates together. We introduce two different memory layouts for indexing multiple columns adapting the weaving techniques from Bitweaving/V, which we call ColumnWeaving/S and ColumnWeaving/L and present scan algorithms on both layouts. After presenting both layouts for ColumnWeaving, we evaluate the early pruning behavior using a synthetic benchmark and compare ColumnWeaving against Bitweaving/V using TPC-H queries. Our synthetic benchmark confirms, that the indexing of multiple columns together improves the early pruning behavior, whereas in the TPC-H queries, ColumnWeaving results in higher response times than expected. Consequently, we propose and evaluate a set of limitations of ColumnWeaving as possible causes for the higher response time.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Along with the steady growth of data and the demands for databases, also the technology evolved immensely in the last decades. Right after the original design of *online transactional processing (OLTP)* for database queries, the need of *online analytical processing (OLAP)* became more important [Pla09, KN11]. To this end, warehouse databases with terrabytes of records have to be scanned to fulfill the analytical requirements of modern companies.

Since the original bottleneck of loading data from a hard-drive disk (HDD) moved to loading data from main memory into CPU cache, the research focusses more on main-memory index structures [BKM08, KKN+08]. Furthermore, the exploitation of data parallelism with the approach of *singe instruction multiple data (SIMD)* highly increases the performance of full table and index-based scans [ZHF14, SGL09]. Also adapting index structures to underlying hardware restrictions like cache and register size lead to improved read performance [KCS+10].

Although all of these adaptations increase the performance of database operations, there is a need of more specific adaptations for OLAP queries to keep up with the fast evolution of requirements to those. For OLAP queries, in most cases, full table scans are performed by the database systems. However, dependent on the selectivity, an index-based scan can reach a better performance. Das et al. propose to use an index structure for very low selectivities of smaller than 2% [DYZ+15] .

Several authors show improved secondary and primary index structures for read-only operations, like Column Imprints [SK13] and Bitweaving [LP13]. Both are improved for column-based storage of tables and perform well on queries containing predicates over multiple columns, but they treat columns independently in the query execution process. As Broneske et al. [BKSS17] show, the selectivity of a query can highly decrease if predicates of multiple columns are considered together. Consequently, we want to examine if those index structures can also benefit from this observation. We focus on Bitweaving, especially Bitweaving/v, since we assume that early pruning can be improved by indexing multiple columns together.

Goal of this Thesis

The goal of this thesis is to evaluate the hypothesis that considering multiple predicates together improves the sequential scan approach of Bitweaving/V. Therefor, we present ColumnWeaving, an extension of Bitweaving/V for multi-column indexing and evaluate the performance compared to the original implementation of Bitweaving/V. To this end, we provide the following contributions to reach a comprehensive evaluation of our extension.

1. Based on the decreasing selectivity considering predicates of a query together, we define two layouts of our extension of Bitweaving/V, called *ColumnWeaving/S* and *ColumnWeaving/L*. Both layouts span over multiple columns, which enables evaluating multi-column selection predicates together.

2. Along with the layouts for multi-column selection predicate evaluation, we present algorithms to evaluate predicates on ColumnWeaving/S and ColumnWeaving/L.

3. Li and Patel introduced early pruning as an essential feature of Bitweaving that increases the performance [LP13]. Consequently, we also implement early pruning for both layouts and show how the pruning behaves over different configurations.

4. We contribute an analysis of query execution performance with the TPC-H dataset of ColumnWeaving/S and ColumnWeaving/L and the original Bitweaving/V implementation.

Structure of the Thesis

This thesis is structured as follows. We start introducing necessary terms to understand the design choices for ColumnWeaving and present Bitweaving in Chapter 2. In Chapter 3 and Chapter 4 we define both variants of ColumnWeaving along with the algorithms to evaluate multi-column selection predicates and adapted algorithms for early pruning. After presenting ColumnWeaving, we explain our evaluation setup and environment in Chapter 5 and present the results of our evaluation. We continue this thesis naming related work in Chapter 6, end with a conclusion and show steps that have to be done in the future to continue our work in Chapter 7.

2. Background

In the following, we present background information and define important terms on which we rely in upcoming chapters. Starting with an introduction of important terms, we present different parallelization approaches and introduce what the state of the art of full table scans is and which problems exist. Furthermore, we present layouts, concepts and algorithms of the index structure Bitweaving, focussing on the *Vertical Bit-Parallel* (VBP) method, early pruning and Bitweaving/V.

2.1 Important Terms

Before introducing state of the art concepts for query execution, we define a set of important terms used in the rest of this work.

CPU Cache

To load data into processor registers, the data has to be loaded into *cache* beforehand. Since loading data from main memory is an expensive operation for the CPU, because the data has to pass the CPU bus, many cycles can be saved if the data can be cached on the CPU itself. In modern processors, the cache is split into the L1, L2 and L3 cache. In Figure 2.1 we present the cache hierarchy of all these three layers. Data can be transferred only from one cache layer to the adjacent layers. Consequently, loading data into processor registers requires loading them into all 3 cache layers, starting from L_3 over L_2 to L_1. In most modern CPUs, the L_2 and L_3 caches are shared across all CPU cores, which means all cores can access the caches, whereas each CPU core has its own L_1 cache. To load data into a processor register, the CPU checks if the data is available in the cache, starting from L_1 over L_2 to L_3. With each cache layer, the available memory grows along with the access times. Hennessy and Patterson present a server architecture having 64KB L_1 cache, 256KB L_2 cache and 2-4 MB L_3 cache [HP11]. The response time for L_1 cache is 1ns, for L_2 cache 3-10ns and for L_3 cache 10-20ns.

Reaching a good cache usage is an important target for modern software. Consequently, also index structures for databases are adapted for better cache usage.

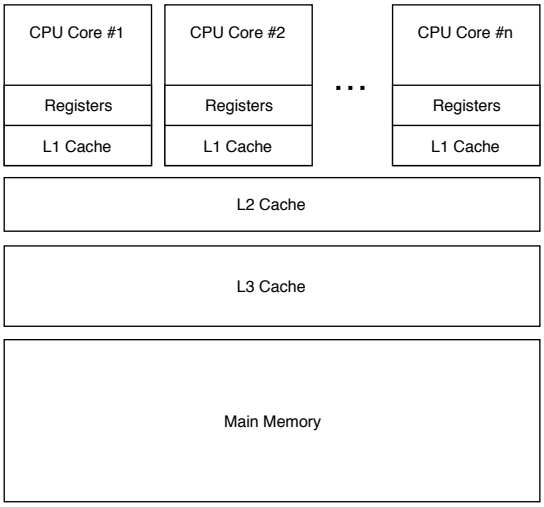


Figure 2.1: Cache hierarchy of modern processors

Kim et al. introduce FAST [KCS+10], a hardware-sensitive binary tree, adapting its node sizes to the underlying cache and register sizes. Rao et al. make B+-trees cache conscious in main memory to reach better cache usage [RR00]. Furthermore, Cha et al. evaluate cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems [CHKK01].

Cache line size

To transfer data from the L_1 cache to the processor register, the data has to be stored in a *cache line*. We define the term *cache line size* as the size of the L1 cache line of CPUs. In most x86 and x64 systems, the cache line size is 64byte, including our setup used for the evaluation.

Since the cache line size is mostly limited to 64byte, programmers aim to optimize the cache line behavior. Read and write operations perform most efficiently if the data address is a multiple of the data size, which we call *naturally aligned*. We define the insertion of *padding* between structured elements to reach their natural *data alignment*.

Cache Miss

As mentioned before, loading data from main memory into the CPU is an expensive operation. Consequently, storing data in the CPU cache leads to reduced latencies and the CPU looks always into the cache to find required data before loading it from the main memory. Since the cache size on the CPU is highly limited compared to the main memory, cache lookups often result in not finding the data in the cache. We define this incident as *cache miss*.

Cache misses may have an important influence on the program execution performance, because the data has to be loaded from the main memory over the CPU bus and this forces the CPU to wait with the current operation until the data is available. Consequently, aiming a good cache behavior and avoiding cache misses is an important target for modern programs. Skadron et al. evaluate performance

trade-offs for cache usage [SAMC99]. Veidenbaum et al. present a different approach adapting cache line size to application behavior [VTG+99]. Furthermore, Kaxiras et al. present an analytical view on cache misses [GMM97].

Processor Word

A *processor word* is a block of data whose size matches the register size of the processor. Consequently, a processor word can be processed by the processor as one unit. We differentiate between *Arithmetic Logical Unit* (ALU) words reaching from 32 to 64bit and SIMD words, which may be from 64bit to 512bit long. In Table 2.1, we present different processor word sizes depending on the system architecture and usage of SIMD extensions.

Name	Size in bits
x86	32
x64	64
SSE 1-2 (x86)	64
SSE 1-4 (x64)	128
AVX/AVX2 (x64)	256
AVX 512 (x64)	512

Table 2.1: Register sizes of different system architectures and SIMD extensions

Code Word

Since our implementation bases on Bitweaving, which processes data as encoded bit strings, we introduce the term *code*, which represents an encoded column value. Depending on the method used for encoding, the number of bits used for a code differs from the bits used to store the original column value. In most cases, compression is used to transform a column value to a code and reduce the storage size.

2.2 Parallelism Types

To reach a high level of performance, the structures presented in this work use different techniques of parallelization. In Table 2.2 we present flynns taxonomy [Fly72], which classifies computer architectures depending on data and instruction streams. The data stream as well as the instruction stream can contain a single element or multiple elements, resulting in 4 different approaches. In this work, we focus on *Single Instruction Single Data* and *Single Instruction Multiple Data*.

Single Instruction Single Data

Basically, a program is a set of instructions stored in the main memory, which are processed one after another. We define the term *Single Instruction Single Data* as executing one instruction with exactly one data item. This approach is widely used

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.2: Flynns taxonomy describing a classification of computer architectures

in modern personal computers. Although they may contain multiple cores, each core follows the SISD approach executing one instruction on a single data item per cycle.

Those instructions can depend on each other or can be independent. In one cycle, one CPU core can process at maximum one instruction, but to increase the execution speed of a program, the CPU can prepare the execution of the next instruction while processing an instruction, which we call *instruction parallelism*. In contrast to linear program flows, where instruction parallelism may highly increase the execution performance of a program, each *branch* in the program flow may lead to the case, that the CPU prepares the wrong instruction. This situation is called a *branch misprediction* and has a high impact of computing performance, since the CPU has to revert the preparation.

Branch mispredictions may reduce the execution performance of programs significantly [ESE06]. Broneske et al. present processing capabilities and describe code optimizations to exploit these capabilities [BS17a]. Furthermore, they evaluate database operations in a hardware-sensitive context [BBHS14].

Multiple Instruction Single Data

In contrast to executing one operation on one data item at once, there is also the approach of executing multiple instructions on a single data item at once, called *multiple instruction single data*. This approach is not spread widely in multi-core processors and mostly used for highly specific use cases, for example highly parallelized integration or matrix operations [KL79].

Single Instruction Multiple Data

As third block of flynns taxonomy we present the execution of a single instruction on multiple data items in one cycle, called *Single Instruction Multiple Data* (SIMD). Examples of SIMD instructions are Intels SSE or AVX extensions. Those are instructions that work on processor words with a size of a multiple of those from normal ALUs and provide an extra set of extended CPU instructions along with additional registers. In Figure 2.2 we present the instruction execution of SISD and SIMD. Whereas SISD executes an operation on one data item, SIMD executes one operation on n data items in parallel giving results for all n items in one cycle. Hence, an n-fold performance improvement has to be expected.

Considering the performance, an n-fold performance improvement can massively speed up programs. Consequently, SIMD is an often used approach to increase performance of processing huge amounts of data. Franchetti et al. present techniques for efficient utilization of SIMD extensions [FKLU05]. Although, using SIMD can

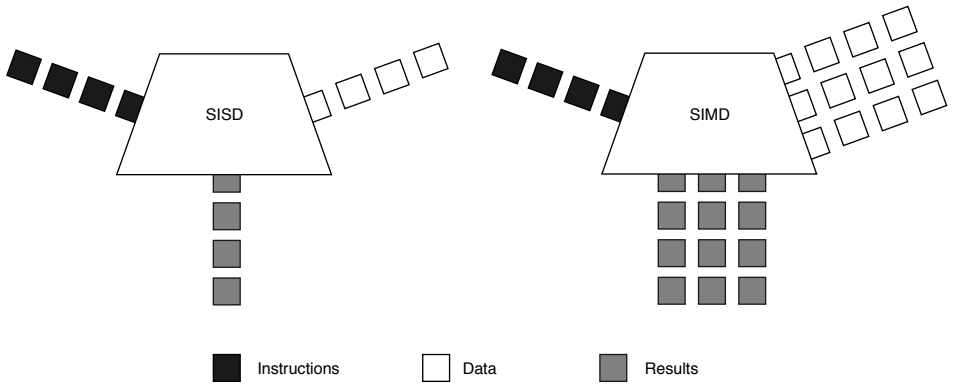


Figure 2.2: Comparison of SISD and SIMD

increase the performance up to n times, it is not always the best solution for speeding up programs. Broneske et al. evaluate use cases for SIMD usage and present situations, in which SIMD will not lead to the expected performance increase [BS17b]. Also in database systems, SIMD is often used as approach to execute performance of database operations [ZR02, PRR15].

Multiple Instruction Multiple Data

Compared to SISD mostly used in personal computers, super computers containing a huge number of cores execute multiple instructions on multiple data at once, called *Multiple Instruction Multiple Data*. Using shared memory to access the data processed by other cores, this approach allows highly parallelized work on the same data.

2.3 Database Query Processing

Database queries often written in SQL are represented as execution plan, which is basically a tree of operators connected with logical operators. Those execution plans can be optimized by database systems to increase the execution speed. Mostly, there are two different approaches for fetching rows from a table: Index-based scan vs full table scan. Both can be better in specific situations, depending on the expected number of rows that have to be checked. Index-based scans perform better for a fine granular search, full table scans may be better if all rows have to be checked. To define a value for deciding which approach may be better, we introduce the selectivity of a query.

