Super-Scalar RAM-CPU Cache Compression

Marcin Zukowski, Sándor Héman, Niels Nes, Peter Boncz
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
{M.Zukowski, S.Heman, N.Nes, P.Boncz}@cwi.nl

Abstract

High-performance data-intensive query processing tasks like OLAP, data mining or scientific data analysis can be severely I/O bound, even when high-end RAID storage systems are used. Compression can alleviate this bottleneck only if encoding and decoding speeds significantly exceed RAID I/O bandwidth. For this purpose, we propose three new versatile compression schemes (PDICT, PFOR, and PFOR-DELTA) that are specifically designed to extract maximum IPC from modern CPUs.

We compare these algorithms with compression techniques used in (commercial) database and information retrieval systems. Our experiments on the MonetDB/X100 database system, using both DSM and PAX disk storage, show that these techniques strongly accelerate TPC-H performance to the point that the I/O bottleneck is eliminated.

1 Introduction

High-performance data processing tasks like OLAP, data mining, scientific data analysis and information retrieval need to go through large amounts of data, causing a need for high I/O bandwidth. One way to provide this bandwidth is to use large multi-disk (RAID) systems. Table 1 lists the hardware configurations used in the current official TPC-H [15] results on the 100GB database size (for the most common case of a 4-CPU system). These configurations contain 42 to 112 disks, connected through a high-end storage infrastructure. A disadvantage of this brute-force approach of meeting I/O throughput requirements is a high system cost. The last column of Table 1 shows that between 61% and 78% of hardware price can be attributed to disk storage.

In real-life, though, it is highly questionable whether databases of "only" 100GB are likely to get stored on 100 drives or more, as in these TPC-H benchmark setups. Using an order of magnitude more disks than required for storage size alone is unsustainable on large data sizes due to the hardware cost, expensive maintenance and increased failure rate. Currently, it is cheaper and more manageable to store a database of 100GB on a RAID of 4-12 drives, which can be accommodated inside a server case. However, such a more realistic configuration delivers significantly less I/O throughput; which necessarily leads to real-life OLAP performance that is lower than suggested by the official TPC-H results.

CPUs	RAM	Disks
4xPower5 1650MHz (9%)	32GB (13%)	42x36GB=1.6TB (78%)
4xItanium2 1500MHz (24%)	32GB (15%)	112x18GB=1.9TB (61%)
4xXeon MP 2800MHz (25%)	4GB (3%)	74x18GB=1.2TB (72%)
4xXeon MP 2000MHz (30%)	8GB (7%)	85x18GB=1.6TB (63%)

Table 1. TPC-H 100GB Component Cost.

1.1 Contributions

To obtain high query performance, even with more 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 super-scalar 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-pipelinable* by modern compilers and allows for *out-of-order* execution on modern CPUs that achieve high Instructions Per Cycle (IPC) efficiency. On current hardware, PFOR, PFOR-DELTA and PDICT compress more than a GB/s, and decompress a multitude of that, which makes them more than 10 times faster than previous speed-tuned compression algorithms. This 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 [10, 7, 18]. 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 [2] ("carryover-12"), while retaining a 7-fold compression and decompression speed advantage.

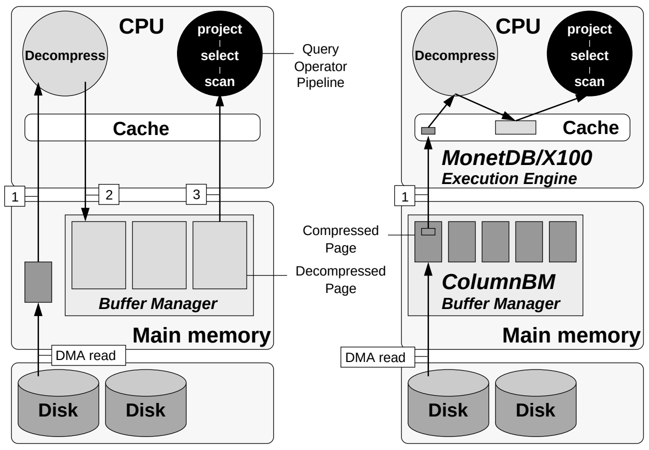

Figure 1. I/O-RAM vs RAM-CPU compression.

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 [14] 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 side of Figure 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 in the new ColumnBM storage manager of the MonetDB/X100 system [3], developed at the CWI for OLAP, data mining and multimedia/information retrieval applications. Our experiments show that on the 100GB TPC-H benchmark, our compression methods can improve performance with 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 [6] as well as a PAX layout, where data within a single disk page is stored in a vertically decomposed fashion [1]. 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.

