Lecture #03: Data Formats & Encoding II

15-721 Advanced Database Systems (Spring 2024)
`https://15721.courses.cs.cmu.edu/spring2024/`
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

Continuing with our study of analytics workloads in the cloud and data moving into data lakes that reside on object storage services like S3, we look at more storage formats based on the Partition Attributes Across (PAX) hybrid storage model. In the previous lecture, we also covered the N-ary and decomposition storage models. Rather than storing all the attributes for a single tuple contiguously in a single page (in the case of N-ary), or storing a single attribute for all tuples in a block of data (with decomposition), PAX vertically partitions attributes within a database page.

We covered open storage formats like Apache Parquet based on PAX that support this new cloud data warehousing paradigm. However, many existing formats are not optimized for remotely-accessed datalakes and today's high-throughput networks. In this lecture, we explore several modern storage formats that make use of recent advances in networked storage and data-parallel CPU architectures.

In particular, we will look at the following storage formats:

1. BtrBlocks [4] (from the Technical University of Munich)
2. FastLanes [1] (from the Centrum Wiskunde & Informatica in The Netherlands)
3. BitWeaving [5] (from the University of Wisconsin)

2 BtrBlocks

The BtrBlocks paper argues that inefficient decompression with formats like Apache Parquet makes scans CPU-bound and increases query time and cost. To address this, BtrBlocks uses additional encoding schemes for different data types. However, the effectiveness of these encodings differs depending on the data distribution. Given a set of encodings, we also need an algorithm for deciding which encoding is most effective for a specific data block.

2.1 Overview

BtrBlocks is an open columnar storage format designed for data lakes. It is designed to minimize the overall workload cost through low storage costs and fast decompression. It uses more aggressive nested encoding schemes than Parquet and ORC. It uses seven existing and one new encoding scheme for floating-point values.

BtrBlocks uses a greedy algorithm to select the best encoding for a column chunk, based on sampling, and then recursively tries to encode outputs of that encoding. The benchmarks in the paper show scans on real-world data are 2.2× faster and 1.8× cheaper when using BtrBlocks over Parquet. BtrBlocks stores a file's meta-data separately from the data, arguing for the ability to prune data using statistics and indices before accessing a file through a high-latency network.

2.2 Sampling technique

BtrBlocks collects a sample from the data and then tries out all viable encoding schemes for three rounds. Samples are collected by selecting multiple small runs from non-overlapping random positions. The BtrBlocks paper finds that sampling multiple small chunks across the entire block improves accuracy, although there is little difference between strategies that choose chunks of ≥ 16 tuples. The intuition is that the sample needs to capture both **data locality** and **data distribution** across the entire block.

2.3 Encoding schemes

BtrBlocks selected the following encoding schemes by analyzing a collection of real-world datasets:

- **Run-length encoding (RLE) & One Value**: involves storing the run $(42, 3)$ instead of $\{42, 42, 42\}$. One value is an extreme case for columns with only one unique value per block.
- **Frequency encoding [8]**: addresses skewed distributions, using several code lengths based on data frequency. For example, the two most frequent values can be represented by a one-bit code, and the next eight most frequent values using a three-bit code. BtrBlocks adapts this by only storing the top value, a bitmap encoding which values are the top value, and the exception values which are not the top value.
- **Frame of Reference (FOR) + Bit-packing**: FOR encodes each integer as a delta to a chosen base value, for example storing the base 100 and $\{5, 1, 13\}$ instead of $\{105, 101, 113\}$. Useful in combination with bit-packing, which truncates unnecessary leading bits. For example, $\{5, 1, 13\}$ can be bit-packed using 4 bits for each value instead of 8.
- **Dictionary encoding**: covered in the previous lecture
- **Fast static symbol table (FSST)**: see section 2.3.1
- **Roaring bitmaps for NULLs + exceptions**: see section 2.3.2
- **Pseudodecimals**: see section 2.3.3

Note that these encoding schemes can be used in a cascade. For example, you could encode with RLE and then with bit-packing.