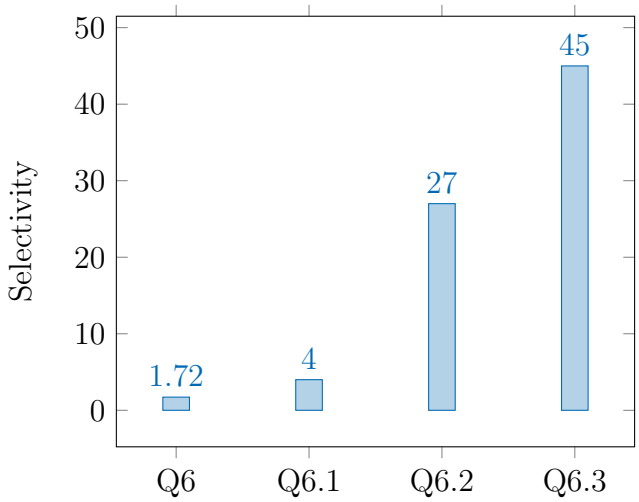
2.3.1 Selectivity

The selectivity of a query or predicate is defined as the number of tuples matching the query or predicate divided by the number of tuples checked in total. Consequently, a small selectivity means a small set of tuples matching the query. For small selectivities, an index scan may outperform a scan over the whole table, whereas for high selectivities, a full table scan may be better. Consequently, a database system can use statistics to evaluate this value and decide which approach to use for executing the given query. The selectivity of a query depends on the number of predicates

that have to be evaluated. In Figure 2.3, we present a typical TPC-H query containing multiple predicates as visualized WHERE-clause and for each part of the WHERE-clause, we show the selectivity.

```
Q6.1:   l_shipdate >= [DATE] AND l_shipdate > [DATE] + '1 year'
Q6.2:   AND l_discount BETWEEN [DISC] - 0.01 and [DISC] + 0.01
Q6.3:   AND l_quantity < [QUANTITY]
```

(a)



(b)

Figure 2.3: a) WHERE clause of TPC-H query split into three parts, b) Selectivity of all three parts and the whole WHERE clause

We decide to use TPC-H dataset because it represents real-world data used for testing index structures and database systems and it is a good basis for comparing against other state-of-the-art index structures. Since full table scans are often preferred over index structures for queries with high selectivities, we focus on queries that have low selectivity in this work. For these queries, we know the selectivity beforehand. However, in real-world applications, it is not a trivial task to evaluate the selectivity of a query to decide if a full table or index-based scan lead to better results. Consequently, several contributions were made to this research area. Chen et al. present algorithms for adaptive selectivity estimation using query feedback [CR94]. Getoor et al. propose an estimation approach for selectivity using probabilistic models [GTK01].

Since a query may contain predicates over multiple columns, we want to formally define the query as a set of predicates over multiple columns called *multi-column selection predicates*.

2.3.2 Multi-Column Selection Predicates

We define a multi-column selection predicate $MCSP$ over a subset of columns C from table T as a set of predicates P that are evaluated together. Consequently, $MCSP = \{P_1, ..., P_n\}$ with $|MCSP| \geq 1$. Each predicate $P_i = (C_i, O_i, V_i)$

is a 3-tuple containing a column $C_i \in C$, an operation $O_i \in \{=, <, \leq, \geq, >, \neq, BETWEEN\}$ and a list of constants V_i, whereas each constant $v_i \in V_i$ is a code word. As important limitation, a multi-column selection predicate must not contain two predicates having the same column. So for each column there can be at maximum one predicate in a single MCSP. Furthermore, all predicates contained in one MCSP are connected via the logical AND operator. To support a logical OR between predicates, they have to be split into multiple MCSP.

To handle all operations $o \in \{=, <, \leq, \geq, >, \neq, BETWEEN\}$ the same way, we adopt the columnar selection predicate translation proposed by Broneske et al. [BKSS17]. In Table 2.3, we present a common representation of all required operations using upper and lower bound windows. We adopt the definition *min* being the domain minimum and *max* the domain maximum of each column.

Predicate	Window
$= x$	$[x, x]$
$< x$	$[min, x - 1]$
$\leq x$	$[min, x]$
$> x$	$[x + 1, max]$
$\geq x$	$[x, max]$
$\leq x$ and $\geq y$, with $x \leq y$	$[x, y]$

Table 2.3: Columnar selection predicate translation proposed by Broneske et al. [BKSS17]

2.4 Bitweaving

Bitweaving is a framework designed to overcome the bottleneck of accessing main memory. Instead of relying only on data parallelism with SIMD, Bitweaving exploits *intra-cycle* parallelism, which means to achieve data parallelism in a single CPU word. Bitweaving focusses on increasing the speed of full table scans by compressing data in order to overcome bandwidth limitations. Thus, data can be processed at the speed of the processor core, which is called at bare-metal speed in the following. Additionally, it can be extended using data parallelism with SIMD, but it is not a mandatory part of the Bitweaving framework.

To achieve intra-cycle parallelism, Li and Patel [LP13] limit the data domain which can be processed by encoding it to fixed-size codes from 1 bit to 32bit, whereas the performance increases with smaller code sizes because of a higher grade of parallelism. Since decoding is required to get the original data out of the code words, late materialization is commonly used in the framework. Bitweaving relies on commonly used column compression methods including null suppression [FHL10], prefix suppression [ZHNB06] and order-preserving dictionary encoding [BBC+12, FML+12].

In the following, we present the two core layouts on which Bitweaving is build, the horizontal and vertical bit parallel methods.

2.4.1 Bit-parallel Methods

Fundamentally, Bitweaving provides two layouts. The horizontal bit-parallel method (HBP) is based on a row-oriented bit storage and the vertical bit-parallel method (VBP) is based on a column-oriented bit storage. In Figure 2.4, we show the layout of both methods placing codes into processor words.

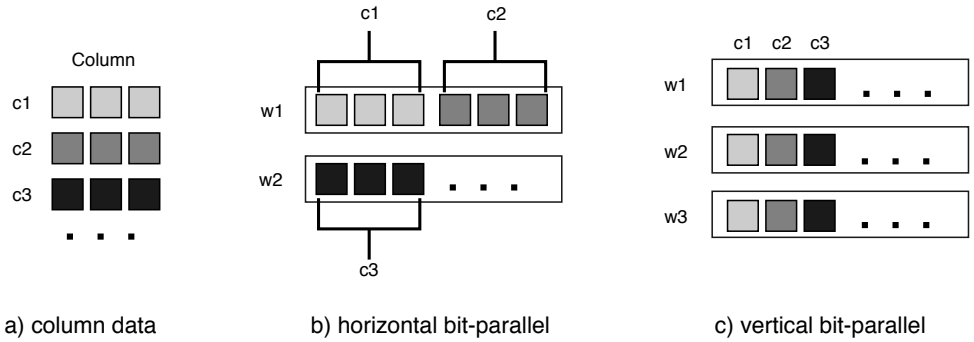


Figure 2.4: a) Sample Column containing 3 3bit codes, b) its encoding using horizontal bit parallel method, c) and vertical bit parallel method

The HBP method places codes one after another in processor words padded by separation bits until the processor word is full. In contrast, the VBP methods splits up codes and put the i_{th} bit of multiple codes into one processor word, starting from the most significant bit to the least significant bit. Since we focus on extending the VBP method, we only present the HBP method shortly to differentiate it to the vertical method. In the following, we only consider the VBP method.

2.4.2 Vertical Bit-parallel Method

VBP is inspired by the bit-sliced method [OQ97] with a different data organization around word boundaries. The original codes $c_1, ..., c_w$ are transposed into an adapted encoding and grouped into segments. Each segment contains k codes, where k represents the number of bits in each code. The k codes are transposed into a set of codes $v_1, ..., v_k$, so that the jth bit in v_i is equals to the original code c_j. The transposed words inside a segment are physically stored in continuous memory to allow hardware prefetching using a sequential access pattern.

In Figure 2.5, we give an example of the VBP storage layout for one column containing 11 3bit codes $c_1, ..., c_{11}$ and a processor word size of 8bit. Those 11 codes are divided into 2 segments containing 3 processor words per segment. Those segments do not exist in the original column, but we indicate them to clarify which codes are put into which segments in VBP. The first 8 codes fully fill the first segment in VBP and the remaining 3 codes filling up only a part of the second segment, whereas the remaining bits in segment 2 are filled up with 0s.

Compared to HBP, the VBP method does not need additional bits to separate codes. Due to the required bit extraction to store the bits in a vertical fashion, the index creation of VBP is more complex compared to HBP. Furthermore, the reconstruction of codes out of the VBP is slower compared to HBP. Whereas the code size affects

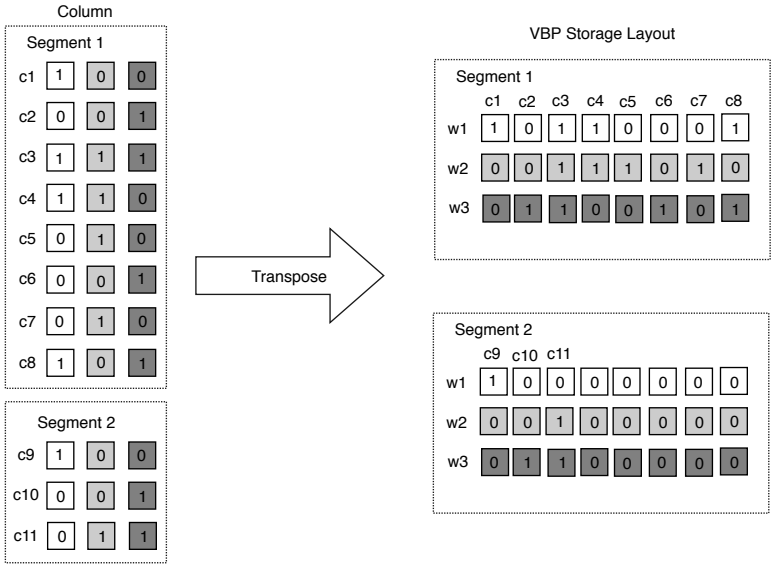


Figure 2.5: Vertical bit parallel method

the number of codes per processor word in HBP, the code size does not affect the number of codes stored together in one processor word in VBP, because in each word only the ith bit is used in the processor word. Consequently, the code size does not affect the number of codes stored in each processor word, but the number of processor words required to store a fixed size of codes. For example, considering a 64bit processor word and 8bit code size, VBP stores the ith bit of 64 codes in each processor word and requires 8 words to store 64 codes.

Since the VBP layout differs clearly from a traditional column-based storage, adapted algorithms have to be used to run columnar scans on this layout.

2.4.3 VBP Column-Scalar Scan

The VBP column-scalar scan processes the transposed codes and results in a bitvector indicating whether the column code matches the comparison condition at the specific index. To apply a columnar-scan, the transposed codes are read in their located order and compared bitwise using a specific comparison condition. Consequently, the original codes are checked against the comparison condition starting from the most significant bit until the last significant bit. In each step, a vector of w bits is processed and the ith bit of k words is compared in parallel.

Algorithm 1 shows the pseudo code for a column-scalar scan evaluating the predicate BETWEEN for two constants c_1 and c_2. As first step of the algorithm (Lines 1-12) we create two lists of literals, one list for C_1 and one for C_2. The literals $C_{11}, ..., C_{1k}$ represent the input constant C_1 in the VBP storage format. If the ith bit in C_1 is set to 1, we set C_{1i} to 1^w, otherwise to 0^w. We set the bits of $C_{21}, ..., C_{2k}$ checking the ith bit of C_2, respectively. In the second step, we iterate over all segments and evaluate all w codes of each segment in one iteration. We use the bitmask m_{gt} to indicate the codes that are greater than the constant C_1 and m_{lt} to indicate the codes that are smaller than the constant C_2. The bitmasks m_{eq1} and m_{eq2} represent the codes that are equivalent to the constants C_1 and C_2, respectively. In the

inner loop (Line 14-18), we iterate over all k bits of the codes starting from the most significant to the least significant bit. At each step $s = 1, ..., k$, we compare the literals at position s from C_1 and C_2 to the sth processor word of the current segment and update the masks m_{lt}, m_{gt}, m_{eq1} and m_{eq2}. m_{gt} is updated via the assignment of $m_{gt} := m_{gt} \vee (m_{eq1} \wedge \neg C_{1i} \wedge s.v_i)$, where $s.v_i$ is the ith bit of the current processor word at step s. Consequently, m_{gt} is set to 1 at a specific position i, if the corresponding bit of the constant C_2 is 0 and $s.v_i$ is set to 1. m_{gt} is set to 1 at bit position i, if the corresponding bit of the constant C_1 is set to 1 and $s.v_i$ is 0. m_{eq1} and m_{eq2} are updated for the codes that differ from the constants C_1 and C_2 at bit position i. As last step, a conjunction of m_{gt} and m_{lt} is appended to the result bit vector to fulfill the predicate $C_1 < c < C_2$, whereas this line can be replaced for supporting other predicates, like $m_{gt} \wedge m_{lt} \vee m_{eq1} \vee m_{eq2}$ for $C_1 \leq c \leq C_2$.

