Balancing vectorized query execution with bandwidth-optimized storage

Żukowski, M.

Citation for published version (APA):
Chapter 6

Light-weight data compression

The previous chapter demonstrated how the vectorized execution model can be used to achieve high performance for main-memory processing tasks. As discussed in Section 4.3, scaling this performance to disk-resident datasets poses a challenge. The main reason is the imbalance between the continuously increasing CPU power and the relatively stagnated disk performance. Even with a scan-based processing approach, presented in Section 4.3.1, the bandwidth of a decent RAID system cannot cope with the high performance of the MonetDB/X100 execution layer.

To obtain high query performance, even with modest disk configurations, we propose new forms of lightweight data compression that reduce the I/O bandwidth need of database and information retrieval systems. Our work differs from previous use of compression in databases and information retrieval in the following aspects:

**Super-scalar Algorithms.** We contribute three new compression schemes (PDICT, PFOR and PFOR-DELTA) that are specifically designed for the superscalar capabilities of modern CPUs. In particular, these algorithms lack any if-then-else constructs in the performance-critical parts of their compression and decompression routines. Also, the absence of dependencies between values being (de)compressed makes them fully loop-pipelineable by modern compilers and allows for out-of-order execution on modern CPUs that achieve high instructions per cycle (IPC) efficiency. As a result, PFOR, PFOR-DELTA and PDICT
spend as little as 1-2 CPU cycles per 1 byte of source data, and a fraction of that during decompression. This makes them even 10 times faster than previous speed-tuned compression algorithms and allows them to improve I/O bandwidth even on RAID systems that read and write data at rates of hundreds of MB/s.

**Improved Compression Ratios.** PDICT and PFOR are generalizations of respectively dictionary and Frame-Of-Reference (FOR) or prefix-suppression (PS) compression that were proposed previously [NCR02, GRS98, WKHM00]. In contrast to these schemes, our new compression methods can gracefully handle data distributions with outliers, allowing for a better compression ratio on such data. We believe this makes our algorithms also applicable to information retrieval. In particular, we show that PFOR-DELTA compression ratios on the TREC dataset approach that of a recently proposed high-speed compression method tuned for inverted files [AM05] (“carryover-12”), while retaining a 7-fold compression and decompression speed advantage.

**RAM-CPU Cache Compression.** We make a case for compression/decompression to be used on the boundary between the CPU cache and RAM storage levels. This implies that we also propose to cache pages in the buffer manager (i.e. in RAM) in compressed form. Tuple values are decompressed at a small granularity (such that they fit the CPU cache) just-in-time, when the query processor needs them.

Previous systems [Syb] use compression between the RAM and I/O storage
levels, such that the buffer manager caches decompressed disk pages. Not only does this mean that the buffer manager can cache less data (causing more I/O), but it also leads the CPU to move data three times in and out of the CPU cache during query processing. This is illustrated by the left-most side of Figure 6.1: first the buffer manager needs to bring each recently read disk block from RAM to the CPU for decompression, then it moves it back in uncompressed form to a buffer page in RAM, only to move the data a third time back into the CPU cache, when it is actually needed by the query. As buffer manager pages are compressed, a crucial feature of all our new compression schemes is fine-grained decompression, which avoids full page decompression when only a single value is accessed.

We implemented PDICT, PFOR and PFOR-DELTA as an integral part of ColumnBM. Our experiments show that on the 100GB TPC-H benchmark, our compression methods can improve performance by a factor equal to the compression ratio in I/O constrained systems, and eliminate I/O as the dominant cost factor in most cases. We tested our compression methods both using DSM column-wise table storage [CK85] as well as a PAX layout, where data within a single disk page is stored in a vertically decomposed fashion [ADHS01]. While the TPC-H scenario favors the column-wise approach, PAX storage also strongly benefits from our compression, extending its use to scenarios where the query mix contains more OLTP-like queries.

The outline of this chapter is as follows. In Section 6.1 we relate our algorithms to previous work on database compression. Section 6.2 then introduces our new PFOR, PFOR-DELTA and PDICT compression algorithms. We use CPU performance counters on three different hardware architectures to show in detail how and why these algorithms achieve multi GB/s (de)compression speeds. We evaluate the effectiveness of our techniques using MonetDB/X100 on TPC-H in Section 6.3 as well as on information retrieval datasets from TREC and INEX in Section 6.4. Section 6.5 concludes this chapter and outlines future work.

### 6.1 Related work

Previous work on compression in database systems coincides with our goal to save I/O, which requires lightweight methods (compared with compression that minimizes storage size), such that decompression bandwidth clearly outruns I/O bandwidth, and CPU-bound queries do not suffer too great a setback by additional decompression cost. In the following, we describe a number of previously
proposed database compression schemes [WKHM00, GS91, GRS98]:

Prefix Suppression (PS) compresses by eliminating common prefixes in data values. This is often done in the special case of zero prefixes for numeric data types. Thus, PS can be used for numeric data if actual values tend to be significantly smaller than the largest value of the type domain (e.g. prices that are stored in large decimals).

Frame Of Reference (FOR), keeps for each disk block the minimum \( \min_C \) value for the numeric column \( C \), and then stores all column values \( c[i] \) as \( c[i] - \min_C \) in an integer of only \( \lceil \log_2(\max_C - \min_C + 1) \rceil \) bits. FOR is efficient for storing clustered data (e.g. dates in a data warehouse) as well as for compressing node pointers in B-tree indices. FOR resembles PS if \( \min_C = 0 \), though the difference is that PS is a variable-bitwidth encoding, while FOR encodes all values in a page with the same amount of bits.

Dictionary Compression, also called “enumerated storage” [Bon02], exploits value distributions that only use a subset of the full domain, and replaces each occurring value by an integer code chosen from a dense range. For example, if gender information is stored in a \texttt{VARCHAR} and only takes two values, the column can be stored with 1-bit integers (0="MALE", 1="FEMALE"). A disadvantage of this method is that new value inserts may enlarge the subset of used values to the point that an extra bit is required for the integer codes, triggering recompression of all previously stored values.