2.3.1 Fast static symbol table (FSST)

In many real-world databases strings form a large fraction of data. Strings are often compressed using dictionaries. However, since dictionary compression uniquely maps strings to fixed-size integers, compression can be sub-optimal on data that has many unique values that might be similar but not equal. Further, DBMSs benefit from random access to individual string attributes, which is not possible when using blockwise general string compression like LZ4. FSST, introduced by Boncz et al. [2], is a string encoding scheme that supports random access without decompressing previous entries. It replaces frequently occurring substring (up to 8 bytes) with 1-byte codes. It uses a “perfect” hash table for fast look-up of symbols without conditionals and loops.

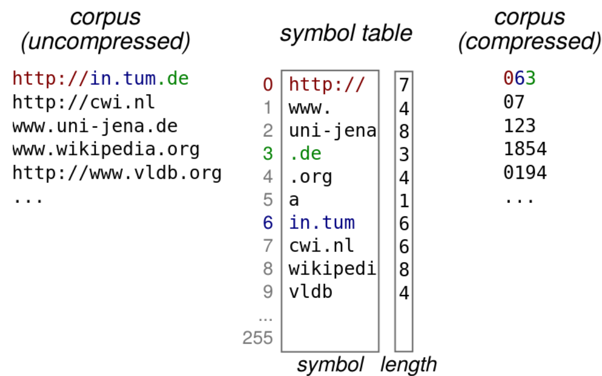


Figure 1: Symbols of length 1-8 are replaced with 1-byte codes. Performance is dependent on finding a good symbol table.

FSST constructs a symbol table using an evolutionary algorithm. Consider an example for encoding the string "tumcwitumvldb".

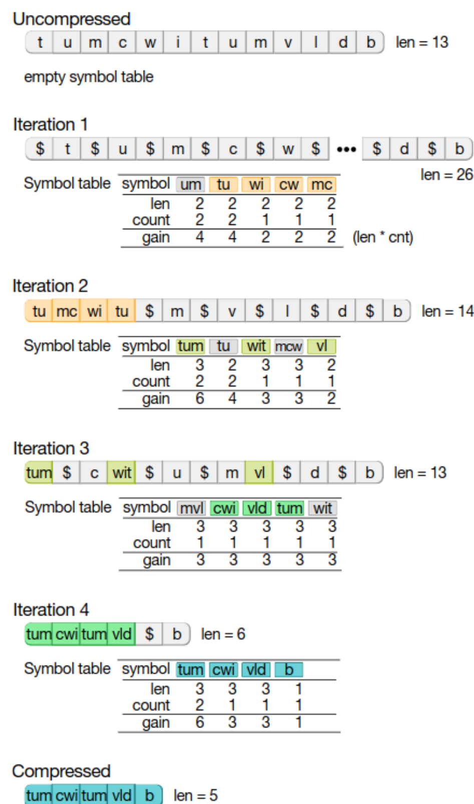


Figure 2: FSST symbol table construction algorithm

Assume we only look at symbols that are at most 3 characters long, and that the symbol table can have at most 5 entries. The algorithm proceeds in multiple iterations, starting with an empty symbol table as shown in figure 2. In each iteration the compressed string is shown at the top, compressed using the previous iteration's symbol table, with '$' used to depict the escape byte needed to escape each symbol. In the

beginning, with an empty symbol table, the length of the compressed string doubles because we must escape each symbol. Each new iteration does two things. First, it computes the compression factor using the current symbol table. This step also counts how often each symbol occurs in the compressed string and the symbols that appear in succession. Second, a new symbol table is constructed by selecting the symbols with the highest apparent gains (number of occurrences * symbol length), including new symbols generated by concatenating pairs of successive symbols found in the compressed string. The idea is that frequently occurring longer, higher-gain symbols found in this manner will replace less worthwhile symbols from the previous iteration.

For example, in iteration 2, we generate the symbol `tum`, from successively occurring symbols `tu` and `mc`, which has higher gains than any other symbols from the symbol table from iteration 1. `tum` ends up replacing the symbol `um`, with ties broken arbitrarily.

2.3.2 Roaring bitmaps for NULLs + exceptions

Roaring bitmaps, introduced by Chambi et al. [3], are bitmap indices that switch which data structure to use for a range of values depending on the local density of bits. Dense chunks are stored using uncompressed bitmaps. Sparse chunks are stored using bit-packed arrays of 16-bit integers.