Algorithm 1: VBP column-scalar scan for BETWEEN predicate

Input: Predicate $C_1 < c < C_2$ for column c
Result: V_{Out}

```
1  for i:= 1...k do
2      if i-th bit of C_1 set to 1 then
3          C_1i = 1^w
4      else
5          C_1i = 0^w
6      end
7      if i-th bit of C_2 set to 1 then
8          C_2i = 1^w
9      else
10         C_2i = 0^w
11     end
12 end
13 for each segement s in c do
14     m_lt := m_gt := 0
15     m_eq1 := m_eq2 := 1^w
16     for i:= 1...k do
17         m_gt := m_gt ∨ (m_eq1 ∧ ¬C_1i ∧ s.v_i)
18         m_lt := m_lt ∨ (m_eq2 ∧ C_2i ∧ ¬s.v_i)
19         m_eq1 := m_eq1 ∨ ¬(s.v_i ⊕ C_1i)
20         m_eq2 := m_eq2 ∨ ¬(s.v_i ⊕ C_2i)
21     end
22     append m_gt ∧ m_lt to V_out
23 end
```

Processing multiple tuples in parallel starting from the most significant bit leads to an important optimization regarding the number of codes that have to be processed: If all currently processed bits do not match the comparison condition, the remaining codes of the segment can be skipped. This idea is called early pruning.

2.4.4 Early Pruning

As mentioned before, CPU cycles and loading data into CPU cache are important performance factors. Consequently, modern index structures aim to reduce them.

With early pruning, a fine granular reduction of processing codes is possible with skipping the rest of a segment if all most significant bits do not match the comparison condition.

In Figure 2.6, we show the application of early pruning on comparing 3 VBP words w_1, w_2, w_3 against the transposed Constants C_{11}, C_{12} and C_{13} out of the input constant c. Starting from the first bit, we show the current word w_i compared to the transposed constant C_{1i} compared for equality in the result masks m_{eq}. After comparing the first bit, m_{eq} contains two codes that match, but after comparing the w_2 and C_{12}, all bits of m_{eq} are 0. Consequently. none of the codes match the constant after comparing the second bit and we can skip checking the last bit of all codes. We call this skip of operations early pruning.

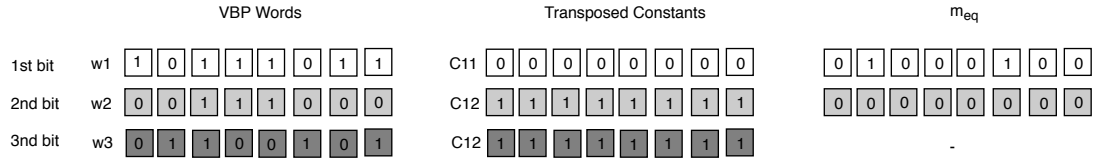


Figure 2.6: Early pruning on VBP

Li and Patel introduce the *pruning probability* [LP13] on a segment containing m codes, whereas $m = w * f$ is the fill factor f of the segment multiplied with the processor word size w. They define the pruning probability $P(b)$, depending on the number of most significant bits b, in Equation 2.1.

$$P(b) = (1 - (\frac{1}{2})^b)^m = (1 - (\frac{1}{2})^b)^{w*f} \tag{2.1}$$

Furthermore, Li and Patel evaluate the pruning probability with different fill factors and most significant bits, concluding that the pruning probability reaches up to 100% for at maximum 12 most significant bits, for lower fill factors, the probability of 100% is already reached with 4-8 bits.

Although the number of codes that have to be processed by the CPU in VBP can be reduced with early pruning, the codes are loaded into the CPU cache, since cache lines are loaded fully from memory. Consequently, regarding the number of processed codes, a bad cache behaviour occurs. To overcome this, Bitweaving/V was introduced.

2.4.5 Bitweaving/V

Bitweaving/V is an adaption of VBP locating the most significant bits of codes together and cutting off the less significant bits. Cutting off means to locate the bits in another memory location, which does not directly follow the location of the most significant bits. It has three key features, which lead to better scan performance: 1) The storage layout extends the VBP introducing *bit groups*, in which the most significant bits of the transposed codes are located in a sequential order for better cache behaviour; 2) Because of the extended storage layout, early pruning can be

applied; 3) SIMD instructions can be used to extend Bitweaving/V to improve scan performance.

In Figure 2.7 we show the extended storage layout of Bitweaving/V compared to original VBP. In this sample, we put the first 3 words of segment 1 into the first bit group and perform a so called *cut-off*, which means that after these 3 words containing the most significant bits of segment 1, the first 3 words of segment 2 are located directly after. The middle significant bits of segment 1 are stored at the beginning of the second bit group. Depending on the pruning probability and the number of most significant bits, words of the bit groups 2 and 3 may not be processed at all, leading to a highly increased cache line usage.

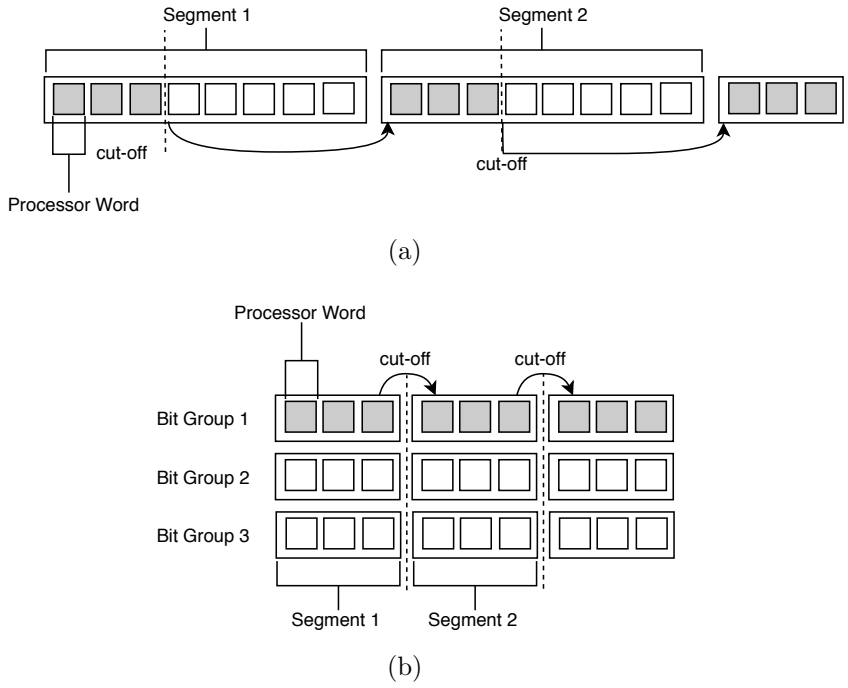


Figure 2.7: Early pruning on a) VBP and b) BitWeaving/V

Bitweaving/V comes with a significant benefit compared to VBP: The cut-off used to create bit groups leads to highly improved cache line usage if early pruning can be applied. As mentioned before, with early pruning, not all words have to be processed, if the most significant bits do not match the predicate. In VBP, although we can skip words applying early pruning, they are located next to the words containing the most significant bits. Consequently, they are loaded into the CPU cache without being processed and lead to bad cache behavior. The grouping of bits in Bitweaving/V, depending on the significance of the bits, leads to store the less significant bits in another memory location and avoid them to be loaded into the cache if they could be pruned. In the best case, only the words of the first group have to be checked and the words located in the remaining groups are not loaded into the cache.

After presenting the storage layout of Bitweaving/V, we focus on the adapted column-scalar scan implementation. In Algorithm 2, we show the pseudo code for

columnar-scan on Bitweaving/V using the BETWEEN operation including early pruning.

Algorithm 2: Bitweaving/V column-scalar scan for BETWEEN predicate

Input: Predicate $C_1 < c < C_2$ for column c
Result: V_{Out}

```
1  Initialize C1 and C2 (analogue to Lines 1-10 in Algorithm 1)
2  for each segement s in c do
3      m_lt := m_gt := 0
4      m_eq1 := m_eq2 := 1^w
5      for g:= 1...⌊k/B⌋ do
6          if m_eq1 == 0 ∧ m_eq2 == 0 then
7              break;
8          else
9              for i:= gB + 1 ... min(gB + B, k) do
10                 m_gt := m_gt ∨ (m_eq1 ∧ ¬C_1i ∧ s.v_i)
11                 m_lt := m_lt ∨ (m_eq2 ∧ C_2i ∧ ¬s.v_i)
12                 m_eq1 := m_eq1 ∨ ¬(s.v_i ⊕ C_1i)
13                 m_eq2 := m_eq2 ∨ ¬(s.v_i ⊕ C_2i)
14             end
15         end
16     end
17     append m_gt ∧ m_lt to V_out
18 end
```

At first, the initialization of the literal bits for the constants C_1 and C_2 is the same as shown in Algorithm 1. Furthermore, we keep the iteration over all segments, as it can be seen in line 2. In line 5, we start iterating over all used bit groups. Defining k as the number of words in each segment and B as the size of each bit group, we have to iterate over $\lfloor \frac{k}{B} \rfloor$ groups. The fist step before processing the words of one bit group is to check if early pruning can be applied. This is done with the condition $m_{eq1} == 0 \wedge m_{eq2} == 0$ in line 6. If both equal masks have all bits off, we have no code matching the constants and we can continue with the next segment. In line 9, we iterate over the number of words contained in the current group. Full groups contain always k words, but the last group may not be full and consequently we need to perform edge checking to get the real number of words in the last group.

Li and Patel observe a performance increase by up to 40% using the advanced column-scalar scan of Bitweaving/V compared to the VBP implementation [LP13]. Furthermore, applying early pruning to the Bitweaving/V scan implementation decreases the number of cycles per tuple.

3. ColumnWeaving/S

In this section, we introduce *ColumnWeaving/S*, an adapted memory layout for Bitweaving/V and a set of algorithms to evaluate multi-column selection predicates on this layout, where the S stands for the slim memory footprint. In Chapter 4 we will present *ColumnWeaving/L*, which has a larger memory footprint because of additional bits placed between the codes. The core goal of ColumnWeaving is to keep benefits from Bitweaving/V and to make use of a reduced selectivity evaluating predicates on multiple columns together.

We organize this section as follows: At first, we introduce the memory layout of ColumnWeaving/S. Secondly, we define a set of algorithms to evaluate multi-column selection predicates on the presented spanning techniques. As next step, we present an extended version of early pruning and present how the number of processed codes can be reduced applying it on the memory layout.

3.1 Storage Layout

To index multiple columns with the benefits of Bitweaving/V, we adopt storing codes from the most significant bits to the least significant bits using an alternating order of bits of all indexed columns. In Figure 3.1, we present the storage layout of ColumnWeaving/S on an example with 2 columns C and D for the first 8 codes. Each column contains a segment of 8 codes, which are transposed into one segment of 6 processor words. The word w_1 holds the most significant bit of the codes c_1 to c_4 of column C and d_1 to d_4 of column D, whereas the codes are treated in an alternating fashion, resulting in w_1 containing the first bit of the mentioned codes in the following order $c_1, d_1, c_2, d_2, c_3, d_3, c_4, d_4$. w_2 holds the middle significant bits of the same codes of w_1 in the same order. With w_1, w_2, w_3, the codes of both columns of the first 4 tuples are stored in the layout of ColumnWeaving/S. Starting with the next 4 codes, w_5 holds the first bit of $c_5 - c_8$ and $d_5 - d_8$ in the same order and w_6, w_7 the middle and last bit, respectively.

In our example, both columns have the same code size, which is 3. In most cases, columns contained in a query must not have the same code size. ColumnWeaving/S

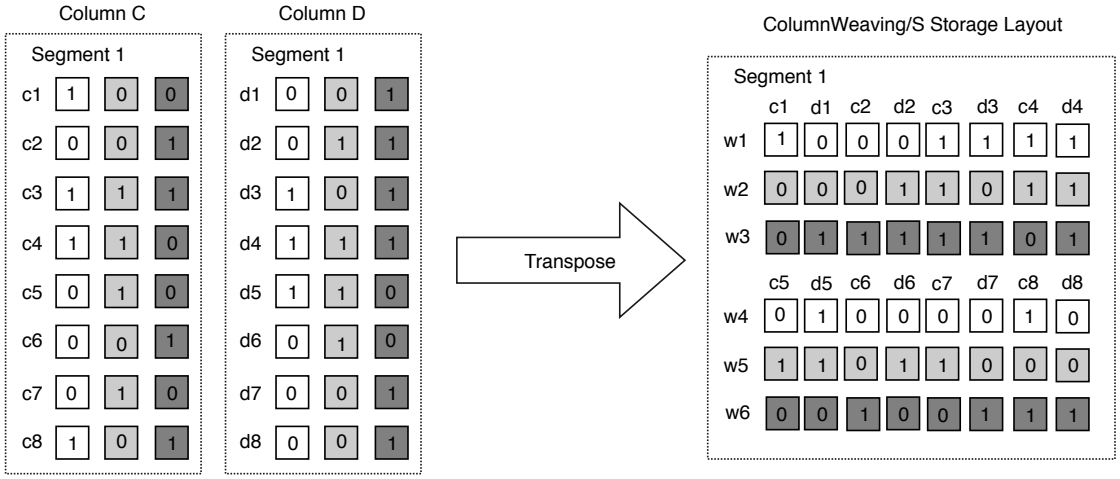


Figure 3.1: ColumnWeaving/S Storage Layout

examines the largest code size out of all indexed columns and fills up remaining bits of processor words with 0s. This will lead to many unused bits the higher the difference of code sizes is in all indexed columns. A more efficient proposal is to make a cut-off after the smallest code size and store the remaining bits of all columns at the end of the index structure. We plan to evaluate this layout in future work. For different number of codes per column, ColumnWeaving/S also fills up remaining codes with 0s. Also adopted from Bitweaving/V, we group bits depending on their significance bit groups, to reach better cache line performance.