Several commercial database systems use data compression; especially node pointer prefix compression in B-trees is quite prevalent (e.g. in DB2). Teradata’s Multi-Valued Compression [NCR02] uses dictionary compression for entire columns, where the DBA has the task of providing the dictionary. Values not in the dictionary are encoded with a reserved \textit{exception value}, and are stored elsewhere in the tuple. Oracle also uses dictionary compression, but on the granularity of the disk block [PP03]. By using a separate dictionary for each disk block, the overflow-on-insert problem is easy to handle (at the price of additional storage size).

The use of compressed column-wise relations in our approach strongly resembles the Sybase IQ product [Syb]. Sybase IQ stores each column in a separate set of pages, and each of these pages may be compressed using a variety of schemes, including dictionary compression, prefix suppression and LZRW1 [Wil91]. LZRW1 is a fast version of common LZW [Wel84] Lempel-Ziv compression, which uses a hash table \textit{without} collision list to make value lookup during compression and decompression simpler (typically achieving a reduced compression ratio when compared to LZW). While faster than the common Lempel-Ziv compression utilities (e.g. gzip), we show in Section 6.2 that LZRW1 is still an order of
magnitude slower than our new compression schemes. Another major difference with our approach is that the buffer manager of Sybase IQ caches decompressed pages. This is unavoidable for compression algorithms like LZRW1 that do not allow for fine-grained decompression of values. Page-wise decompression fully hides compression on disk from the query execution engine, at the expense of additional traffic between RAM and CPU cache (as depicted in Figure 6.1).

An interesting research direction is to adaptively determine the data compression strategy during query optimization [CGK01, GS91, WKHM00]. An example execution strategy that optimizes query processing by exploiting compression may arise in queries that select on a dictionary-compressed column. Here, decompression may be skipped if the query performs the selection directly on the integer code (e.g. on gender=1 instead of gender="FEMALE"), which both needs less I/O and uses a less CPU-intensive predicate. Another opportunity for optimization arises when (arithmetic) operations are executed on a dictionary compressed column. In that case, it is sometimes possible to execute the operation only on the dictionary, and leave the column values unchanged [Syb] (called “enumeration views” in [Bon02]). Optimization strategies for compressed data are described in [CGK01], where the authors assume page-level decompression, but discuss the possibility to keep the compressed representation of the column values in a page in case a query just copies an input column unchanged into a result table (unnecessary decompression and subsequent compression can then be avoided).

Finally, compression to reduce the volume of transferred and memory-resident data has received significant attention in the information retrieval community, in particular for compressing inverted lists [WMB99]. Inverted lists contain all positions where a term occurs in a document (collection), always yielding a monotonically increasing integer sequence. It is therefore effective to compress the gaps rather than the term positions (Delta Compression). Such compression is the prime reason why inverted lists are now commonly considered superior to signature files as an IR access structure [WMB99]. Early inverted list compression work focused on exploiting the specific characteristics of gap distributions to achieve optimal compression ratio (e.g. using Huffman or Golomb coding tuned to the frequency of each particular term with a local Bernoulli model [Huf52]). More recently, attention has been paid to schemes that trade compression ratio for higher decompression speed [Tro03]. In Section 6.4, we show that our new PFOR compression scheme compares quite favorably with a recent proposal in this direction, the word-aligned compression scheme called “carryover-12” [AM05].
6.2 Super-scalar compression

In this section we describe how insight in extracting high IPC (Instructions Per Cycle) efficiency from super-scalar CPUs led us to the design of PFOR, PFOR-DELTA and PDICT. Figure 6.2 shows that state-of-the-art “fast” algorithms such as LZRW1 or LZOP usually obtain 200-500MB/s decompression throughput on our evaluation platform (a 2.0GHz Opteron processor). However, we aim for 2-6GB/s.

Let us first motivate the need for such speed with the following simple model (all bandwidths in GB/s):

\[
B = \text{I/O bandwidth}
\]
\[
r = \text{compression ratio}
\]
\[
Q = \text{query bandwidth}
\]
\[
C = \text{decompression bandwidth}
\]
\[
R = \text{result tuple bandwidth}
\]

Our goal with compression is to make queries that are I/O bound (i.e. \( Q > B \)) faster:

\[
R = \begin{cases} 
Br & : \frac{Br}{C} + \frac{Br}{Q} \leq 1 \quad (\text{I/O bound}) \\
\frac{Qr}{Q+C} & : \frac{Br}{C} + \frac{Br}{Q} \geq 1 \quad (\text{CPU bound}) 
\end{cases} \tag{6.1}
\]

Many datasets in e.g. data warehouses and information retrieval systems can be compressed considerably \cite{GS91, GRS98}. Section 6.3 shows that even the synthetic TPC-H dataset, with its uniform distributions, allows for good compression ratios. With these ratios, we often have \( B < Q < Br \), such that
the query becomes CPU bound using compression. Also, modern high-end RAID controllers deliver \( B > 0.6\text{GB/s} \) \cite{App06}, so with \( r = 4 \) one needs \( C = 2.4\text{GB/s} \) just to keep up with that. As we desire to spend only a minority of CPU time on decompression, we need \( C = 4.8\text{GB/s} \) to keep overhead to 50\% of CPU time, and \( C = 12\text{GB/s} \) to get it down to 20\%. Since disk bandwidth is typically shared among multiple CPU cores, these numbers motivate our goal of \( C = 2 - 6\text{GB/s} \).

We must point out that achieving such high decompression bandwidth is hard. If we assume the decoded values to be 64-bit integers, e.g. \( C = 3\text{GB/s} \) means that 400M integers must be decoded per second, such that we can spend at most five cycles per tuple on our 2.0GHz machine! This motivates our interest in getting high IPC out of modern CPUs.