Roaring partitions the space $[0, n)$ into chunks of 2^{16} integers ($[0, 2^{16}), [2^{16}, 2 \times 2^{16}), \ldots$). Dense chunks, containing more than 4096 integers, are stored using conventional bitmap containers (made of 2^{16} bits or 8kB) whereas sparse chunks use smaller containers made of packed sorted arrays of 16-bit integers.

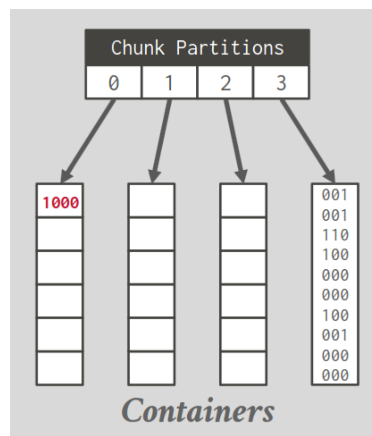


Figure 3: Roaring bitmap example

In figure 3, to insert the value 1000, we first find the chunk associated with it. Since $\lfloor 1000/2^{16} \rfloor = 0$, it will be inserted in the first container. Since there are no values in the first chunk yet we will store this value in an array container. If we wanted to insert the value 199658, we would follow the same process, computing $\lfloor 199658/2^{16} \rfloor = 3$. This time, let's say there are at least 4096 values in the third container so we are using a bitmap. Then we will simply go ahead and just set the $199658 \bmod 2^{16} = 3050$th bit.

BtrBlocks uses roaring bitmaps for storing `NULL` values for each column as well as for exceptions in frequency encoding.

2.3.3 Pseudodecimals

There has not been a lot of interest in floating-point compression until now because relational systems typically represent real numbers as `Decimal` or `Numeric`, which can be physically stored as integers. However, this is changing with the move to data lakes and the subsequent integration with non-relational systems. For example, Tableau's internal analytical DBMS encodes all real numbers as floating-point numbers and machine-learning systems rely on floating-point numbers.

Pseudodecimal encoding uses a decimal representation for encoding doubles, using two integers: *significant digit with sign* and *exponent*. For example, 3.25 becomes $(+325, 2)$. And for 0.99 the encoding stores $(+99, 2)$. The encoding algorithm determines the compact decimal representation by testing all powers of 10 and checking whether any of them correctly multiply the double to an integer value. The algorithm creates exceptions for values like negative zero, $\pm\infty$, and $\pm NaN$, as well as extremely small values like 5.5×10^{-42} that it cannot successfully encode. Refer to the BtrBlocks paper for details about the encoding and decoding process.

Pseudodecimal encoding is not well-suited for columns with many exception values, or columns with few unique values. Columns with few unique values compress almost as well by using dictionary encoding.

3 FastLanes

BtrBlocks, Parquet, and ORC generate variable-length runs of values. This wastes cycles during decoding for both scalar and vectorized operations. Parquet and ORC, in particular, use Delta encoding where each tuple's value depends on the preceding tuple's value. This is impractical to process with SIMD because you cannot pass data between lanes in the same register.

FastLanes is a suite of encoding schemes that achieve better data parallelism through clever reordering of tuples to maximize useful work in SIMD operations. These algorithms can be emulated on AVX512 or scalar instructions. The FastLanes authors define a "virtual" ISA with 1024-bit SIMD registers, designed to operate on any current and future register widths. FastLanes' novel "Unified Transposed Layout" exploits the property of relational algebra that it is based on unordered sets.

3.1 Background: Sequential Data Dependencies

Delta encoding, particularly when combined with run-length encoding, can yield immense compression ratios for some data types. However, the canonical Delta model is largely sequential because it relies on computing the accumulated differences between neighboring values in the data layout.