3.2 Worst Case Storage Consumption

As first step to transpose the codes into the ColumnWeaving/S storage layout, we examine how many codes of each column we can put into each processor word, which we call N_{tb}. Since N_{tb} depends on the number of columns and the processor word size, we calculate it with the following formula Equation 3.1

$$N_{tb} = \lfloor \frac{|W|}{NUM\_COLUMS} \rfloor \tag{3.1}$$

In Table 3.1, we show the number of codes indexed per processor word depending on the number of indexed columns and processor word size. Compared to Bitweaving/V, which fills up each processor word completely, indexing multiple columns lead to unused bits. Consequently, we add the number of unused bits N_{ub} for each configuration. The code size of the columns has no effect on N_{tb}, since N_{tb} covers only the ith bit of each code. Using N_{tb} and N_{ub}, we calculate the number of words used by ColumnWeaving/S W_c at index creation time with Equation 3.2.

$$W_c = \lceil \frac{CODE\_COUNT}{N_{tb}} \rceil * CODE\_SIZE + \lceil \frac{CODE\_COUNT}{|W| - N_{ub}} \rceil \tag{3.2}$$

Basically, the formula of W_c consists of two essential parts. The first part calculates the number of words used to store the transposed codes without considering the

unused bits N_{ub}. Dividing $CODE\_COUNT$ by N_{tb} results in the factor, how many words are required to store 1bit of all codes. To get the number of words used to store all bits of the codes, we multiply this factor with the code size. The second part of the formula calculates how many additional words are required to store the unused bits. Calculating $|W| - N_{ub}$, we get the number of used bits per processor word. Dividing $CODE\_COUNT$ by this number results in the number of additional words required by ColumnWeaving/S due to unused bits.

Considering $W_c = 689$ for 5 columns and 64 bit processor word width, ColumnWeaving/S has $689 - 640 = 49$ unused processor words, which results in 3136 unused bits overall.

# Columns	|W|	N_{tb}	N_{ub}	W_c	Bitweaving/V (W_c)
2	64	32	0	256	256
3	64	21	1	400	384
4	64	16	0	504	504
5	64	12	4	689	640

Table 3.1: Number of codes indexed per processor word with different number of columns, processor word size and the resulting word count W_c using 1000 codes of 8bit for all columns compared to the number of words used by Bitweaving/V

3.3 Column-scalar Scans

In Chapter 2 we defined the multi-column selection predicate, which we use for our column-scalar scan implementation for ColumnWeaving/S. We evaluate one multi-column selection predicate per scan, which we transpose into a specific layout before executing the scan. Using this layout, we adopt the original Bitweaving/V column-scalar scan algorithm for ColumnWeaving/S. Consequently, we focus on the transposition of the multi-column selection predicate.

In Bitweaving/V, k literals are created for the constants C_1 and C_2 for the BETWEEN scan. In ColumnWeaving/S, for each column $c_i \in C$, we create l literals and perform disjunction operations on them to result again in two constants C_1 and C_2 that we use to apply the Bitweaving/V column-scalar scan operation. To disjunct all l literals for each indexed column, we introduce *spanning*. We define spanning as a set of bitmasks $S = \{S_1, ..., S_i\}$ indicating for each column c_i which bits this column uses in each processor word. The spanning itself does not depend on the code size but only on the number of indexed columns.

In Figure 3.2, we present a sample spanning for ColumnWeaving/S indexing 3 columns. S_1 indicates the bits to process for the predicate containing the first column, S_2, S_3 respectively. The spanning is evaluated at index creation time and

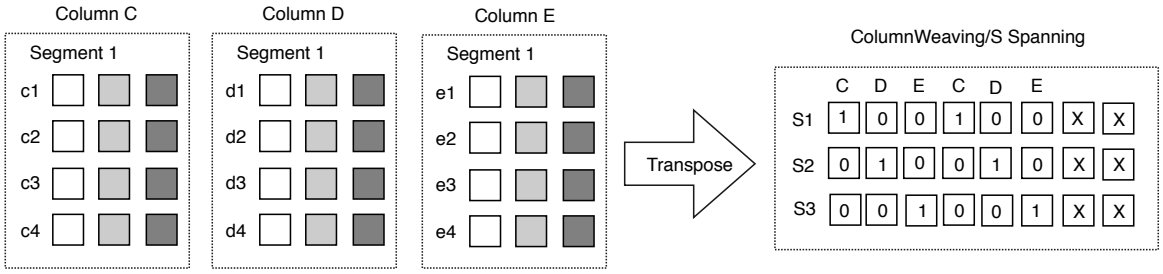


Figure 3.2: ColumnWeaving/S Spanning

remains the same for all executed scan operations. In Algorithm 3, we present ColumnWeaving/S column-scalar scan operation for the BETWEEN operation.

Algorithm 3: ColumnWeaving/S column-scalar scan for BETWEEN predicate

Input: $MCSP = \{P_1, ..., P_n\}$ with $P_l = (C_l, O_l, [V_{1l}, V_{2l}])$, Spanning $S = \{S_1, ..., S_n\}$

Result: V_{Out}

```
1  for i:= 1...k do
2      C_1i = 0^w
3      C_2i = 0^w
4      for l:= 1...n do
5          if i-th bit of V_1li set to 1 then
6              C_1i = C_1i ∨ (1^w ∧ S_l)
7          else
8              C_1i = C_1i ∨ (0^w ∧ S_l)
9          end
10         if i-th bit of V_2li set to 1 then
11             C_2i = C_2i ∨ (1^w ∧ S_l)
12         else
13             C_2i = C_2i ∨ (0^w ∧ S_l)
14         end
15     end
16 end
17 Bitweaving/V column-scalar scan (analogue to lines 2-20 in Algorithm 2)
```

In contrast to Bitweaving/V, the input to the column-scalar scan algorithm is not one predicate containing up to 2 parts (depending on the operator), but a multi-column selection predicate along with a spanning for all indexed columns. As defined in Chapter 2, each column can occur in only one predicate in a multi-column selection predicate. To support queries that have multiple predicates referring to one column, we apply multiple scans instead of considering them together. In future work, we plan an evaluation technique for getting the optimal set of MCSP out of a query to process with ColumnWeaving/S.

The second input parameter is the spanning created for each ColumnWeaving/S instance. To adopt the original Bitweaving/V scan algorithm, we transform the input MCSP into two lists of constants C_1, C_2 using the spanning. Again, for each ith bit, a mask for C_{1i} and C_{2i} is created. To create C_1, we iterate over all i bits and

for each bit, we iterate over all lower-bound values in V_1 and perform bitwise AND with the spanning S_l for the column depending on the current value. If the ith bit of the lth predicate in V_1 is set to 1, which we call V_{1li}, we perform $C_{1i} = C_{1i} \vee (1^w \wedge S_l)$ to update C_{1i} with the literal bits for the ith bit of the lth literal matching the spanning for the current predicate. To create C_2, we perform the same operations with V_2, respectively. After creating the two lists of constants C_1, C_2, we can perform the original Bitweaving/V scan on our input MCSP.

3.4 Early Pruning

In Bitweaving/V, the early pruning applies if the whole m_{eq} mask is set to 0. In our default scan implementation for ColumnWeaving/S, we adopt this early pruning logic, which comes with a huge drawback for indexing multiple columns: Checking for $m_{eq} = 0$ will only apply early pruning if the ith bit of the contained codes in the current processor word differs from the scan constants for all columns, which means that all predicates of the MCSP do not match the current tuple, which we call the predicate *fails*. If we consider predicates with highly different selectivities, we aim to prune already when only one predicate of the MCSP fails, instead of waiting for all predicates to fail.

To clarify this idea, we consider the 8bit processor word $w = 11011110$ as input for the early pruning of ColumnWeaving/S indexing 4 columns. Consequently, the first 4bit of w represent the result of executing 4 predicates on tuple t_1, the last 4bit represent the result for t_2. For both tuples t_1, t_2, one predicate failed. For t_1 the third predicate fails and for t_2 the fourth. The early pruning idea of ColumnWeaving is, to transform w into $w' = 00000000$ and to apply Bitweavings early pruning, because at least one predicate fails for all tuples in w. This transformation process consists of two steps.

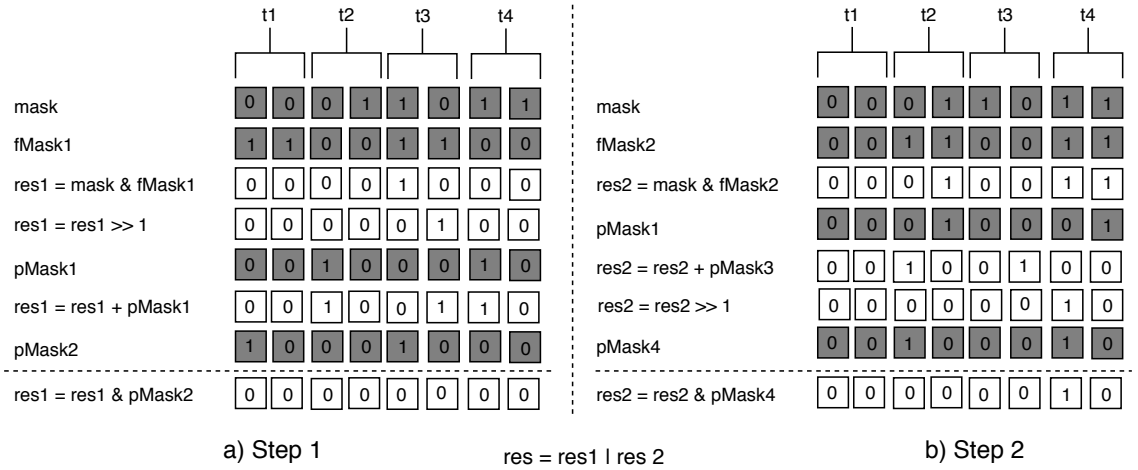


Figure 3.3: ColumnWeaving/S Early Pruning, consisting of the two steps a) step 1, b) step 2, c) connected with bitwise *OR*

In Figure 3.3, we present the two steps of ColumnWeaving/S to apply of early pruning. We split our early pruning implementation into two steps to consider the bits of the odd and even tuples in each processor word independently. This allows

us to perform add operations without having overflows that change the bits of the neighbor tuples bits (e.g. tuple t_2 has the neighbors t_1 and t_3). Consequently, we can perform the early pruning with a fixed size of masks instead of depending on the number of indexed columns. The aim of our early pruning implementation is to transform the bits of all tuples t_i to 0s if they are not all set to 1, because one bit of a tuple represents one predicate and all predicates have to match the scan constants. The early pruning algorithm gets the *mask* processor word as input, indicating the results of the current scan iteration. In our example, only in tuple $t4$ all predicates match, because all bits of t_4 are set to 1. Consequently, the algorithm should set all bits to 0 except for the first bit of t_4. For further processing, we aim to have only one bit representing the result for each tuple, consequently we only set the first bit of t_4 to 1. For each step of the early pruning algorithm, we need two filter masks $fMask1, fMask2$ and 2 pruning masks $pMask1, pMask2$. Since these masks do not depend on the current processor word or on the stored codes, we can define them at index creation.

Step 1: Odd tuples

As first part of step 1, we filter the bits of all odd tuples t_1, t_3 performing $mask$ & $fMask1$. As next part, we shift the result to left by 1bit, to avoid overflows for the first tuple. In the third part, we add the pruning mask $pMask1$ to make use of the overflow for the tuples where all bits are set to 1. As last part, we apply bitwise AND with $pMask2$ to the result to filter out the tuples having the first bit set to 1. The last part is important to filter all 1s that are not located on correct slots. For example, having 2 columns indexed and a tuple with 1100 as the result for a scan iteration, performing the left shift results in 0110. Adding the pruning mask $pMask1$ will result in 1000, indicating the correct result for this tuple.

Step 2: Even tuples

To handle all even tuples t_2, t_4, the second step applies nearly identical operations except for switching the order of the shift and add operation. After executing both steps, we perform a bitwise OR operation on both results to get the pruned mask out of the input mask.

Compared to Bitweaving‘s early pruning, this approach comes with benefits and drawbacks. Our early pruning implementation for ColumnWeaving/S consists of two steps with a set of bitwise and arithmetic operations. Since early pruning is applied very often in the scan execution process, this complex pruning algorithm will increase the response time if the pruning probability is low. Compared to the nearly effortless early pruning of Bitweaving/V, we assume that our early pruning gives better results for higher code sizes due to the complexity of the pruning implementation.

Furthermore, to apply early pruning, all tuples of the input word must have at least one predicate that failed. Considering 3 columns and 64bit processor words, this would mean 21 tuples must have at least one predicate failed for early pruning. We assume that similar data is stored next to each other in each column in real-world data, consequently, the chance to have all tuples failing at least one predicate is sufficiently high.

3.5 Result Shrinking

As last step of the column-scalar scan iteration of ColumnWeaving/S, we have to transform the result to achieve a representative form of it, for which we use the *bitvector* of Bitweaving. This bitvector is an array of processor words containing scan results for each tuple at a specific index. Considering 64bit as processor word size, the first word of the bitvector contains the scan results of the first tuples 1-64 (starting at index 0), the second word containing the scan results of the tuples 65-128, and the following words, respectively. Since ColumnWeaving needs more words to store the data because codes of multiple columns are merged together, we have to combine the results of scan iterations to match the result specification of the bitvector.

At each iteration step, we use the *pext_u64* operation [Int] to extract the results and to group them before calculating the position in the bitvector word, at which we have to write the result of the current iteration. We define the number of words per code block W_{cb} as unit to represent how many processor words we have to shrink into one result word. We calculate W_{cb} using the following equation Equation 3.3.

$$W_{cb} = \lfloor \frac{|W|}{N_{tb}} \rfloor \tag{3.3}$$

In Figure 3.4, we present how we combine those scan results into a result word. Our result shrinking implementation consists of 3 steps. At first we apply the pext_u64 with a predefined bitmask on the intermediate scan result to extract the bits representing the result of each tuple and to shrink the result. As second step, we calculate $shiftCount$, which defines how many positions we have to shift the shrinked result to match the correct position in the result word. As third step, we shift the extracted and shrinked bits according to $shiftCount$ to the correct position.