### 6.2.1 Design guidelines

Our approach to super-scalar data (de)compression is similar to that of our CPU-efficient arithmetic primitives of MonetDB/X100, namely to create vectorized compression and decompression algorithms that follow these guidelines:

1. (small) arrays of values should be compressed/decompressed in a tight loop.
2. if-then-else inside the loop should be avoided;
3. the loop iterations should be kept independent.

The computational complexity of generic compression algorithms (e.g. LZW) makes it very challenging to adhere to these guidelines, while the new algorithms we propose are specifically designed to meet this challenge.

An additional guideline for our compression strategies is to allow them to work efficiently with the update mechanisms of MonetDB/X100, described in Section 4.3.4. In this architecture, the modifications are stored in in-memory delta structures, similar to differential files \cite{SL76}. The tables on disk are treated as “immutable” objects that are only updated in a batched manner. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state. As depicted in Figure 6.1, ColumnBM stores disk pages in a compressed form and decompresses them just before execution on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk, saving both cache misses and allowing more data to be cached in RAM. This approach nicely fits the delta-based update mechanism, as merging the
deltas can be applied after decompression, and blocks (see 6.2.3) need to be re-compressed only periodically.

6.2.2 PFOR, PFOR-DELTA and PDICT

All our compression methods classify input values as either coded or exception values. Coded values are represented as small integers of arbitrary bit-width $b$, with $1 \leq b \leq 24$. The bit-width used for code values is kept constant within a disk block. Exception values are stored in uncompressed form, thus they should be infrequent in order to achieve a good compression ratio.

Our compression schemes are defined as follows:

**PFOR** Patched Frame-of-Reference: the small integers are positive offsets from a base value. One (possibly negative) base value is used per disk block. Unlike standard FOR, the base value is not necessarily the minimum value in the block, as values below the base can be stored as exceptions.

**PFOR-DELTA** PFOR on value deltas: it encodes the differences between subsequent values in the column. Decompression consists of first applying PFOR and then computing the running sum on the result.

**PDICT** Patched Dictionary Compression. Integer codes refer to a position in an array of values (the dictionary). Not all values need to be in the dictionary; there can be exceptions. A disk block can contain a new dictionary but can also re-use the dictionary of a previous block.

The microbenchmarks presented throughout this section all compress 64-bit data items into 8 bits codes, but we implemented and tested our algorithms for all (applicable) datatypes and bit-widths $b$. In general, we found that, (de)compression bandwidth varies proportionally with the compression ratio.

Datasets encountered in practice are often skewed, both in terms of value distribution and frequency distribution. However, the existing FOR and dictionary compression cannot cope well with this. FOR compression needs $\lceil \log_2(max - min + 1) \rceil$ bits, and is thus vulnerable to outliers if the data (i.e. value) distribution is skewed. In contrast, our new PFOR stores outliers as exceptions, such that the $[max_{coded}, min_{coded}]$ range is strongly reduced. Similarly, dictionary compression always needs $\lceil \log_2(|D|) \rceil$ bits, even if the frequency distribution of the domain $D$ is highly skewed. In PDICT, however, infrequent values become exceptions, such that the size $|D_{coded}|$ of the frequent domain is strongly reduced on skewed frequency distributions.
6.2.3 Disk storage

As discussed in Section 4.3, ColumnBM stores data in blocks, grouped in large chunks. Blocks contain one or more segments. In case of column-wise storage, a segment is identical to a block. In case of PAX [ADHS01], a block contains a segment for each column, and all segments in the block contain the same number of values, which implies that these segments may have different byte-sizes (that sum to a number close to the block size).

Uncompressed fixed-width data types are stored in a segment as a simple array of values.\(^1\) Figure 6.3 shows the structure of a compressed segment that divides the segment in four sections:

- a fixed-size header that contains compression-method specific info as well as the sizes and positions of the other sections.
- the entry point section that allows for fine-grained tuple access. For every 128 values, it contains an offset to the next exception in the code section, and a corresponding offset in the exception section.
- the code section is a forward-growing array with one small integer code for each encoded value. This section takes the majority of the space in the block.
- the exception section, growing backwards, stores non-compressed values that could not be encoded into a small integer code.

6.2.4 Decompression

A pre-processing step in decompression is bit-unpacking: the transformation of \( b \) bits-wide code patterns in the disk block into an array of machine-addressable integers. Symmetrically, we use bit-packing as post-processing for compression. These phases take only a moderate fraction of our algorithms cost, thanks to using highly optimized routines that are loop-unrolled to handle 32 values each iteration:

\[
/* \text{example: routine to unpack 12-bits codes into integers} */
void UNPACK12(unsigned int* out, unsigned int* in, int n) {
    for(int i = 0, j = 0; i < n; i += 8, j += 3) {
        out[i + 0] = ((in[j + 0] & 0x00000fff) >> 0);
        out[i + 1] = ((in[j + 0] & 0x00fff000) >> 12);
    }
}\]

\(^1\)Variable-width data types such as strings are stored in two segments: one byte-array that contains all values concatenated and a segment with integer offsets to their start positions.
Figure 6.3: Compressed Segment Layout (encoding the digits of $\pi$: 31415926535897932 using 3-bit PFOR compression)

```c
out[i + 2] = ((in[j + 0] & 0xff000000) >> 24) | (in[j + 1] & 0x000000ff);
out[i + 3] = ((in[j + 1] & 0x0000ffff) >> 4);
out[i + 4] = ((in[j + 1] & 0xff000000) >> 16);
out[i + 5] = ((in[j + 1] & 0xf0000000) >> 28) | (in[j + 2] & 0x000000ff);
out[i + 6] = ((in[j + 2] & 0x000fff00) >> 8);
out[i + 7] = ((in[j + 2] & 0xfff00000) >> 20);
```

The naive way to implement any decompression scheme that distinguishes between coded and exception values, is to use a special code (MAXCODE) for exceptions, and continuously test for it while decompressing:

```c
/* NAIVE approach to decompression */
for (i = j = 0; i < n; i++)
    if (code[i] < MAXCODE)
        output[i] = DECODE(code[i]);
    else
        output[i] = exception[--j]);
```

The above decompression kernel is applicable to both PFOR and PDICT, though the way they encode/decode values differs. In our pseudo code, we abstract from these differences using the following macros: (i) int ENCODE(ANY), which transforms an input value into a small integer, and (ii) ANY DECODE(int), which produces the encoded input value given a small integer code.