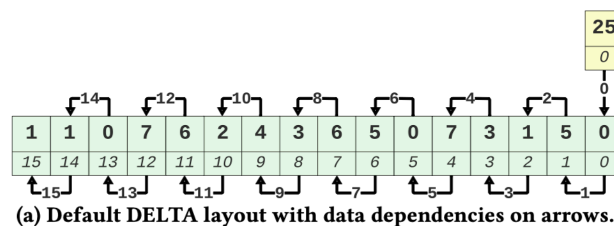


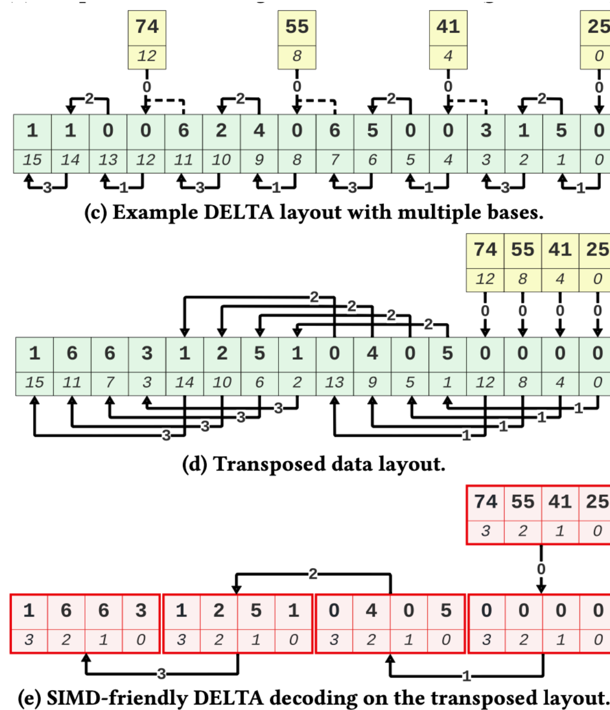
Figure 4: Sequential data dependencies in the default Delta layout.

Figure 4 illustrates the data dependencies that are created across multiple vector lanes. Even with a large register capable of holding many values, the CPU cannot make maximal use of SIMD parallelism because it still must iteratively calculate the deltas across its lanes.

3.2 The Unified Transport Layout

Upon decoding, FastLanes makes a critical adjustment to the data layout. First, it splits the data into many sequential runs, which in practice consists of 16 sets of 8 sequential values. Each run begins with its own *base* value (shown in yellow), which differs from the default format, which only has one base value.

It then reorders the values, moving each base value to the front, and then shifting over chunks of sequential values starting from each base value. [1]



(c) Example DELTA layout with multiple bases.

(d) Transposed data layout.

(e) SIMD-friendly DELTA decoding on the transposed layout.

Figure 5: Improved Delta format with UTL.

Figure 5 shows an example of the reordering. Values 0, 4, 8, and 12 are designated as the base values and are reordered to the front of the layout in step (d). Each of their dependencies, values 1, 5, 9, and 13, are moved to the next chunk over, and so on. With this transposing of data, resolving the Delta values only takes three SIMD additions on a vector that holds four values, rather than 15 additions.

3.3 A Portable SIMD Interface

There are two main challenges with designing SIMD-targeted software:

- **Varying SIMD Widths:** Modern ISAs have different register widths, ranging from 64 bits to 512 bits and possibly more. Furthermore, the FastLanes authors note that SIMD ISAs have widened dramatically in the past 25 years (by a factor of 8), and this trend is likely to continue in the future.
- **Heterogeneous ISAs:** SIMD instructions are specific to each architecture. Although high-level code can often emit SIMD instructions via compiler intrinsics, this approach is not portable across architectures. Additionally, the set of logical SIMD instructions may not be supported in some architectures; therefore, some approaches that rely on specific bitwise operators may not work on unsupported ISAs.

FastLanes resolves these challenges by introducing a "virtual instruction set" called `FLMM1024` which simulates operations on a 1024-bit register set. This virtual ISA is comprised of a minimal set of 1024-bit operations (`LOAD, STORE, AND, XOR, ADD`, etc.) that can fully execute data-parallel FastLanes decoding.

```
struct { uint64 val[16]; } FLMM1024; // 16*uint64 = FLMM1024
FLMM1024 AND<8>(FLMM1024 A, FLMM1024 B) {
  FLMM1024 R;
  for(int i=0; i<16; i++) R.val[i] = A.val[i] & B.val[i];
  return R;
}
```

Figure 6: An example of the FLLM1024 AND instruction.