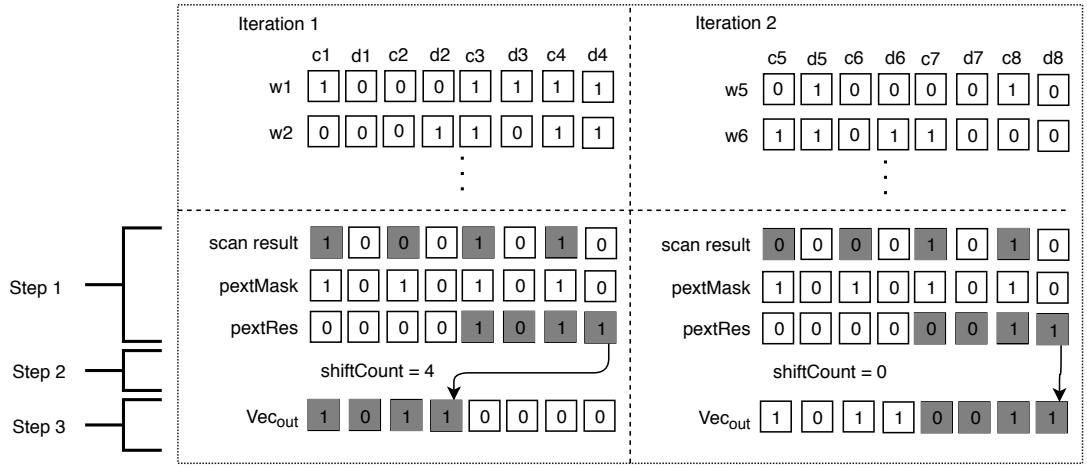


Figure 3.4: ColumnWeaving/S Result Shrink implementation

To clarify this approach, we consider the example of ColumnWeaving/S indexing 2 columns using 8bit code size and 8bit processor words shown in Figure 3.4. For this setup, $W_{cb} = 2$, because we can store bits of 4 tuples in a processor word.

Consequently, we have two iterations where we have to shrink the scan result into one 8bit result word. Furthermore, $pextMask = 10101010$ for this setup. The first result shrinking iteration results in $shiftCount = 4$ to fill up the first half of the result word with the extracted result, whereas for the second iteration, $shiftCount = 0$ to fill up the second half of the result word. In the following, we present the 3 steps of our result shrinking approach in detail.

Applying pext_u64

To extract and shrink the tuples matching the query, we define a bitmask indicating which positions represent the scan results, which we call *pextMask*. For each bit set to 1 in the pextMask, we have to extract the bit of the scan result at the corresponding position and store the bit from right to left in the *dest* bitmask. The pext_u64 instruction from Intel intrinsics implements exactly this procedure. In Algorithm 4, we present the implementation of pext_u64.

Algorithm 4: pext_u64 algorithm

Input: pextMask p, scan result res
Result: $dest$

```
1 i := 0, j := 0
2 while i < |res| do
3 |   if p_i == 1 then
4 |   |   dest_j = res_i
5 |   |   j := j + 1
6 |   i := i + 1
7 end
```

The pext_u64 algorithm gets a bitmask $pextMask$ and a scan result res as input to extract and to shrink the bits as described above. The result is stored in $dest$. In line 2, an iteration over all bits of res starts and in each iteration i, the ith bit of $pextMask$ is checked. If this bit is set to 1, the ith bit of res is set to the position j of $dest$. Then j is incremented.

The $pextMask$ required to execute this algorithm depends on the number of indexed columns. Consequently, it can be defined at index-creation time and reused for each scan iteration.

Calculate shiftCount

As second step, we have to calculate the position to which we have to shift the extracted and shrinked results, so the extracted bits are stored to the correct position representing the tuple ids. We calculate shiftCount using the following equation Equation 3.4.

$$shiftCount = N_{tb} * (W_{cb} - 1 - (s_{id} \mod W_{cb})) + N_{ub} \tag{3.4}$$

To calculate shiftCount, we start multiplying N_{tb} with a term representing the index of the current word reaching from 0 to $W_{cb} - 1$. This term is used as position

corresponding to W_{cb} to which we have to shift the extracted result, and multiplied with N_{tb}, we get the exact number of how many bits we have to shift if we do not consider the unused bits. Since the unused bits are defined per processor word, we only need to add them to the rest of the calculation to respect them for calculating shiftCount.

Shift the extracted result

As last step, we need to shift the extracted result res with $shiftCount$ bits to left. With this operation, the result shrinking approach is complete. As seen in the pext_u64 algorithm and the equation for shiftCount, the result shrinking process is quite complex. Since this process is executed many times in the scan execution, we plan to test an alternative implementation without the pext_u64 instruction and the complex calculation of shiftCount to improve performance in Chapter 7.

3.6 Summary

In this section, we presented ColumnWeaving/S, an extension of Bitweaving/V to index multiple columns using the memory layout of Bitweaving/V. The memory layout of ColumnWeaving/S stores bits of codes from multiple columns together in processor words, adapting vertical bit packing and bit grouping from Bitweaving/V. Compared to Bitweaving/V, which can fully pack processor words with column codes, ColumnWeaving/S memory layout requires additional bits depending on the number of indexed columns. Furthermore, we present a scan algorithm for ColumnWeaving/S along with an adapted early pruning algorithm that is capable of applying early pruning if at least one predicate for each tuple fails. As last part of this chapter, we present our result shrinking approach to transform the results of scan operations into a bitvector representation.

4. ColumnWeaving/L

In this chapter, we introduce *ColumnWeaving/L*, a second variant of ColumnWeaving, and present core differences to ColumnWeaving/S. The aim of ColumnWeaving/L is to reduce the effort of the early pruning algorithm shown in Chapter 3 adding an extra bit to processor words for each tuple. According to ColumnWeaving/S, we present the storage layout, column-scalar scan execution, early pruning and result shrinking, but we focus only on differences to ColumnWeaving/S.

4.1 Layout

To reduce early pruning effort, we add for the ith bit of each tuple an additional bit that allows us to apply early pruning in one step instead of two steps. In Figure 4.1, we present the storage layout for ColumnWeaving/L indexing two columns. We tag the additional bits with A and unused bits with X. For the ith bit of each tuple, starting with the first bit of c_1 and d_1, we add an additional bit as prefix, continuing with the first bit of c_2 and d_2 with the additional bit prefix, until the processor word is exhausted. The additional bits are located before the ith bit of all columns of a tuple. This bit indicates the result of executing the multi-column selection predicate on a set of tuples in ColumnWeaving/L.

This approach leads to a significantly higher memory consumption for ColumnWeaving/L depending on the number of indexed columns. As seen in Figure 4.1, in each of the 8bit processor words, we have 2 unused bits. In addition to the two unused bits, we have 2 additional bits to indicate the result for the tuples stored in each processor word. Consequently, in this configuration, we have only a usage of 50% of each processor word for the transposed codes and we need 12 processor words to store the 8 codes of both columns, which is 2 times the amount of words required by ColumnWeaving/S for storing the same amount of codes. To further evaluate this approach regarding used memory, we evaluate the worst case memory consumption for a varying number of columns.

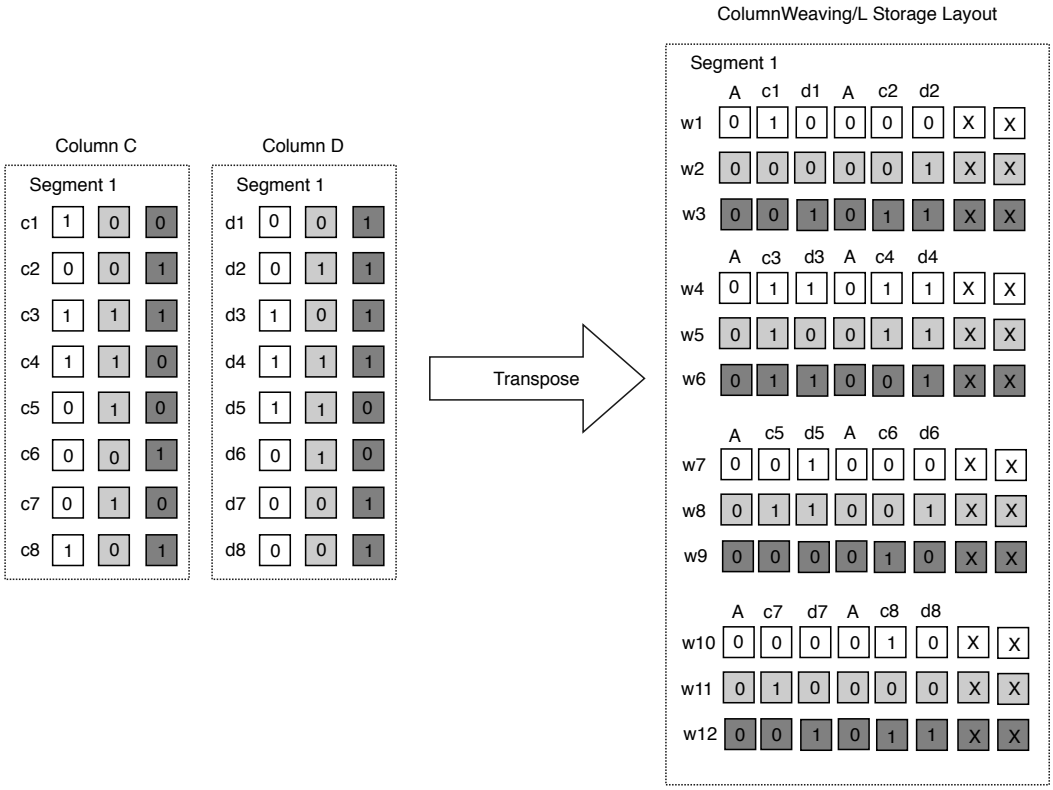


Figure 4.1: ColumnWeaving/L Storage Layout, A additional Bits, X unused bits

4.2 Worst Case Storage Consumption

Compared to ColumnWeaving/S, indexing 2 columns with 8bit processor words results in 12 processor words required to store 8 tuples instead of 6 words used by ColumnWeaving/S. Consequently, we define N_{tb} for ColumnWeaving/L in Equation 4.1.

$$N_{tb} = \lfloor \frac{|W|}{NUM\_COLUMS + 1} \rfloor \tag{4.1}$$

The calculation of the number of used words W_c, we use the same equation as for ColumnWeaving/S, again shown in Equation 4.2.

$$W_c = \lceil \frac{CODE\_COUNT}{N_{tb}} \rceil * CODE\_SIZE + \lceil \frac{CODE\_COUNT}{|W| - N_{ub}} \rceil \tag{4.2}$$

In Table 4.1, we show the number of codes indexed per processor word depending on the number of indexed columns and processor word size. Due of the additional bit, ColumnWeaving/L needs more words to store the same amount of codes than ColumnWeaving/S. Depending on the number of columns indexed and the processor word size, ColumnWeaving/L may fully use the processor word and reach the same storage consumption as ColumnWeaving/S. Considering $W_c = 817$ for 5 columns and 64bit processor word width, ColumnWeaving/S has $817 - 640 = 177$ unused processor words, which results in 11328 unused bits overall.

Compared to ColumnWeaving/S, the additional bits have a high impact on storage layout and number of words required to index the same number of codes. This impact becomes smaller, the more columns we index in ColumnWeaving/S. Consequently, for more indexed columns, we achieve less space consumption through the additional bits. Furthermore, the number of unused bits is in most cases also higher than for ColumnWeaving/S, which also leads to higher space consumption.

# Columns	|W|	N_{tb}	N_{ub}	W_c	ColumnWeaving/S (W_c)	Bitweaving/V (W_c)
2	64	21	1	400	256	256
3	64	16	0	504	400	384
4	64	12	4	689	504	504
5	64	10	4	817	689	640

Table 4.1: Number of codes indexed per processor word with different number of columns, processor word size and the resulting word count W_c using 1000 codes of 8 bit for all columns compared to the number of words used by Bitweaving/V

4.3 Column-scalar Scans

ColumnWeaving/L uses the same scan algorithm presented for ColumnWeaving/S, but with different spanning. In Figure 4.2, we present a sample spanning for ColumnWeaving/L indexing 3 columns. Again, we tag the additional bits with an A. Compared to ColumnWeaving/S, the spanning described in Figure 4.2 can fully use the 8bit processor word size, whereas this depends on the number of indexed columns, too.

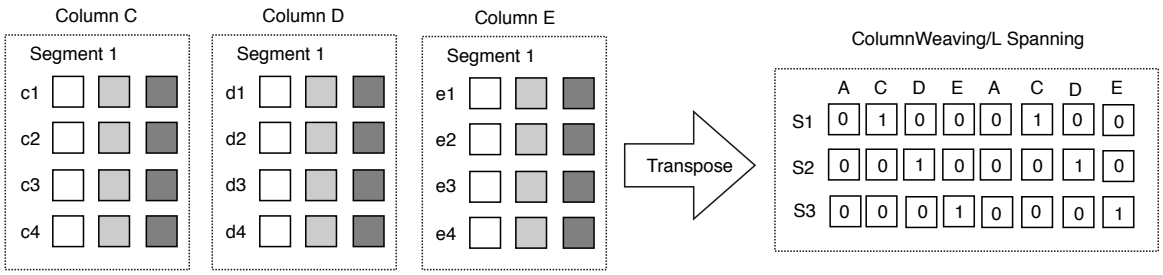


Figure 4.2: ColumnWeaving/L Spanning

To execute the scan algorithm, we perform the same transformation of the input codes shown in Algorithmn 3. Using the transformed codes, we can execute the original Bitweaving/V scan implementation with our adapted early pruning approach for ColumnWeaving/L. Since we have significantly more words to store the same number of codes compared to ColumnWeaving/S, the column-scalar scan requires more iterations to be executed. We assume that the reduced complexity of early pruning for ColumnWeaving/L will compensate the higher number of words.