The problem with the NAIVE approach is that it violates our guideline to avoid if-then-else in the inner loop. This hinders loop pipelining by the compiler, and also causes branch mispredictions when the else-branch is taken (assuming exceptions are the less likely event). The bottom-left part of Figure 6.4
demonstrates most clearly on Pentium4 how NAIVE decompression throughput rapidly deteriorates as the exception rate gets nearer to 50%. The cause are branch mispredictions\(^2\) on the if-then-else test for an exception, that becomes impossible to predict. In the graph on top, we see that the IPC takes a nosedive to 0.5 at that point, showing that branch mispredictions are severely penalized by the 31 stage pipeline of Pentium4.

To avoid this problem, we propose the following alternative “patch” approach:

```c
int Decompress<ANY>( int n, int bitwidth,
    ANY *__restrict__ output,
    void *__restrict__ input,
    ANY *__restrict__ exception,
    int *next_exception)
{
    int next, code[n], cur = *next_exception;

    UNPACK[bitwidth](code, input, n); /* bit-unpack the values */
    /* LOOP1: decode regardless of exceptions */
```

\(^2\)We collected IPC, cache misses, and branch misprediction statistics using CPU event counters on all test platforms.
for (int i = 0; i < n; i++) {
    output[i] = DECODE(code[i]);
}

/* LOOP2: patch it up */
for (int i = 1; cur < n; i++, cur = next) {
    next = cur + code[cur] + 1;
    output[cur] = exception[-i];
}

*next_exception = cur - n;
return i;

Different from the NAIVE method, decompression is now split in two tight loops without any if-then-else statements, which both can be efficiently optimized by a compiler and executed on superscalar CPUs.

Figure 6.3, depicting the integer sequence of \( \pi \) stored using 3-bit PFOR with \( \min_{\text{coded}} = 0 \), shows that all exception values (i.e. digits \( r_{qual} \geq 8 \)) use their code value to store an offset to the next exception, forming a linked list.

The first loop simply decodes all values, which will generate wrong values for the exceptions. The second loop then patches up the incorrect values by walking the linked exception list and copying the exception values into the output array. The idea of patching rather than escaping exception values is central to our new algorithms, hence the “P” in their name derives from it.

Following the linked list during patching violates our guideline that one iteration should be independent of the previous one. Iterating the list poses a data hazard to the CPU, however, and not a control hazard, such that it is not very expensive. Moreover, the second loop processes only a small percentage of values, and the data it updates is in the CPU cache. This makes its overhead easily amortized by the performance improvement of the first loop.

The results in Figure 6.4 show that the performance of our patching algorithms decreases monotonically with increasing exception rates. Contrary to the NAIVE approach, decompression bandwidth degrades roughly proportionally with the compression ratio, or the size of the compressed data, as one would expect. The relatively flat IPC lines suggest that the overhead of the data dependency in LOOP2 is negligible with respect to the increase in memory traffic.

This does not hold for the NAIVE kernel, for which on Pentium4 and Opteron we observe a clear increase in decompression bandwidth towards an exception rate of one. This suggests that its performance is not determined by the size of the compressed data, but by branch mispredictions in the CPU, as both decompression bandwidth and IPC follow the inverse of the bell-shaped branch misprediction curve.
Section 6.2: Super-scalar compression

On Itanium2, the branch mispredictions are avoided thanks to branch predication explained in Section 2.1.5.2. As a result, the performance of the NAIVE kernel closely tracks that of PFOR and PDICT, as presented in the rightmost graph in Figure 6.4. Overall, the patching schemes are clearly to be preferred over the NAIVE approach, as they are faster on all tested architectures.

6.2.5 Compression

Previous database compression work mainly focuses on decompression performance, and views compression as a one-time investment that is amortized by repeated use of the compressed data. This is caused by the low throughput of compression, often an order of magnitude slower than decompression (see Figure 6.2), such that compression bandwidth is clearly lower than I/O write...
bandwidth. In contrast, our super-scalar compression can be used to accelerate I/O bound data materialization tasks. In OLAP and data mining environments, such materialization happens quite frequently for sorting, ad-hoc joins that require partitioning, or (view) materialization of intermediate results that are re-used by a subsequent query batch. Efficient compression is also important for re-compression of data chunks occurring in case of updates. Note that I/O write bandwidth tends to be considerably lower than read bandwidth, especially on RAID devices with mirroring. Therefore, the design goal of compression throughput can be lower than for decompression, e.g. 1-2GB/s. The bottom graphs in Figure 6.5 show that PFOR compression meets this target on all our test platforms.

To achieve such high throughput, we again use the principle of avoiding if-then-else in the inner loop. The first loop uses a temporary array miss to make a list of exception positions. The second loop constructs the linked patch list and copies the exception values.

```c
int Compress<ANY>( int n, int bitwidth,
    ANY *__restrict__ input,
    void *__restrict__ code,
    ANY *__restrict__ exception,
    int *__lastpatch)
{
    int miss[N], data[N], prev = *lastpatch;

    /* LOOP1: find exceptions */
    for (int i = 0, j = 0; i < n; i++) {
        int val = ENCODE(input[i]);
        data[i] = val;
        miss[j] = i;
        j += (val > MAXCODE);
    }

    /* LOOP2: create patchlist */
    for (int i = 0; i < j; i++) {
        int cur = miss[i];
        exception[-1 - i] = input[cur];
        data[prev] = (cur - prev) - 1;
        prev = cur;
    }

    PACK[bitwidth](code, data, n); /* bit-pack the values */
    *lastpatch = prev;
    return j; /* #exceptions */
}
```