FLMM instructions consist of a simple loop that executes basic C bitwise operators. This approach relies on the fact that modern optimizing compilers can auto-vectorize suitable code, ensuring that any architecture will be able to run FastLanes on the widest registers available.

4 BitWeaving

The encoding schemes we have discussed so far scan data by examining the entire value of each attribute and cannot short-circuit comparison with integer types, known as *early pruning*, because CPU instructions operate on entire words. Further, modern CPUs have a word width of 64 bits but values in a database table column are often represented with fewer bits. This leads to underutilization of the width of a word. In the case of SIMD, compressed column values are often packed into four 32-bit slots in a 128-bit SIMD word [9]. But for compressed values encoded with fewer bits, this still requires padding the values to 32-bit boundaries. This means that even with state-of-the-art techniques it still takes many cycles per input tuple to apply simple predicates on a single column of a table.

4.1 Background: Bit-sliced encoding

Noting the read-mostly environment of data warehousing, O'Neil et al. [6] introduced a bit-sliced indexing method taking an orthogonal bit-by-bit view of the same data. Consider the following data:

id	zipcode
1	21042
2	15217
3	02903
4	90220
5	14623
6	53703

And the following query:

```
SELECT * FROM customer_dim
WHERE zipcode < 15217
```

Instead of having to query all the bits of each zip code with 15217, we could represent the zip codes as *bit slices*:

Each bit slice can be represented as a bitmap, and the bit-sliced index for the `zipcode` column is just the set of these bitmaps. To execute the range query with this bit-sliced representation, we can walk each slice and construct a result bitmap that skips entries that have a 1-bit set in the first three slices (16, 15, and 14) since 15217 only has 14 bits in its binary representation.

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1
0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1
0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	1	1
0	0	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1

These bit slices can also be used for efficient aggregate computations, for example, to calculate the sum of a column using hamming weight. This algorithm would process as follows:

1. Count the number of 1-bits in the slice in base-17, and multiply the count by 2^{17}
2. Count the number of 1-bits in the slice in base-16, and multiply the count by 2^{16}
3. And so on for the remaining slices..

The `POPCNT` instruction can be used to efficiently count the number of 1-bits in a register.

BitWeaving improves upon the bit-sliced index with a new storage layout that is optimized for memory access and early pruning.

4.2 Overview

BitWeaving is an encoding scheme for columnar databases that supports efficient predicate evaluation on compressed data using SIMD. It was implemented in the Quickstep [7] engine. It aims to aggressively exploit *intra-cycle* parallelism in main-memory analytic DBMSs. The idea is that even within a single processor cycle there is abundant parallelism as the circuits in the processor core are simultaneously computing on multiple bits of information, even when working on simple ALU operations. The technique tries to exploit this parallelism to process data at or near the speed of the processor.

4.3 Mechanism

BitWeaving exploits the parallelism available at the bit level in modern processors. It does not rely on hardware-implemented SIMD capability. There are two flavors of BitWeaving:

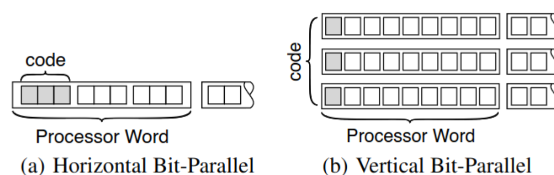


Figure 7: BitWeaving/H and BitWeaving/V layouts for column values encoded with 3 bits.

- **BitWeaving/V**: uses bit-level columnar data organization, packed into processor words. However, looking up a value can incur many CPU cache misses and hurt performance as all the bits of a value are spread out.
- **BitWeaving/H**: all the bits of a column value are stored together. This provides high performance when fetching the entire column value but uses an additional bit as a delimiter between adjacent values. In extreme cases, this flavor can be 2× slower than BitWeaving/V because of this additional bit that prevents it from storing two 32-bit values in a 64-bit processor word. Further, this flavor does not perform early pruning as that would require performing extra processing that impacts performance.

4.4 Filtering with BitWeaving: Examples

The following illustrates an example of how columnar scans with predicates can be accelerated using BitWeaving on 3-bit tuples with an 8-bit word size. The advantage of this approach is that it requires minimal instructions to evaluate a predicate on many tuples at once.