4.4 Early Pruning

Using the additional bit in ColumnWeaving/L aims to reduce the complexity of early pruning. In Figure 4.3, we present the early pruning implementation for ColumnWeaving/L. Due to the additional bit in front of each tuple, we can directly add the pruning mask $pMask1$ saving the overflow of each tuple in the additional bit. As next part, only performing an AND operation with the result and the second pruning mask $pMask2$ results in a mask fulfilling our early pruning criteria that the bits of all tuples have to be set to 0s if at least one predicate fails.

Compared to 6 masks and 9 operations in the early pruning of ColumnWeaving/S, we need only 2 bitmasks and 2 operations to get the pruning result for ColumnWeaving/L. Furthermore, we assume that ColumnWeaving/L reaches a higher pruning probability compared to ColumnWeaving/S, because less tuples are spanned together in each processor word. The core idea of ColumnWeaving/L is the tradeoff of requiring more storage to store codes compared to reduced complexity of early pruning.

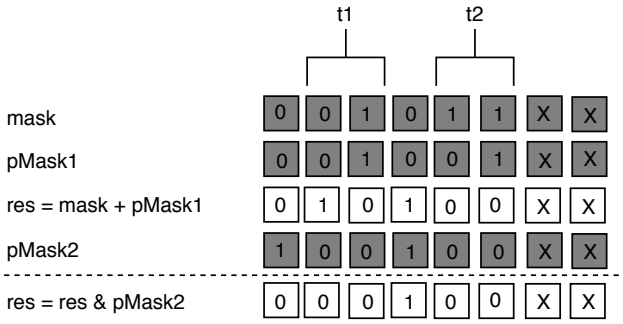


Figure 4.3: ColumnWeaving/L Early Pruning implementation

4.5 Result Shrinking

To transform the result of each scan iteration from ColumnWeaving/L into the Bitweaving bitvector representation, we use the same algorithm for result shrinking as shown for ColumnWeaving/S, whereas only the used pextmasks and shift count for each iteration changes. We calculate shiftCount using Equation 3.4 presented for ColumnWeaving/S.

In Figure 4.4, we present the result shrinking implementation of ColumnWeaving/L. indexing 2 columns using 8bit code size and 8bit processor words. For this setup, $W_{cb} = 2$, because we can store bits of 2 tuples in a processor word having 2 additional bits and 2 unused bits. Compared to ColumnWeaving/S, we have 1 additional bit per tuple, resulting in 2 codes per processor word. Consequently, we have 4 iterations where we have to shrink the scan result into one 8bit result word.

Again, $pextMask = 10101010$ for this setup. The first result shrinking iteration results in $shiftCount = 6$ to fill up the first 2 bits of the result word with the extracted result, whereas for the second iteration, $shiftCount = 4$ to fill up the next two bits of the result word, continuing with the third and fourth iteration,

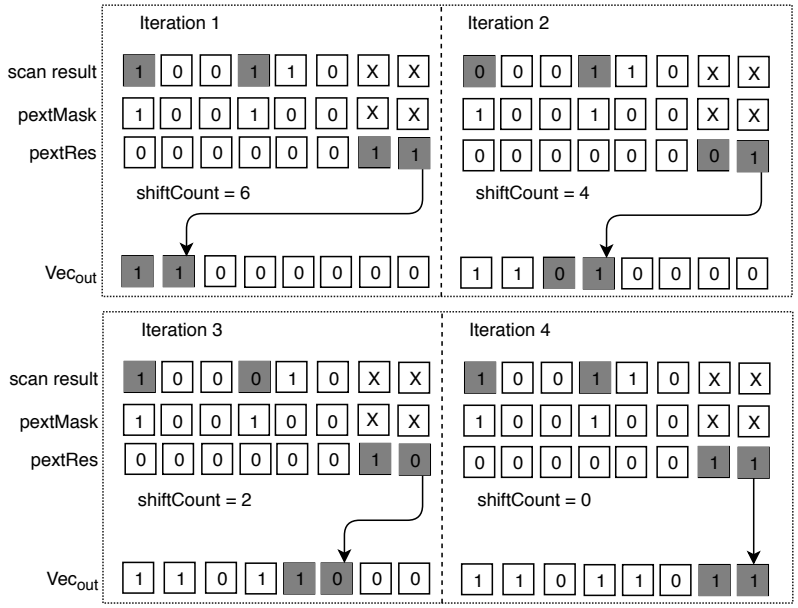


Figure 4.4: ColumnWeaving/L Result Shrink implementation

respectively. In this setup, ColumnWeaving/L requires 4 iterations to fill a single 8bit processor word with scan results, compared to ColumnWeaving/S with 2 iterations. Although the additional bits of ColumnWeaving/L increase the required iterations, we have 25% of unused bits in each processor word, which depends on the current setup.

4.6 Summary

In this section we presented ColumnWeaving/L, a second variant of ColumnWeaving. This approach uses an additional bit to store intermediate scan results for the tuples to reduce the complexity of the early pruning implementation. These additional bits affect the spanning used to create the index and lead to significantly higher memory consumption compared to ColumnWeaving/S. We accept this tradeoff, because we assume an improved pruning probability as well as reduced execution time for the early pruning algorithm. We can apply the same scan and result shrinking algorithms for ColumnWeaving/L as used for ColumnWeaving/S.

5. Evaluation

In this chapter, we evaluate the early pruning and the performance of ColumnWeaving/S and ColumnWeaving/L. At first, we describe our evaluation setup. Secondly, we run a set of synthetic benchmarks to evaluate how the early pruning behaves on different selectivities along with the performance of the scan operation. After the synthetic benchmarks, we run ColumnWeaving against Bitweaving/V using TPC-H queries over multiple columns. As last step, we summarize the results of our experiments and evaluate our research question.

5.1 Evaluation Setup

To describe our evaluation setup, we start presenting the hardware configuration used for all benchmarks including compiler options. As next step, we present the benchmarks we use to evaluate the early pruning behavior of ColumnWeaving in synthetic benchmarks. To not only rely on synthetic results, we present real world benchmarks using TPC-H queries comparing ColumnWeaving/S and ColumnWeaving/L against Bitweaving/V.

5.1.1 Hardware Configuration

We run our experiments on a machine with Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz with 128KB L1 cache, 1MB L2 cache and 6MB L3 cache. Our testing machine has 16GB of SODIMM DDR3 RAM. As operating system we use ArchLinux on stable branch with Linux Kernel 5.0.9. We use GCC in version 8.3.0 as compiler and compile all experiments with the O3 optimization flag. To get meaningful results and to eliminate deviations, we run all of our benchmarks with 20 repetitions and build average values out of the results for our evaluation.

5.1.2 Benchmarks

We divide the evaluation of ColumnWeaving/S and ColumnWeaving/V into two steps: We start with synthetic benchmarks to evaluate the pruning behavior and continue executing a benchmark on real-world data using TPC-H queries.

Synthetic Benchmarks

As first steps of our evaluation, we examine the early pruning behavior of ColumnWeaving/S and ColumnWeaving/L with a dataset containing 2 columns with 10 million 16bit codes. We execute a scan on this dataset with two predicates with varying selectivities from 0.01 to 0.5 using ColumnWeaving/S and ColumnWeaving/L. According to the fact, that considering predicates together reduces the overall selectivity, we expect low response times for both ColumnWeaving implementations for configurations, where at least one predicate has a low selectivity. Because only low response times are not a clear indicator for early pruning working as expected, we evaluate how many codes could be pruned for ColumnWeaving/S. According to the equivalent behavior of response times, we assume that ColumnWeaving/L reaches nearly identical pruning rates.

Real-world Benchmark

To see how the response time of ColumnWeaving is compared to Bitweaving/V and if the early pruning works equally for real-world queries as for synthetic ones, we run scan operations using different TPC-H queries. We perform the Q6 and Q19 queries on the lineitem table (we name them LQ6 and LQ19 for the rest of this work) and the Q17 and Q19 queries on the part table (names PQ17 and PQ19) and measure the response time needed to execute the queries.

We use LQ6, which we introduced as sample for reduced selectivity combining predicates, as first query having selectivity around 1.7%. As second query, we use LQ19, which is composed of 3 multi-column selection predicates, each having 3 predicates. Furthermore, we use PQ17 as candidate with very low selectivity ($< 1\%$) and PQ19 as second query to evaluate the response time for 3 multi-column selection predicates.

From this benchmark, we expect that ColumnWeaving/S and ColumnWeaving/L perform at least as fast as Bitweaving/V, since all used queries index multiple columns and have low selectivities and may match the requirements for our improved early pruning implementation. Furthermore, we expect ColumnWeaving/S to result in lower response times than ColumnWeaving/L. Although ColumnWeaving/L needs less effort to apply early pruning, it needs significantly more storage and consequently more iterations to execute the same query as ColumnWeaving/S.

Since the response time of ColumnWeaving/S, as well as ColumnWeaving/L, is significantly higher in all queries, we execute additional experiments to evaluate causes for the higher response time. As first cause, we assume bad pruning behavior for both index structures. Consequently, we evaluate the pruning behavior of ColumnWeaving/S and ColumnWeaving/L using the same setup and queries. As second cause, we assume that additional bits required for ColumnWeaving to index all columns lead to higher response times. Consequently, we use Bitweaving/V with the same code sizes as ColumnWeaving and evaluate the results.

5.2 Synthetic Benchmarks

Since the major benefit of ColumnWeaving compared to Bitweaving/V is the improved early pruning combining multiple predicates together to reduce overall selec-

tivity, we start our evaluation with measuring the response time of ColumnWeaving/S and ColumnWeaving/L using predicates with different selectivities. Since only response time is not a valid result to prove how good early pruning works with different selectivities, we measure the number of codes that could be pruned while executing the queries.

5.2.1 ColumnWeaving/S Response Time

In Figure 5.1, we present a scan execution using 2 predicates with selectivities reaching from 0.01 to 0.5 and measure the response time of ColumnWeaving/S using 10 million data items and 16bit code size for each column. We expect significantly reduced response time for all scans having at least one predicate with low selectivity (< 0.1) according to our hypotheses for the improved early pruning. For selectivities > 0.1, ColumnWeaving/S results in nearly identical response times, whereas for selectivities < 0.1, the response time falls along with the selectivity. Matching our hypothesis, ColumnWeaving/S needs only one predicate with low selectivity to apply improved early pruning and to reduce the response time. If we consider the point $P_1 = 0.01$ and $P_2 = 0.5$, we have one predicate with high selectivity and one with low selectivity and ColumnWeaving/S results in a comparatively less response time compared to $P_1 = 0.5$ and $P_2 = 0.5$.

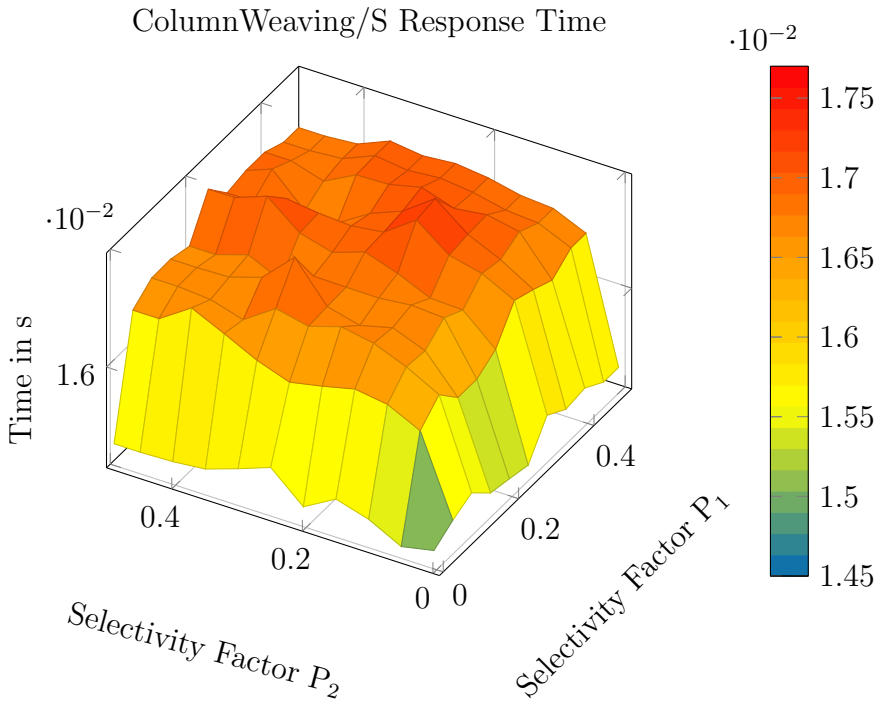


Figure 5.1: Response time of ColumnWeaving/S executing a scan using two predicates under varying selectivities with a dataset of 10 million items and 16bit code size for both columns

Although ColumnWeaving/S results in lower response time for queries with at least one predicate with low selectivity, the difference in response time is comparatively small. We measure a maximum difference in response time of around 2 milliseconds, which is compared to 16ms response time only a small performance increase.

5.2.2 ColumnWeaving/L Response Time

After getting results indicating to fulfill our hypothesis for ColumnWeaving/S, we perform the same experiment on ColumnWeaving/L to evaluate if this approach also fulfills our hypothesis. In Figure 5.2, we present the response time of ColumnWeaving/L using the same setup with using two predicates with selectivities from 0.01 to 0.5.

Along with ColumnWeaving/S, also ColumnWeaving/L results in reduced response time if at least one predicate has low selectivity, consequently we assume that both implementations match our hypothesis for improved early pruning. Compared to ColumnWeaving/S, ColumnWeaving/L needs in average around 2ms more response time to execute the scan operation. Since this synthetic benchmark focusses on applying early pruning, we inspect a potential performance difference of both implementations in our TPC-H benchmarks.