Appending a position to the miss list without if-then-else uses a technique similar to the one used in [Ros02] for selection computation: the current position is always copied to the end of the list, and the list pointer is incremented with
a Boolean value. This technique transforms a control dependency into a data dependency, which is more efficient. Still, the presence of a data dependency on the variable \( j \) in the first, performance-critical loop, violates our guideline that iterations should be independent. Data dependencies cause delay slots in the CPU pipeline. The left-upper graph of Figure 6.5 shows that Pentium4 has an IPC of \(< 1\). We can try to improve IPC by offering it more independent work using a technique called **double-cursor**. It runs two cursors through the to-be-encoded values, one from the start, and one from halfway. Two independent miss lists are used to detect exceptions, processed one after the other in the sequel (omitted):

```c
/* LOOP1a: find exceptions */
int m = n / 2;
for(int i = 0, j_0 = 0, j_m = 0; i < m; i++) {
    int val_0 = ENCODE(input[i + 0]);
    int val_m = ENCODE(input[i + m]);
    code[i + 0] = val_0;
    code[i + m] = val_m;
    miss_0[j_0] = i + 0;
    miss_m[j_m] = i + m;
    j_0 += (val_0 > MAXCODE);
    j_m += (val_m > MAXCODE);
}
```

Double-cursor is not the same as loop-unrolling, and cannot be introduced automatically by the compiler.

Figure 6.5 shows that double-cursor significantly improves the IPC and throughput of PFOR on Pentium4, while it behaves the same as single-cursor PFOR on Opteron. On Itanium, where single-cursor already achieved a very high IPC (4), performance degrades somewhat. As the gains on Pentium4, which is also the more prevalent, outweigh the loses on Itanium, double-cursor can be considered the overall winner.

### 6.2.6 Fine-grained access

While we anticipate that most performance-intensive queries will decompress all values in a compressed segment sequentially, some queries may perform random value accesses. A random lookup in the buffer manager will likely cause a CPU cache miss, so if decompression overhead stays in the same ballpark as DRAM access (i.e. 150-400 CPU cycles per cache miss), we deem it efficient enough.

It is easy to randomly access the code section at any position \( x \), but we should also know whether position \( x \) is an exception and if so, where in the exception section the real value is stored. For this purpose, the *entry point*
section keeps a pointer to the next exception, as well as its position in the
exception section, for each position that is an exact multiple of 128. Each entry
point, stored once every 128 values, is a combination of a 7-bits patch start_list
and a 25-bits start_exception, hence the storage overhead of fine-grained access
is \( \frac{32}{128} = 0.25 \) bits per value. Note that 25-bits exception codes limit our
segments to a maximum of 32MB, which is more than sufficient for now to
obtain high sequential bandwidth on any RAID system. We can obtain the
value at position \( x \) in the block, as follows:

```c
ANY finegrained_decompress(int position,
   int*__restrict__ code,
   ANY*__restrict__ exception,
   entry_t*__restrict__ entry)
{
   int i = entry[position >> 7].start_list + position & ~127;
   int j = entry[position >> 7].start_exception;
   while(i < x) {
      i += code[i];
      j--;
   }
   return (i == x) ? exception[j] : DECODE(code[x]);
}
```

This tight pipelinable loop that walks the linked list takes 8, 9 and 11 cycles
per iteration on respectively the Opteron, Itanium2 and Pentium4 CPUs. Even
in the worst realistic case of 30% exceptions, it thus takes on average only a
limited \( 128 \times 0.3 / 2 = 19 \) number of iterations on average, such that random
access decoding takes around 200 CPU work cycles per value.

In case of PFOR-DELTA, we must also store the current running total for
each entry point. Sticking with 64-bit integers, this induces an additional storage
overhead of 0.75 bit per value. Also, fine-grained PFOR-DELTA access requires
decompressing a vector of 128 values (which usually causes one cache miss in
both the code and exception sections, bringing memory access cost to 300-800
cycles). Since our decompression algorithms typically spend between 3-6 cycles
per value, uncompressing 128 values is in the same order of cost.

## 6.2.7 Compulsory exceptions

A complication of patching is that the compressed integer codes only have a
range from \([0,2^b-1]\); hence the maximum distance between elements in the linked
list of exceptions is \(2^b\). If gaps exceeding this distance occur, so-called compulsory
exceptions must be introduced. A compulsory exception is a value that can be
Section 6.2: Super-scalar compression

Figure 6.6: Impact of the compulsory exceptions on the real exception rate $E'$ for $b \leq 4$

compressed but is represented as an exception anyway, just in order to use its code value to keep the exception list connected.

We do not always have to insert compulsory exceptions if the gap is larger than $2^b$ though. Each entry point starts a new exception list, and these lists need not be connected to each other. Thus, gaps between exceptions at the start and end of each 128-value sequence never need compulsory exceptions. This effectively reduces the area in the code section that must be “covered” by a linked exception list per 128 values by $1/E$, where $E$ is the exception rate caused by the data distribution. From this, we can compute $E'$, which is the effective exception rate after taking into account compulsory exceptions as $E' = \text{MAX}(E, \frac{128E - 12b}{128E})$. Figure 6.6 shows that with bit-width $b = 1$ for miss rates $E > 0.01$, the effective exception rate $E'$ quickly increases to a rather useless 0.47. With $b = 2$, it goes to an already more usable $E' = 0.22$, while for all bit-widths $b > 4$, the effect of compulsory exceptions is negligible.

6.2.8 RAM-RAM vs. RAM-cache decompression

Figure 6.7 presents the results of a micro-benchmark conducted to evaluate our choice for fine-grained, into-cache decompression, as opposed to decompression on the granularity of disk pages. Into-cache decompression is achieved by decompressing a page on a per-vector basis, always storing the result in the same cache-resident result vector, overwriting any previous results. In the page-wise approach, the full, uncompressed page is materialized in RAM.

Results show that RAM-cache decompression is much more efficient than RAM-RAM decompression. Performance of the former approach degrades with the exception rate, and thus the size of the compressed data. The flat shape
of the latter approach suggests that performance is constrained by the need to materialize the uncompressed result, which is always constant in size.