We will consider the following query on a table with 10 tuples:

```
SELECT * FROM TABLE
WHERE val < 5
```

- **BitWeaving/H:**

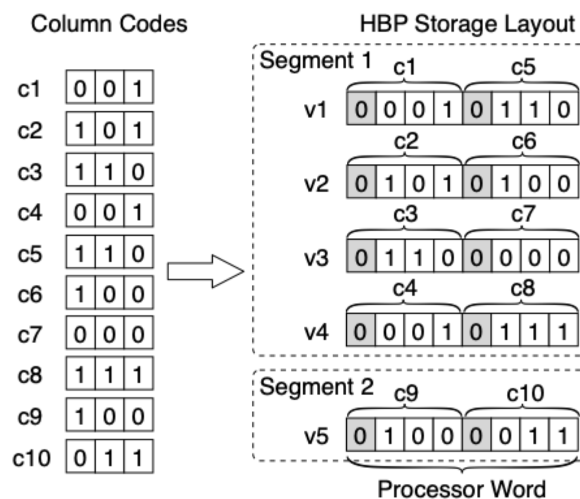


Figure 8: The BitWeaving horizontal storage layout.

The first step in the horizontal model is to pack the column values into registers with a 1-bit delimiter, initially zero, prefixed to each value. Ordering the 1st and 5th values together, then the 2nd and 6th, etc. will enable us to produce a selection vector very efficiently after the predicate is evaluated.

$$
\begin{aligned}
& && v_1(c_1, c_5) \\
X &= && (0001\ \ 0110)_2 \\
Y &= && (0101\ \ 0101)_2 \\
mask &= && (0111\ \ 0111)_2 \\
X \oplus mask &= && (0110\ \ 0001)_2 \\
Y + (X \oplus mask) &= && (1011\ \ 0110)_2 \\
Z = (Y + (X \oplus mask)) \wedge \neg mask &= && (1000\ \ 0000)_2
\end{aligned}
$$

Figure 9: Evaluating the predicate $c < 5$.

Now we evaluate the predicate. This is done by creating a constant word Y containing two instances of the filter comparison constant 5 from the query, as well as a $mask$ with ones everywhere but the delimiter. We then take $Y + (X \oplus mask)$, which has the effect of computing $5 + (2^3 - X - 1)$ since both $X \oplus mask$ and $2^3 - X - 1$ compute the 3-bit logical complement of X. This sets the delimiter only if $X \leq 5 - 1 = 4$. Filtering the result with the complement of $mask$ leaves us with just the delimiters.

Figure 10: Generating the selection vector.

The horizontal layout requires an additional step to generate the selection vector. The original tuples were reordered in the horizontal layout such that this can be accomplished in a short series of shifting operations.

- **BitWeaving/V:**

Figure 11: The BitWeaving vertical storage layout.

The vertical model instead transposes sequential column values into a series of words divided by bit order, so the most significant bits of each value are stored in the same word. This method is not sensitive to the size of column values and does not require a delimiter.

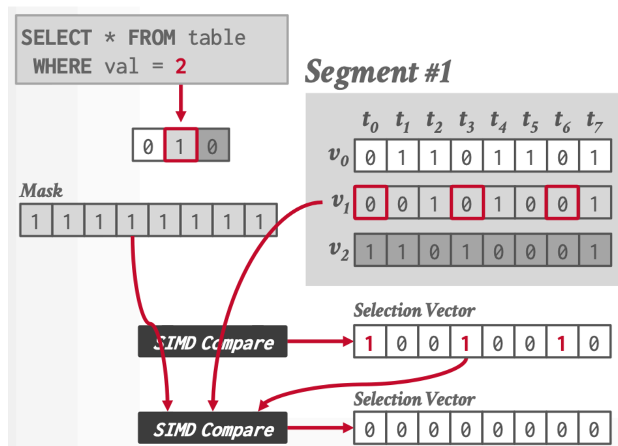


Figure 12: Evaluating the predicate $v = 2$.

> As with the horizontal layout, we construct constant-value masks to compare multiple tuples in parallel. The algorithm considers one bit position of the lookup value at a time. This time, each mask consists entirely of the bit at the position being evaluated: to compare against the value 2, we compare all most significant bits to the value 0, using `xor` and `and` operations to update the selection vector. This process is repeated for all bit positions until the end is reached or, critically, as soon as the selection vector equals zero.