1.2 Outline

In Section 2 we relate our algorithms to previous work on database compression. We also give a short overview of CPU trends and show how they impact algorithm efficiency, and sketch how our MonetDB/X100 database system is designed to extract maximum performance from modern CPUs. Section 3 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 4, as well as on information retrieval datasets from TREC and INEX in Section 5. We conclude and outline future work in Section 6.

2 Background

2.1 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 [18, 8, 7]:

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" [4], 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 `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 compression; especially node pointer prefix compression in B-trees is quite prevalent (e.g. in DB2). Teradata's Multi-Valued Compression [10] 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 *exception value*, and are stored elsewhere in the tuple. Oracle also uses dictionary compression, but on the granularity of the disk block [11]. 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 MonetDB/X100 system strongly resembles the Sybase IQ product [14]. 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 [19]. LZRW1 is a fast version of common LZW [17] Lempel-Ziv compression, that uses a hash table *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 3 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 1).

An interesting research direction is to adaptively determine the data compression strategy during query optimization [5, 8, 18]. 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 [14] (called "enumeration views" in [4]). Optimization strategies for compressed data are described in [5], 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 I/O has received significant attention in the information retrieval community, in particular for compressing inverted lists [20]. 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 [20]. 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 [9]). More recently, attention has been paid to schemes that trade compression ratio for higher decompression speed [16]. In Section 5, 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" [2].

2.2 Super-Scalar CPUs

In a *pipelined* CPU, execution of an instruction is split into several subsequent stages. Each clock cycle, a new instruction is taken into execution, and all instructions in the pipeline advance one stage. An instruction leaves the pipeline upon completion of its final stage. By dividing the pipeline into more stages, each stage has to do less work, thus terminates quicker, allowing for a higher CPU clock rate. While the 1988 Intel 80386 CPU executed one instruction in one (or more) cycles, the 1993 Pentium already had a 5-stage pipeline, to be increased to 14 in the 1999 PentiumIII and to 31 in the 2004 Pentium4. This architectural trend has helped increase the CPU frequency significantly above the increase brought by smaller transistor sizes in the last decade.

Super-scalar CPUs add the possibility to issue more then one instruction per cycle. As long as these instructions are independent, each of them is dispatched into one of several parallel pipelines. Therefore, a super-scalar CPU can achieve an IPC (Instructions Per Cycle) higher than 1.

Pipeline throughput suffers from two dangers ("hazards" in VLSI design terminology): a *(i) data hazard* happens if one instruction needs the result of a previous instruction. The second instruction cannot start executing right after the first, but must wait until the first instruction has produced its result. Often, CPUs reduce this amount of wait cycles with *store-and-forward* techniques, that use local data storage (called "reservation stations") located *inside* the execution unit to pass results directly to another waiting instruction (without e.g. going through the registers). Therefore, *(ii) control hazards* are the more costly form of instruction dependencies. They happen when an `if-then-else` or a dereferenced function call (e.g. late method binding) occurs in a program. As these modify the program counter that determines which instruction is next, the CPU does not know which instruction to fetch next, such that it stalls fully.

To avoid control hazards introduced by `if-A-then-B-else-C` constructs, modern CPUs employ *branch prediction* techniques. The CPU tries to predict the outcome of `A`, based on past branching behavior, and starts executing the predicted path. Many stages further, when the evaluation of `A` finishes, it may determine that it guessed wrongly (i.e. *mispredicted* the branch), and then it must *flush* the pipeline (discard all instructions in it) and start over with `B`. Obviously, the longer the pipeline, the more instructions are flushed away and the higher the performance penalty. Translated to database systems, branches that depend on table values, such as those found in a selection operator on data with a selectivity that is neither very high nor very low, are impossible to predict and can significantly slow down query execution [12]. The Intel Itanium2 CPU has a feature called branch *predication* for eliminating branch mispredictions, by allowing to execute *both* the THEN and ELSE blocks in parallel and discard one of the results as soon as the result of the condition becomes known. It is also the task of the compiler to detect opportunities for branch predication.

2.3 MonetDB/X100 Architecture

Our MonetDB/X100 engine is being designed to extract high IPC from super-scalar CPUs. Its *vectorized* query engine is a Volcano-like operator pipeline where each relational algebra operator implements a standard `open-next-close` interface. Contrary to other implementations, the `next()` method yields not one new tuple, but a *vector* – an array typically consisting of a few hundreds of values. Consequently, the *primitive functions*, called by the relational operators to perform computations, are tight loops that perform very simple operations (e.g. addition) on arrays of values, producing arrays of results. The benefits are: *(i)* the compiler can use *loop-pipelining*, and *(ii)* function call cost (in general around 20 cycles) is only paid once per vector.