Another benefit of the RAM-cache approach is that the cache-resident result vector can be fed directly into an operator pipeline. In the RAM-RAM approach, the uncompressed page needs to be read back into the CPU, presenting an additional overhead which is not even incorporated in the RAM-RAM results from Figure 6.7.

6.2.9 Improving memory bandwidth on multi-core CPUs

Algorithms presented in this chapter were originally designed to improve the delivery speeds for disk-resident data. However, disk is not the only media where bandwidth becomes the bottleneck. Even in main memory, with high data consumption rates, as is the case in MonetDB/X100, there are cases when RAM bandwidth limits the performance. This is becoming especially important with the recent popularity of multi-core CPUs (see Section 2.1.7.2). There, memory bandwidth needs to be shared among multiple cores, and with the number of cores continuously increasing, providing enough bandwidth for each core becomes a problem. To demonstrate it, we have performed an experiment in which we run two versions of the TPC-H Query 6 on a dual-chip Core2 Quad 2GHz machine (8 cores in total), using both uncompressed and compressed table rep-
Figure 6.8: Performance of two versions of TPC-H Query 6 (with and without predicates), on a dual-chip Core2 Quad 2GHz (8 cores in total), using uncompressed and compressed table representation.

The first version (A) is the original query, scanning 4 columns, the second (B) is the query with all the predicates removed, scanning only two columns. We used scale-factor 1, making the data fully RAM-resident. In this setup, the original 4 columns occupy 144MB uncompressed and 37.5MB compressed (4-byte wide \texttt{l\_shipdate} and \texttt{l\_quantity} and 8-byte wide \texttt{l\_extendedprice} and \texttt{l\_discount}). The two columns used in version B are 96MB and 21MB, respectively. The original Query 6 is as follows:

```sql
SELECT sum(l\_extendedprice \* l\_discount) AS revenue
FROM lineitem
WHERE l\_shipdate \geq date '1994-01-01'
AND l\_shipdate < date '1995-01-01'
AND l\_quantity < 24;
AND l\_discount BETWEEN 0.05 and 0.07
```

Figure 6.8 presents the results for benchmarks in which we run from 1 to 8 parallel streams of 300 queries, and measure the average query time of the middle 100 queries. For the original query, the uncompressed version always beats the compressed one. This is because the used combination of predicates selects only ca. 2% of the original data, and the compressed run always decompresses all the values, even the ones not used in the actual computation. Also, the impact
of performance on multiple cores is relatively small in the original case without compression. This is again related to the low selectivity of predicates: only a part of the data is actually touched by the query. Also, in this query, the memory-intensive primitives are overlapped with primitives reading the data from the CPU cache, allowing better use of the bus.

The situation changes dramatically when we look at the version B, which does not include predicates and really consumes a full 96MB. Profiling shows that, in the uncompressed case, the majority of time is spent in the memory-intensive multiplication primitive, computing \( \text{l\_extendedprice} \times \text{l\_discount} \). This computation for 6 millions tuples takes on average 33ms. Since the primitive consumes 16 bytes for each tuple, its bandwidth requirement is almost 3 GB/s (per core), which is too high for the memory infrastructure. In the compressed case, most of the time is spent in the decompression phase, which takes 16 ms, resulting in the RAM consumption rate of 1.7 GB/s, and the decompression rate of 6 GB/s. The uncompressed data is in the CPU cache, and the following multiplication primitive only takes 8ms (12 GB/s data input, only available in the CPU cache). This allows the compressed version to improve the performance even on a single core. With more cores, the no-predicate version suffers even more from the insufficient memory bandwidth, making the improvement thanks to the data compression as high as 400%.

This experiment demonstrates that data compression can significantly improve query performance by reducing the main-memory bandwidth requirements. With future CPU generations having dozens, if not hundreds of cores, compressing data in RAM might become more and more important.

### 6.2.10 Choosing compression schemes

The table materialization operator in MonetDB/X100 should automatically decide which compression method to use for each disk block, and with what parameters. The idea is to first gather a sample (e.g. \( s=64\text{K} \) values) and look for the best settings for all applicable schemes. For numeric data types (e.g. integers, decimals) all three schemes apply. Otherwise, only PDICT is usable.\(^3\)

When a column is being compressed, the compression ratio can be easily monitored at the granularity of a disk block or chunk. When it strongly deteriorates, we could re-run the compression mode analysis to adapt the parameters for the next block or even choose another compression scheme. The complexity

---

\(^3\)In the near future, we plan to add new super-scalar compression algorithms targeted at floating point data and text.
of choosing a compression mode is $O(s \log s)$ to the size of the sample $s$, because it must be sorted as a preprocessing step. We now discuss for each method, how the optimal parameters are found using the sorted sample.

In PFOR, we can determine in one pass through the sorted sample where the longest stretch of values starts, such that the difference between first and last is representable in $b$ bits.

```
PFOR_ANALYZE_BITS(int n, ANY *V, int b) {
    int len = 0, min = 0, range = 1 << b;
    for (int lo = 0, hi = 0; hi < n; hi++)
        if (V[hi] - V[lo] >= range) {
            if (hi - lo > len) {
                min = lo; len = hi-lo;
            }
            while (V[hi] - V[lo] >= range)
                lo++;
        }
    return (min, len + 1);
}
```

We simply invoke this function for all relevant bit-widths $b$ and choose the setting that yields best compression, i.e. $1 \leq b < 8 \cdot \text{sizeof}(V)$ where $b + E_{PFOR(b)} \cdot 8 \cdot \text{sizeof}(V)$ is minimal. In this equation the exception rate $E_{PFOR(b)} = \frac{s - len_b}{s}$, where $len_b$ is returned by the above function, when invoked on the sample with parameter $b$.

The parameters for PFOR-DELTA are derived by running this same algorithm on the sorted differences of the sample.