Both BitWeaving/V and BitWeaving/H produce as output a result bit vector, with one bit per input tuple that indicates if the tuple matches the predicate on the column. This result bit vector must be converted to column offsets, which can either be done using iteration or a pre-computed positions table (see figure 13). BitWeaving/V outperforms BitWeaving/H for scan performance whereas BitWeaving/H achieves better lookup performance.

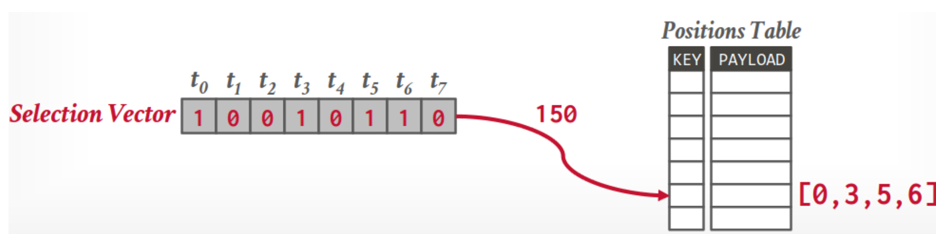


Figure 13: Pre-computed positions table

The key takeaway is that predicates can often be computed by only looking at some of the most significant bits in each column. The compact result bit vectors produced can also be used to evaluate the next stage of a complex predicate efficiently.

BitWeaving specifies bit-parallel algorithms that only require full-word operations common in all modern CPU architectures: logical and, logical or, exclusive or, binary addition, negation, and k-bit left or right shift. These methods can be used either as the primary storage organization technique in a column store database or as an indexing method to index columns.

The paper shows that BitWeaving methods outperform methods like SIMD-scanning and naive methods over all column widths, but they achieve higher speedups when the column representation has fewer bits.

This is because this allows more column predicates to be computed in parallel. With each column encoded using 4 bits BitWeaving is 20× faster than SIMD scanning. For columns wider than 12 bits, it is 4× faster.

5 Conclusion

Logical-physical data independence is one of the key aspects of the relational model. The last two lectures have covered many strategies for representing data, each with unique compute-vs-storage tradeoffs. This data independence allows applications to remain oblivious to the underlying data representation. Fixed-size data, efficient compression, and modern SIMD hardware are critical for enabling large OLAP database systems to execute highly data-parallel queries. Internal storage optimizations such as BtrBlocks, FastLanes, and BitWeaving demonstrate how DBMSs can employ seemingly bizarre physical representations of data on a variety of modern hardware to accelerate queries even at the lowest levels of computation.

References

[1] A. Afroozeh and P. Boncz. The fastlanes compression layout: Decoding $>$ 100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, may 2023. ISSN 2150-8097. doi: 10.14778/3598581.3598587. URL https://doi.org/10.14778/3598581.3598587.

[2] P. Boncz, T. Neumann, and V. Leis. Fsst: fast random access string compression. *Proc. VLDB Endow.*, 13(12):2649–2661, jul 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407851. URL https://doi.org/10.14778/3407790.3407851.

[3] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, Apr. 2015. ISSN 1097-024X. doi: 10.1002/spe.2325. URL http://dx.doi.org/10.1002/spe.2325.

[4] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM Manag. Data*, 1(2), jun 2023. doi: 10.1145/3589263. URL https://doi.org/10.1145/3589263.

[5] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 289–300, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465322. URL https://doi.org/10.1145/2463676.2465322.

[6] P. O'Neil and D. Quass. Improved query performance with variant indexes. *SIGMOD Rec.*, 26(2): 38–49, jun 1997. ISSN 0163-5808. doi: 10.1145/253262.253268. URL https://doi.org/10.1145/253262.253268.

[7] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 11(6):663–676, feb 2018. ISSN 2150-8097. doi: 10.14778/3184470.3184471. URL https://doi.org/10.14778/3184470.3184471.

[8] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. Db2 with blu acceleration: so much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, aug 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536233. URL https://doi.org/10.14778/2536222.2536233.

[9] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687671. URL https://doi.org/10.14778/1687627.1687671.