This design allows primitive operations in MonetDB/X100 to take between 3-10 cycles per value; where in standard relational engines these take 50-100 cycles. It was shown that in a read-only main-memory scenario, MonetDB/X100 achieves impressive TPC-H scores [3]. Our work on the new ColumnBM buffer manager and super-scalar compression methods aims to remove both the "read-only" and "main-memory" restrictions. In fact, the TPC-H 100GB experiments on a 4GB RAM system presented in Section 4 already substantiate the latter claim: all queries run CPU-bound and with high efficiency.

Our approach to super-scalar data (de)compression is similar to that of our CPU-efficient arithmetic primitives, 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.

Work in MonetDB/X100 is ongoing to support efficient updates using differential files [13]. The idea is to store modifications in (in-memory) delta structures, and to treat the tables on disk 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 1, ColumnBM stores disk pages in 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 chunks (see 3.1.1) need to be re-compressed only periodically.

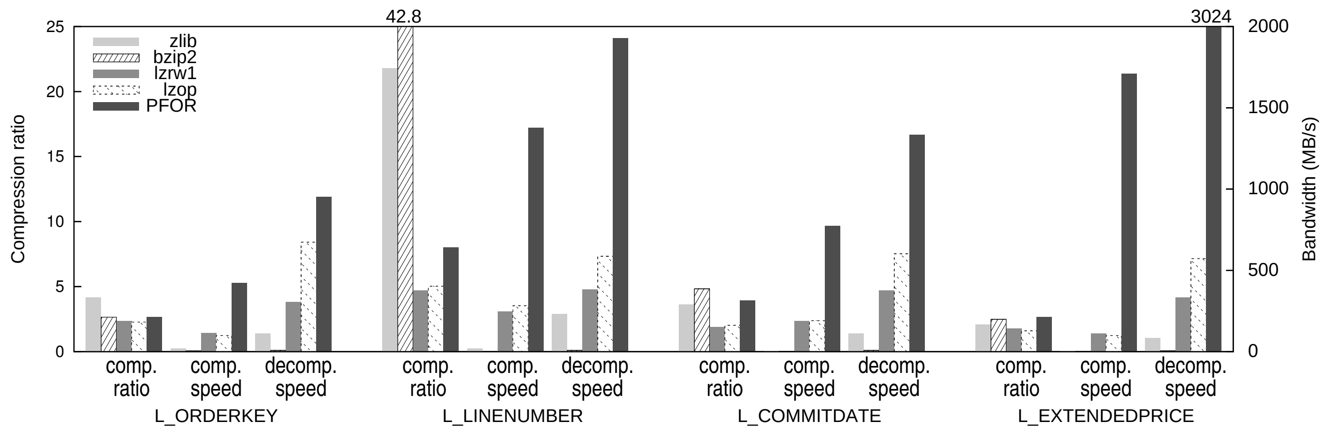

Figure 2. Comparison of various compression algorithms on a subset of TPC-H columns.

3 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 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	= I/O bandwidth
r	= compression ratio
Q	= query bandwidth
C	= decompression bandwidth
R	= result tuple bandwidth

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

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

Many datasets in e.g. data warehouses and information retrieval systems can be compressed considerably [8, 7]. Section 4 shows that even the synthetic TPC-H dataset, with its uniform distributions, allows for a good compression ratio. With these ratios, we often have $B < Q < Br$, such that the query becomes CPU bound using compression. Also, modern RAID systems deliver $B > 0.3$GB/s, so with $r = 4$ one needs $C = 1.2$GB/s just to keep up with that. As we desire to spend only a minority of CPU time on decompression, we need $C = 2.4$GB/s to keep overhead to 50% of CPU time, and $C = 6$GB/s to get it down to 20%. These rules of thumb motivate our design goal of $C = 2 - 6$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. $C = 3$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.

3.1 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 deltas: it encodes the *differences* between subsequent values in the column. Decompression consists of PFOR-decompression, 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.

3.1.1 Disk Storage

The minimum physical granularity for data storage on disk in ColumnBM is the *chunk*, which is a multiple of the filesystem disk block size, chosen such that sequential throughput on single-chunk requests approaches the disk bandwidth (depending on the hardware, values tend to range between 1MB and 8MB). Chunks contain one or more *segments*. In case of column-wise storage, a segment is identical to a chunk. In case of PAX [1], a chunk contains a segment for each column, and all segments in the chunk contain the same number of values, which implies that these segments may have different byte-sizes (that sum to a number close to the chunk size).