For PDICT, we use once again the sorted sample, to create a (smaller) frequency histogram $h$, which we re-sort descending on frequency. PDICT will encode the first (i.e. largest) $2^b$ buckets of this histogram such that the exception rate $E_{PDICT(b)} = 1 - \sum_{i=1}^{2^b} \frac{h[i]}{s}$. Again, by trying all relevant settings of $b$, we can quickly determine the $b$ that yields the highest compression rate. The first $2^b$ values from the histogram are subsequently used to create a super-scalar perfect hash function (memorized probing in [Hem05]) that is used during PDICT compression to compute the integer codes for values that must be compressed. In all, it achieves PDICT compression bandwidth of $> 1GB/s$ on all our three test platforms.
6.3 TPC-H experiments

Table 6.1 shows the performance of compression algorithms in MonetDB/X100 running the TPC-H benchmark [Tra06] with scale factor 100 on two different hardware platforms. Low-end servers are represented by an Opteron 2GHz machine with a 4-disk RAID system delivering around 80 GB/s. The example of a middle-end system is a Pentium4 machine with 12-disk RAID delivering around 350GB/s. Both machines are dual CPU systems with 4GB memory, but MonetDB/X100 currently uses only one CPU.

We used the same data clustering and index structures as in the previous in-memory MonetDB/X100 TPC-H SF-100 experiments [BZN05]. Only a subset of TPC-H queries is presented, since the X100 execution layer currently misses some of the features necessary to run the remaining ones in a disk-based scenario.

While ColumnBM by default uses the DSM storage model [CK85], we also present the results for PAX storage [ADHS01]. I/O-wise they are comparable to an NSM system running DB2, for which the last column lists the official TPC-H scores. This system uses eight Pentium4 Xeon CPUs (2.8GHz), 16GB RAM and 142 SCSI disks. Thus, while the CPU used is roughly equivalent to our middle-end server, it has 4-12x more hardware resources across the board.

The TPC-H data was compressed using PFOR, PFOR-DELTA and PDICT (enum) compression schemes. The second and third columns of Table 6.1 show the compression ratios achieved per query. Note that since the “comment” fields could not be compressed with our algorithms, the PAX queries achieve significantly lower compression ratios. Columns 4 and 10 show that in most cases we reach our decompression speed target of $> 2 \text{ GB/s}$.

On the Opteron system, the speedup for most of the DSM queries is in line with the compression ratio. As the left-most side of Figure 6.9 shows, this is related to the fact that the low-end disk system makes the queries I/O-bound even with compression. The middle part of Figure 6.9 shows that on the Pentium4 system with a faster RAID the situation is different with much higher CPU usage in the uncompressed case. As a result, after increasing the perceived I/O bandwidth with decompression, all the queries become CPU-bound, such that the performance gain is less than the compression ratio. With the PAX storage model and its increased I/O requirements, the CPU processing impact is reduced again, resulting in better speedups than in the DSM case.

We also implemented the possibility to perform full-page decompression in ColumnBM, as described in Section 6.2.8. Column 7 of Table 6.1 shows that such decompression from memory into memory is significantly slower than the fine-grained decompression between RAM and the CPU Cache, presented in col-


### TPC-H Compression

#### Pentium4 Xeon

<table>
<thead>
<tr>
<th>Query</th>
<th>Compression Ratio</th>
<th>12 disks</th>
<th>4GB RAM</th>
<th>Dec. Speed</th>
<th>DS M</th>
<th>PAX</th>
<th>DS M</th>
<th>Unc.</th>
<th>M ⇒ C</th>
<th>M ⇒ M</th>
<th>Unc.</th>
<th>M ⇒ C</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>4.33 2.30</td>
<td>4502</td>
<td>65.9</td>
<td>50.9</td>
<td>63.8</td>
<td>265.0</td>
<td>103.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>3.04 1.66</td>
<td>2306</td>
<td>8.9</td>
<td>6.0</td>
<td>7.1</td>
<td>45.5</td>
<td>27.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>8.15 1.82</td>
<td>3709</td>
<td>4.8</td>
<td>1.8</td>
<td>2.3</td>
<td>30.2</td>
<td>16.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>3.81 2.24</td>
<td>2421</td>
<td>17.2</td>
<td>16.2</td>
<td>16.7</td>
<td>81.2</td>
<td>36.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>4.39 2.25</td>
<td>2200</td>
<td>10.8</td>
<td>4.6</td>
<td>6.1</td>
<td>51.0</td>
<td>22.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>1.71 2.01</td>
<td>1457</td>
<td>34.4</td>
<td>40.8</td>
<td>48.3</td>
<td>158.0</td>
<td>76.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2.14 1.08</td>
<td>4084</td>
<td>18.8</td>
<td>18.5</td>
<td>19.4</td>
<td>38.8</td>
<td>35.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1.91 1.94</td>
<td>3688</td>
<td>5.8</td>
<td>4.9</td>
<td>5.4</td>
<td>22.1</td>
<td>11.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>2.70 2.13</td>
<td>2584</td>
<td>30.3</td>
<td>31.2</td>
<td>31.3</td>
<td>49.6</td>
<td>40.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3.56 2.75</td>
<td>4315</td>
<td>38.9</td>
<td>13.6</td>
<td>21.3</td>
<td>419.9</td>
<td>151.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>4.11 2.12</td>
<td>2600</td>
<td>43.2</td>
<td>24.2</td>
<td>32.1</td>
<td>338.0</td>
<td>157.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Opteron