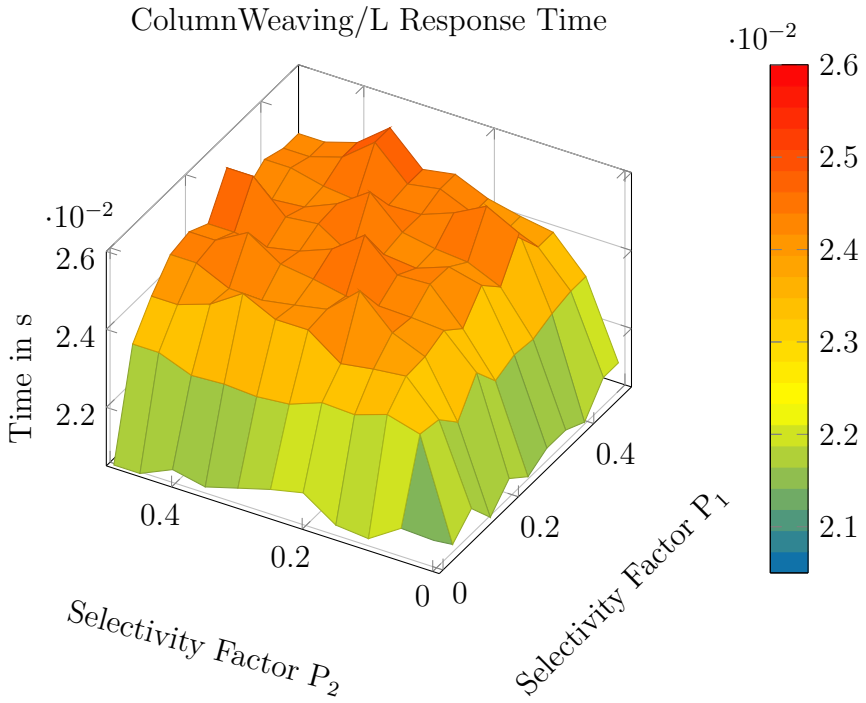


Figure 5.2: Response time of ColumnWeaving/L executing a scan using two predicates under varying selectivities with a dataset of 10 million items and 16bit code size for both columns

5.2.3 ColumnWeaving/S Pruning Rate

Summarizing the measurements of ColumnWeaving/S and ColumnWeaving/L for early pruning, for both implementations the response time indicates that early pruning works well for at lest one predicate with low selectivity. Since we measure the overall response time, this is not a valid indicator if our early pruning implementation really fulfills our hypothesis. Consequently, we introduce the *Pruning Rate* $P_r = \frac{PRUNED\_CODES}{NUM\_CODES}$ as the number of codes on which we can apply early pruning compared to the number of codes indexed by ColumnWeaving.

In Figure 5.3, we present the pruning rates measured using 10 million data items with 16bit code size and two predicates with selectivities from 0.01 to 05. According to our measurements for the response time of ColumnWeaving/S, the percentage of pruned iterations behaves inversely proportional to the response time of the query execution. For at least one predicate with low selectivity percent of pruned iterations, the percentage of pruned iterations is comparatively high, whereas if both predicates have more than 0.1 selectivity, the percentage of pruned iterations becomes nearly 0, which means no early pruning can be applied. Consequently we can acknowledge the hypothesis that considering multiple predicates of a query together reduces the number of words, which we use as basis of our ColumnWeaving implementation.

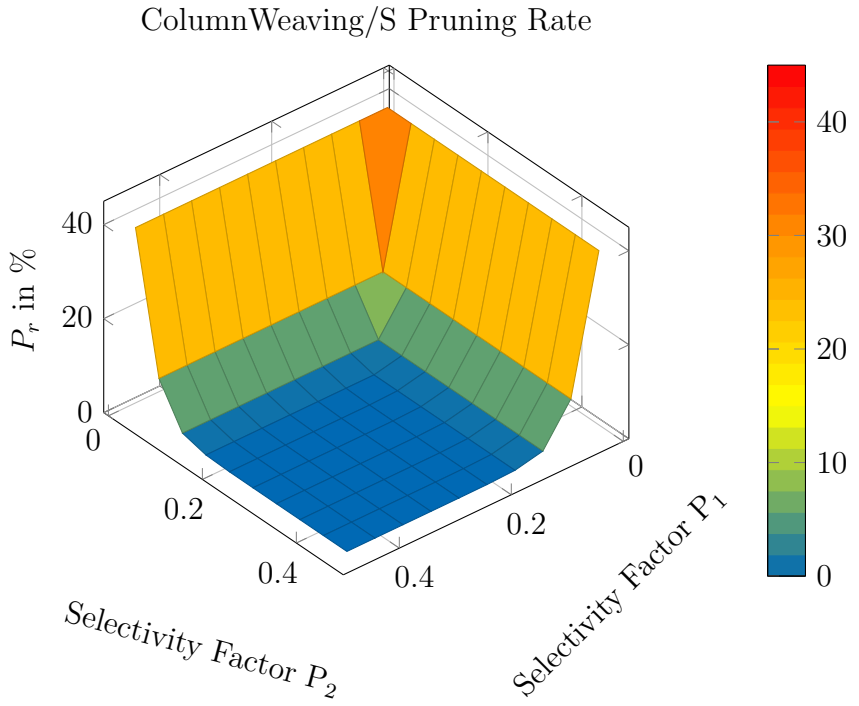


Figure 5.3: Pruning Rate P_r of executing a scan for two predicates under varying selectivities with a dataset of 10 million items and 16bit code size for both columns

Although we measured results indicating that considering multiple predicates together leads to improved early pruning, we cannot conclude information about performance compared to Bitweaving/V, since real world data often does not follow strict limitations of selectivity. Consequently we test our implementation with TPC-H benchmarks against Bitweaving/V.

5.3 TPC-H Benchmark

To get comparable results for query execution time, we benchmark our ColumnWeaving implemenation using TPC-H queries against Bitweaving/V. In Figure 5.4, we present the results comparing ColumnWeaving/S and ColumnWeaving/L against Bitweaving/V. Using the lineitem table, we evaluate the response time of the index structures for the queries LQ6 and LQ19 and for the part table we evaluate the response time for the queries PQ17 and PQ19.

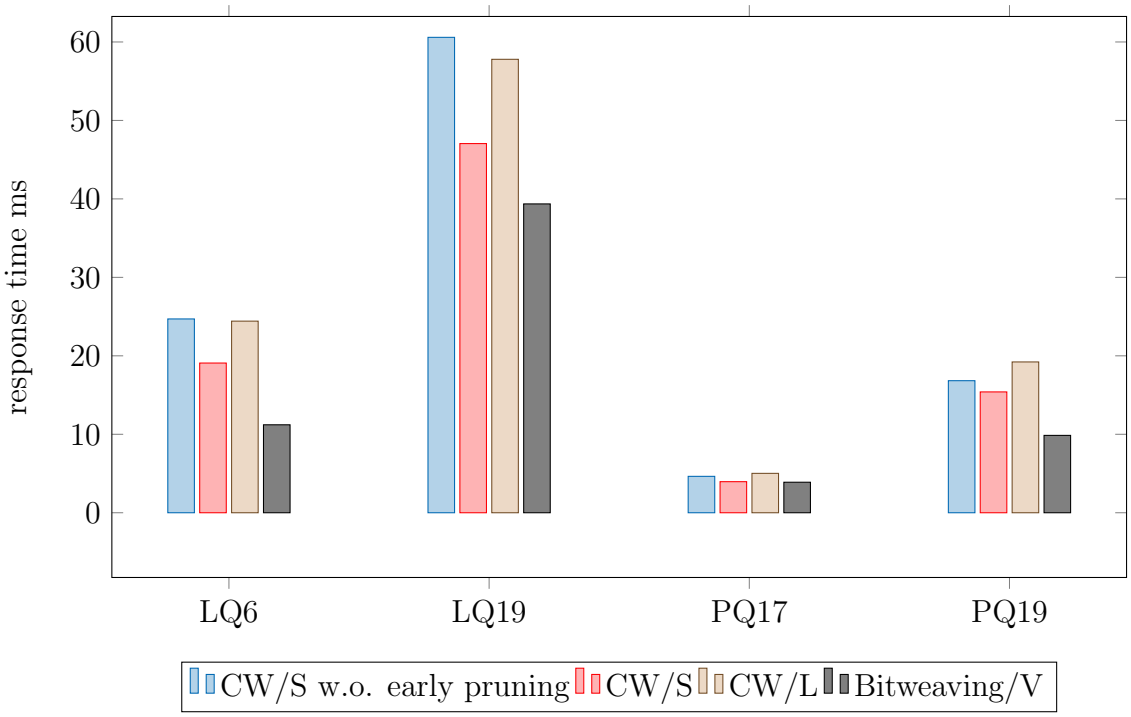


Figure 5.4: Performance on TPC-H queries compared to Bitweaving/V

In contrast to our expectation, ColumnWeaving/S as well as ColumnWeaving/L result in significantly higher response time than Bitweaving/V for all queries except for PQ17. For LQ6 and PQ19, ColumnWeaving/S results in around 1.7 times higher response time and ColumnWeaving/L results in around 2 times higher response time compared to Bitweaving/V. For LQ19, ColumnWeaving/S is around 1.2 times slower than Bitweaving/V. For PQ17, ColumnWeaving/S results in nearly the same response time as Bitweaving/V, whereas ColumnWeaving/L is slightly slower that Bitweaving/V. Furthermore, ColumnWeaving/S without early pruning performs slightly better for PQ17 and PQ19 as ColumnWeaving/L. As possible causes for the higher response times of ColumnWeaving, we consider multiple limitations of ColumnWeaving against Bitweaving/V.

Limited Early Pruning

At first, our adapted early pruning implementation has a limitation regarding the selectivity, which reduces the number of pruned iterations, which we call *Limited Early Pruning* for the rest of this work. To apply early pruning in an iteration, the ith bits of all tuples contained in the processor words have to differ from the query predicate, otherwise early pruning is not possible. For example, indexing 3 columns with ColumnWeaving/S having 64bit processor words, we have the ith bits of 21 tuples in each processor word. In worst case, in 20 of 21 tuples at least one predicate fails and one tuple matches all predicates. Consequently, we have to iterate over all bits of the 20 tuples although a match of these tuples is impossible.

Unused Bits

Secondly, in our current implementation, ColumnWeaving needs to index all columns with the same code size to have a consistent memory layout. Consequently, we have

to take the minimal maximum of code sizes of all columns and fill up columns with less code size with 0s, which may result in more words to process compared to Bitweaving/V. We call that limitation *Unused Bits* for the rest of this work. For example in Q6, the first column has a code size of 12, whereas the second and third column have code sizes of 4 and 6 bits.

Complexity of Early Pruning and Result Shrinking

Furthermore, we assume the overhead of our early pruning algorithm is too complex in ColumnWeaving/S or we have too much storage consumption ColumnWeaving/L to beat Bitweaving/V. As last cause, we assume that the result shrinking approach costs too much time. In the following, we will evaluate the first two mentioned aspects that we assume as possible causes for the high response time of ColumnWeaving compared to Bitweaving/V, to find reference points for improving our index structure. We will propose possible improvements for the early pruning and result shrinking complexity in Chapter 7.

5.3.1 Limited Early Pruning

As first step to evaluate why the response time of ColumnWeaving is significantly higher than Bitweaving/V, we measure the pruning rate of the queries used in the previous benchmark. In Figure 5.5, we present the pruning rates s of ColumnWeaving/S and ColumnWeaving/L for the lineitem queries LQ6 and LQ19 and the part queries PQ17 and PQ19.

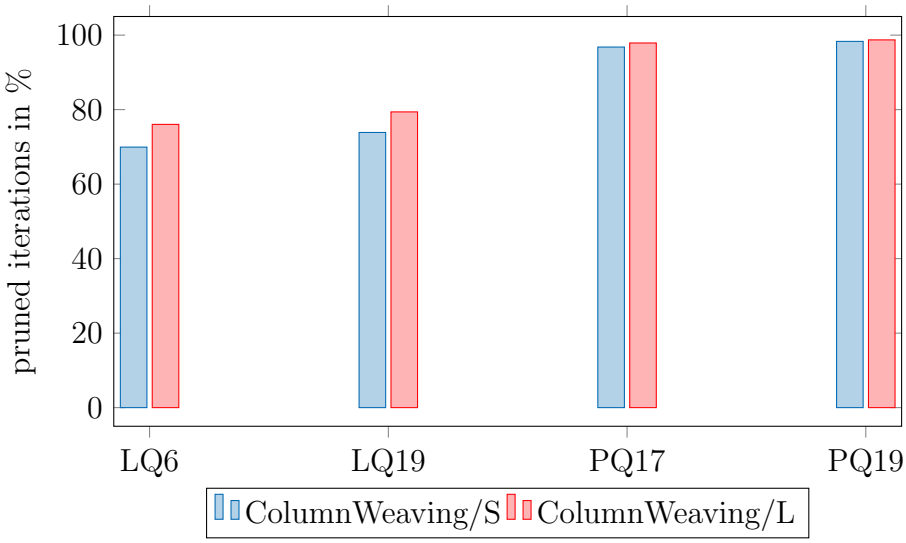


Figure 5.5: Percentage of early pruned iterations in TPC-H queries

For LQ6 and LQ19 both ColumnWeaving implementations reach high pruning rate around 70%. In contrast to 70% pruning rate, both implementation result in significantly higher response time compared to Bitweaving/V. For PQ19 and PQ17 ColumnWeaving reaches a very high pruning rate around 96%. For PQ17, ColumnWeaving/S reaches a lower response time than Bitweaving/V, whereas for PQ19 we measure still a higher response time compared to Bitweaving/V.

In summary, our first assumption for higher response times of ColumnWeaving does not apply, since ColumnWeaving/S and ColumnWeaving/L reach pruning rates from 70% to 96%. This benchmark shows, that ColumnWeaving reaches good pruning rate for real-world data. Because PQ17 is the only query that lead to the assumption, that the pruning rate directly reflects in the the response time, we continue evaluating possible causes for the higher response time of ColumnWeaving compared to Bitweaving/V.