<table>
<thead>
<tr>
<th>Query</th>
<th>Compression Ratio</th>
<th>2GHz</th>
<th>4 disks</th>
<th>Dec. Speed</th>
<th>DS M</th>
<th>PAX</th>
<th>IBM DB2</th>
<th>UDB 8.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>4.33 2.30</td>
<td>3736</td>
<td>307.2</td>
<td>69.6</td>
<td>1098.9</td>
<td>480.3</td>
<td>111.9</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>3.04 1.66</td>
<td>2546</td>
<td>35.0</td>
<td>11.3</td>
<td>183.5</td>
<td>131.6</td>
<td>15.1</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>8.15 1.82</td>
<td>3018</td>
<td>18.2</td>
<td>2.4</td>
<td>115.5</td>
<td>65.9</td>
<td>12.5</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>3.81 2.24</td>
<td>2119</td>
<td>54.3</td>
<td>15.3</td>
<td>300.1</td>
<td>155.9</td>
<td>84.0</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>4.39 2.25</td>
<td>2031</td>
<td>48.2</td>
<td>10.7</td>
<td>232.7</td>
<td>104.3</td>
<td>17.1</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>1.71 2.01</td>
<td>1251</td>
<td>119.8</td>
<td>72.0</td>
<td>614.2</td>
<td>349.4</td>
<td>86.5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2.14 1.08</td>
<td>3225</td>
<td>27.0</td>
<td>14.6</td>
<td>180.9</td>
<td>162.2</td>
<td>19.5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1.91 1.94</td>
<td>2888</td>
<td>23.7</td>
<td>12.2</td>
<td>90.6</td>
<td>46.9</td>
<td>10.9</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>2.70 2.13</td>
<td>2464</td>
<td>44.9</td>
<td>22.4</td>
<td>209.8</td>
<td>97.1</td>
<td>21.6</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3.56 2.75</td>
<td>3833</td>
<td>181.9</td>
<td>50.6</td>
<td>1379.7</td>
<td>704.9</td>
<td>318.2</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>4.11 2.12</td>
<td>2520</td>
<td>197.6</td>
<td>46.6</td>
<td>1423.5</td>
<td>759.2</td>
<td>374.9</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:** unc. = uncompressed data, 
M ⇒ C = memory-to-cache decompression, M ⇒ M = memory-to-memory decompression

Table 6.1: TPC-H SF-100 experiments on MonetDB/X100 (except DB2 results, taken from www.tpc.org)
Figure 6.9: Profiling details for the TPC-H SF-100 results from Table 6.1
6.4 Inverted file compression

Compression of inverted files to improve I/O bandwidth and sometimes latency (due to reduced seek distances on the compressed file) is important for the performance of information retrieval systems [WMB99]. In this area, there is a trend to use lightweight-compression schemes rather than the classical storage-optimal schemes [Tro03, AM05].

We evaluated the performance of PFOR-DELTA with respect to both compression ratio and speed on inverted file data derived from the INEX and TREC document collections, and compared it with the implementation of the recently proposed carryover-12 compression scheme [AM05], which was designed for high decompression speeds. Furthermore, performance was compared to that of a semi-static Huffman coder, which is commonly used for inverted file compression. Table 6.2 summarizes the results on our 3GHz Pentium 4 machine, and shows that PFOR-DELTA improves decompression bandwidth of carryover-12 6.5 times, while only reducing the compression ratio by 15%.

To verify the need for such decompression speeds, we measured the raw query bandwidth of a typical retrieval query that looks up the top-N documents in which a given term from the TREC fbis dataset occurs most frequently (a merge-join of the postings table with the document offsets, followed by ordered aggregation and heap-based top-N). Within our MonetDB/X100 system, this

<table>
<thead>
<tr>
<th></th>
<th>PFOR-DELTA</th>
<th>carryover-12</th>
<th>shuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp ratio</td>
<td>comp MB/s</td>
<td>comp MB/s</td>
<td>comp MB/s</td>
</tr>
<tr>
<td>INEX</td>
<td>1.75</td>
<td>679</td>
<td>3053</td>
</tr>
<tr>
<td>TREC fbis</td>
<td>3.47</td>
<td>788</td>
<td>3911</td>
</tr>
<tr>
<td>TREC fr94</td>
<td>3.12</td>
<td>682</td>
<td>3196</td>
</tr>
<tr>
<td>TREC ft</td>
<td>3.13</td>
<td>761</td>
<td>3443</td>
</tr>
<tr>
<td>TREC latimes</td>
<td>2.99</td>
<td>742</td>
<td>3289</td>
</tr>
</tbody>
</table>

Table 6.2: PFOR-DELTA on Inverted Files

The main reason for this is the high-cost of in-memory materialization of decompressed data. Another interesting feature of our fine-grained decompression can be observed in Figure 6.9, where the processing in the compressed case is slightly faster. This is caused by the fact that the main-memory access is performed by the decompression routines, and the query execution layer reads the data directly from the CPU cache.
query was able to process a list of \( d \)-gaps at 580MB/s, which implies that even on our 350MB/s RAID system it would remain I/O-bound. Using equation 6.1 to compute the decompression bandwidth \( C \) that achieves an equilibrium between CPU time spent on query processing and decompression, yields \( \frac{580 \times C}{580 + C} = 350 \), which leads to \( C = 883\) MB/s. Table 6.2 shows that decompression bandwidths from shuff and even carryover-12 are below this point, hence only make the query slower, while PFOR-DELTA accelerates it from 350MB/s to 504MB/s.

The benefit of using PFOR-DELTA on IR tasks has been evaluated with Terabyte-TREC experiments discussed in Section 8.2.2. Additionally, further research by Zhang et al [ZLS08] has demonstrated that this technique is competitive against a large class of other methods, and the ideas behind it can also be used to improve other compression algorithms.

6.5 Conclusions and future work

This chapter presented our work on using data compression to scale the high performance of MonetDB/X100 engine to disk-based datasets. We proposed a new set of super-scalar compression algorithms. Their “patching” approach allows these algorithms to handle outliers gracefully while still exploit the pipelined features of modern CPUs. Additionally, we introduced the idea of decompressing between RAM and the CPU Cache, rather than the common idea to apply it between I/O and RAM. Our results show that this not only allows the buffer manager to store more (compressed) data, but is also faster to (de)compress. As a result, our algorithms provide decompression speeds in the range of > 2GB/s. This is an order of magnitude faster than conventional compression algorithms, making decompression almost transparent to query execution. By using these techniques in TPC-H, TREC and INEX datasets, we managed to significantly reduce or completely eliminate the I/O bottleneck. In the future, we plan to extend the applicability of our system by introducing additional compression algorithms specialized for other data types and distributions.