5.3.2 Unused Bits

As second step to evaluate why the response time of ColumnWeaving is significantly higher than Bitweaving/V, we examine if the unused bits that result from using the minimal maximum code size of all columns lead to higher response time. We adapt the code sizes used by Bitweaving/V to the same number used by ColumnWeaving to see if the unused bits in ColumnWeaving lead to higher response time.

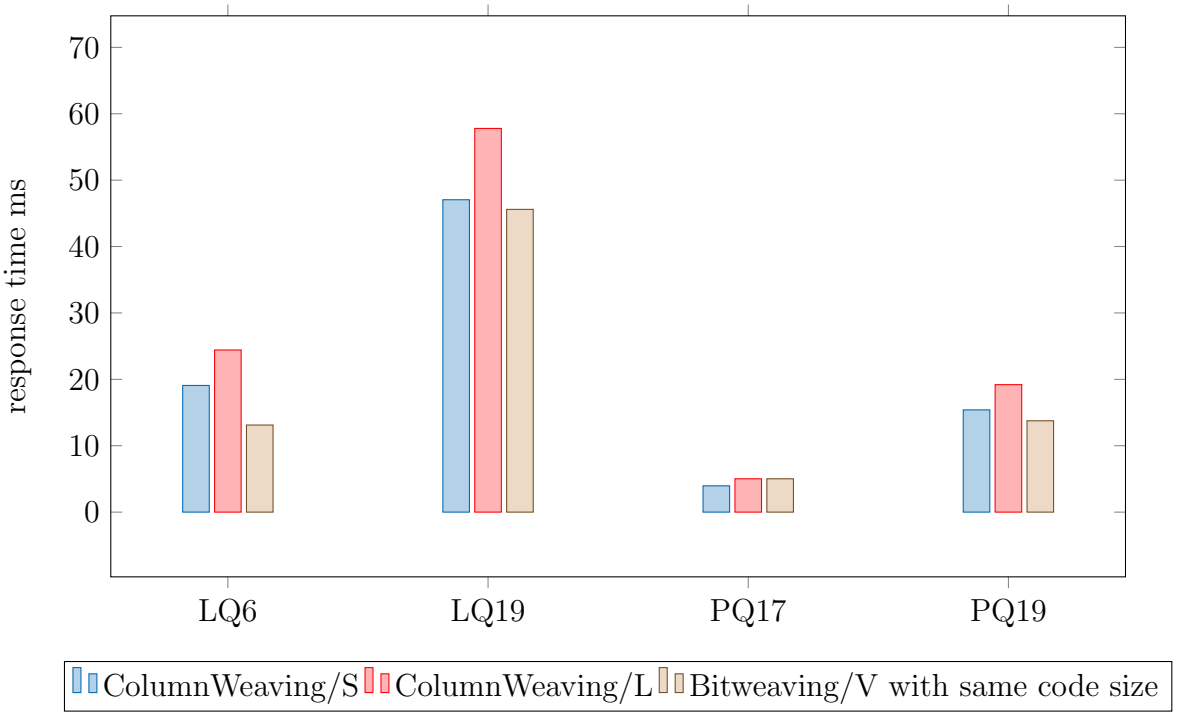


Figure 5.6: Performance of Bitweaving/V using the same code size for all columns as ColumnWeaving

In Figure 5.6, we present the results evaluating the previous queries using the same code sizes for Bitweaving/V that are used by ColumnWeaving to index the required columns for the queries. For the query Q19, ColumnWeaving/S reaches nearly the same response time than Bitweaving/V, whereas for PQ17, ColumnWeaving/S is faster than Bitweaving/V. In summary, the response time of Bitweaving/V is slightly higher for all queries than using the default code sizes giving ColumnWeaving the potential to beat it in one query, but we do not see the number of unused bits as main cause for higher response times of ColumnWeaving. Consequently, we assume the complexity of early pruning and result shrinking as main cause for higher response times and give proposals to reduce complexity in Chapter 7.

5.4 Summary

After evaluating benchmarks on synthetic and real-world data, we summarize the results of our ColumnWeaving implementation. Evaluating the response time and the pruning rate with synthetic benchmarks lead to results that match our initial hypothesis: ColumnWeavings adapted early pruning leads to lower response time for multiple predicates if at least one predicate has low selectivity. As important result, we determine that ColumnWeaving/S performs better than ColumnWeaving/L for all selectivities used in the synthetic benchmark.

In the real-world benchmarks using TPC-H queries, both ColumnWeaving implementations result in significantly higher response time than Bitweaving/V for all queries except for one, whereas we expected at least an equal response time. To find possible causes for the higher response time, we evaluate the pruning behavior of ColumnWeaving on the used TPC-H queries. Both ColumnWeaving implementations reach very high pruning rates from 70% to 96%. Consequently, a low pruning rate for the TPC-H queries is not the main cause for the higher response time.

Furthermore we examine if the number of unused bits arising from indexing all columns with the same code size lead to higher response time. We test Bitweaving/V with the same code size used by ColumnWeaving to see if the number of unused bits lead to the higher response time. As our results show, using the same code sizes as for ColumnWeaving, Bitweaving/V only has slightly higher response times. Consequently, the number of unused bits is also not the main cause for the higher response time.

In addition to the possible causes that we evaluated and which are not the main cause for the higher response time of ColumnWeaving compared to Bitweaving/V, we assume the complexity of early pruning and result shrinking as possible causes for the higher response time, for which we propose solutions that we will prove in the future in Chapter 7.

6. Related Work

To fulfill the requirements of modern systems, the focus of index structures is clearly set to main-memory index structures. Zeuch et al. propose an index structure for applying SIMD powered k-ary search on a B^+-tree [ZHF14]. Kim et al. present a hardware sensitive index structure called FAST, that focusses on main memory but also supports disk-based actions [KCS+10]. Abadi et al. propose *Sorted Projection* [ABH+13], which sorts a set of frequently used columns and add an extra column to speed up the search performance. This approach is also used in [LFV+12]. Furthermore, Boncz et al. present MonetDB [BKM08], another main-memory index structure.

Considering Bitweaving [LP13] as basis for our implementations, there are some more contributions on achieving bit parallelism. O'Neil et al. at first present algorithms performing operations on multiple bits in parallel [OQ97] and serves as basis for Bitweaving. Johnson et al. perform similar operations in IBM's Blink System [JRSS08]. Rinfret et al. also evaluate bit parallel methods [ROO01].

Indexing multiple-columns together, Broneske et al. present the *ELF* index structure [BKSS17], that uses redundancy elimination to create distinct dimensionlists out of multiple columns. They present, that evaluating multi-column selection predicates together may outperform multiple isolated scans. Bohm et al. propose a set of multi-dimensional index structures in relational database [BBKM00]. Bayer et al. adapt the B-tree to support indexing multiple columns [Bay97].

Along with the search speed of modern index structures, the creation speed becomes more important, since the amount of data to store in a single index grows fast. Van et al. propose a generic approach to bulk load multidimensional index structures [VdBSW97]. Berchtold et al. present bulk load operations to improve high-dimensional index structures [BBK98].

7. Conclusion and Future Work

In this work, we adapt Bitweaving/V for multi-column indexing and evaluate the performance compared to the original implementation. We start introducing Bitweaving with focus on Bitweaving/V and present our adaptions ColumnWeaving/S and ColumnWeaving/L. We show how Bitweaving/V implements early pruning and present our advanced early pruning approach for ColumnWeaving.

After presenting our implementations, we evaluate the response time and the early pruning behavior of ColumnWeaving/S and ColumnWeaving/L in a synthetic benchmark and in the TPC-H benchmark comparing against Bitweaving/V. In contrast to our expectations, the current implementation of ColumnWeaving does not reach better performance than Bitweaving/V on queries containing multiple predicates with low selectivity. Since we could exclude limited early pruning and unused bits as main cause for the higher response time of ColumnWeaving compared to Bitweaving/V, we plan to evaluate the complexity of early pruning and result shrinking as main cause for the response time.

Furthermore, we plan to implement a better handling of different code sizes in combination with better storage layout and memory usage. We plan to index the columns with the minimal code size of all columns, cut off the remaining of bits and store them at the end of the used memory. We assume that having a high probability that the most significant bits stored together are enough to apply early pruning. Furthermore, ColumnWeaving can store more tuples in one processor word and in best case, the remaining bits stored at the end of the memory layout can be skipped completely.

As second proposal to improve ColumnWeaving in the future, we plan to replace the static spanning approach of storing bits of all columns in each processor word with a dynamic approach. The idea of dynamic spanning is to span columns together and split the ith bit of columns into multiple processor words instead of one. This provides more flexibility, because not all columns have to be indexed together, but e.g. columns having small selectivity or the same code size can be indexed together. We plan to group the processor words belonging to the same dynamic spanning in the memory layout to achieve better memory usage.

After implementing additional performance improvements, we plan to compare ColumnWeaving against other state-of-the-art index structures, like Column Imprints [SK13], Sorted Projections [LFV+12] and the Elf index structure [BKSS17].

Bibliography

[ABH+13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013. (cited on Page 43)

[Bay97] Rudolf Bayer. The universal B-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997. (cited on Page 43)

[BBC+12] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, et al. Business analytics in (a) blink. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):9–14, 2012. (cited on Page 9)

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. In *International Conference on Extending Database Technology (EDBT)*, pages 229–234, 2014. (cited on Page 6)

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *International Conference on Extending Database Technology (EDBT)*, pages 216–230. Springer, 1998. (cited on Page 43)

[BBKM00] Christian Böhm, Stefan Berchtold, Hans-Peter Kriegel, and Urs Michel. Multidimensional index structures in relational databases. *Journal of Intelligent Information Systems*, 15(1):51–70, 2000. (cited on Page 43)

[BKM08] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008. (cited on Page 1 and 43)

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating multi-column selection predicates in main-memory-the elf approach. In *International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017. (cited on Page 1, 9, 43, and 46)

[BS17a] David Broneske and Gunter Saake. Exploiting capabilities of modern processors in data intensive applications. *it-Information Technology*, 59(3):133–140, 2017. (cited on Page 6)

[BS17b] David Broneske and Martin Schäler. Single instruction multiple data–not everything is a nail for this hammer. *Failed Aspirations in Database Systems*, 2017. (cited on Page 7)

[CHKK01] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *International Conference on Very Large Data Bases (VLDB)*, volume 1, pages 181–190, 2001. (cited on Page 4)

[CR94] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, volume 23. ACM, 1994. (cited on Page 8)

[DYZ+15] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. Query optimization in Oracle 12c database in-memory. *Proceedings of the VLDB Endowment*, 8(12):1770–1781, 2015. (cited on Page 1)

[ESE06] Stijn Eyerman, James E Smith, and Lieven Eeckhout. Characterizing the branch misprediction penalty. In *International Symposium on Performance Analysis of Systems and Software*, pages 48–58. IEEE, 2006. (cited on Page 6)

[FHL10] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010. (cited on Page 9)

[FKLU05] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005. (cited on Page 6)

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972. (cited on Page 5)

[FML+12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database - an architecture overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):28–33, 2012. (cited on Page 9)

[GMM97] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324. Citeseer, 1997. (cited on Page 5)

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *International Conference on Management of Data (SIGMOD)*, volume 30, pages 461–472. ACM, 2001. (cited on Page 8)

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. (cited on Page 3)

[Int] Intel. Intel intrinsics guide - pextu64. (cited on Page 23)

[JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *Proceedings of the VLDB Endowment*, 1(1):622–634, 2008. (cited on Page 43)

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 339–350. ACM, 2010. (cited on Page 1, 4, and 43)

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008. (cited on Page 1)

[KL79] HT Kung and Charles E Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics, 1979. (cited on Page 6)

[KN11] Alfons Kemper and Thomas Neumann. HyPer: a hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *International Conference on Data Engineering (ICDE)*, pages 195–206. IEEE, 2011. (cited on Page 1)

[LFV+12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012. (cited on Page 43 and 46)

[LP13] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *International Conference on Management of Data (SIGMOD)*, pages 289–300. ACM, 2013. (cited on Page 1, 2, 9, 13, 15, and 43)

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *International Conference on Management of Data (SIGMOD)*, 26(2):38–49, 1997. (cited on Page 10 and 43)

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–2. ACM, 2009. (cited on Page 1)

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1493–1508. ACM, 2015. (cited on Page 7)

[ROO01] Denis Rinfret, Patrick O'Neil, and Elizabeth O'Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001. (cited on Page 43)

[RR00] Jun Rao and Kenneth A Ross. Making B+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000. (cited on Page 4)

[SAMC99] Kevin Skadron, Pritpal S Ahuja, Margaret Martonosi, and Douglas W Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, 1999. (cited on Page 5)

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary search on modern processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 52–60. ACM, 2009. (cited on Page 1)

[SK13] Lefteris Sidirourgos and Martin Kersten. Column imprints: a secondary index structure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 893–904. ACM, 2013. (cited on Page 1 and 46)

[VdBSW97] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *International Conference on Very Large Data Bases (VLDB)*, volume 97, pages 406–415, 1997. (cited on Page 43)

[VTG+99] Alexander V Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *International Conference on Supercomputing*, pages 145–154, 1999. (cited on Page 5)

[ZHF14] Steffen Zeuch, Frank Huber, and Johann-christoph Freytag. Adapting tree structures for processing with simd instructions. 2014. (cited on Page 1 and 43)

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A Boncz. Super-scalar ram-cpu cache compression. In *International Conference on Data Engineering (ICDE)*, volume 6, page 59, 2006. (cited on Page 9)

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the international conference on Management of Data (SIGMOD)*, pages 145–156. ACM, 2002. (cited on Page 7)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 28.5.2019