Uncompressed fixed-width data types are stored in a segment as a simple array of values.[1] Figure 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.

[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.

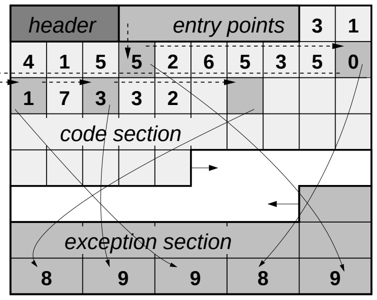

Figure 3. Compressed Segment Layout (encoding the digits of π: 31415*9*26535*89*7*9*32 using 3-bit PFOR compression).

- the *exception section*, growing backwards, stores non-compressed values that could not be encoded into a small integer code.

3.1.2 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 (resp. bit-packing is post-processing for compression). It is done with highly optimized routines that are *loop-unrolled* to handle 32 values each iteration. We found this (un)packing to take up only a moderate fraction of our (de)compression cost, so we omit these details in our code.

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:

```
/* 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)`, that transforms an input value into a small integer, and *(ii)* `ANY DECODE(int)`, that 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 lower-left part of Figure 4 demonstrates most clearly on Pentium4 how

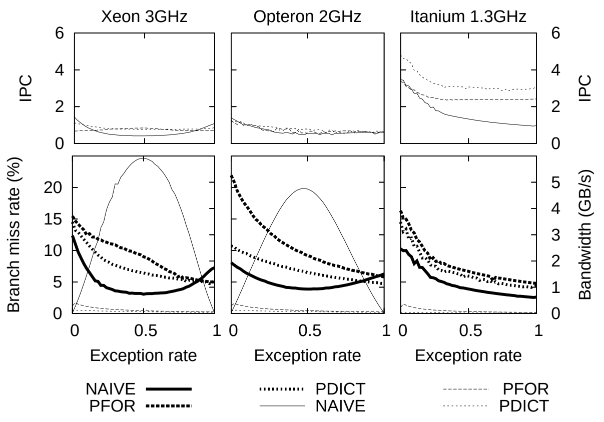

Figure 4. Decompression bandwidth (thick lines) and branch miss rate (thin lines) as a function of the exception rate.

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 (Instructions Per Cycle) 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:

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

  UNPACK[b](code, input, n); /* bit-unpack the values */

  /* LOOP1: decode regardless */
  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, that all can be loop-pipelined by a compiler.

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

[2]We collected IPC, cache misses, and branch misprediction statistics using *CPU event counters* on all test platforms.

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. That makes its overhead easily amortized by the performance improvement of the first loop.

The results in Figure 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.

On Itanium2, the branch mispredictions are avoided thanks to branch predication explained in Section 2.2. As a result, the performance of the NAIVE kernel closely tracks that of PFOR and PDICT, as presented in the rightmost graph in Figure 4. Overall, the patching schemes are clearly to be preferred over the NAIVE approach, as they are faster on all tested architectures.

3.1.3 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 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

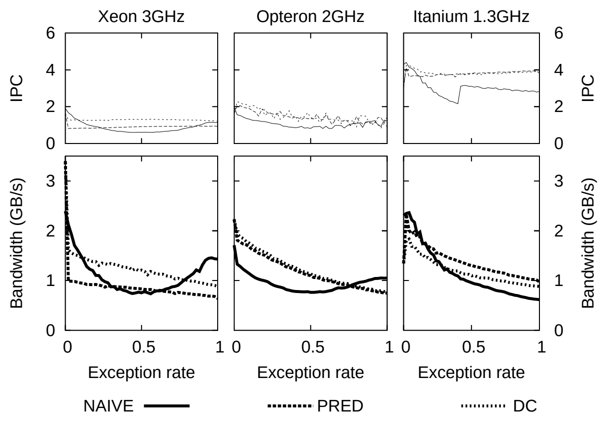

Figure 5. PFOR compression bandwidth as a function of exception rate, using an `if-then-else` (NAIVE), predication (PRED) and double-cursor predication (DC).

also important for re-compression of data chunks occurring in case of updates, as described in Section 2.3. Note that I/O write bandwidth tends to be considerably lower than read bandwidth. Therefore, the design goal of compression throughput can be lower than for decompression, e.g. 1-2GB/s. The bottom graphs in Figure 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.

```
int Compress<ANY>( int n, int b,
  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[-i] = input[cur];
    data[prev] = (cur - prev) - 1;
    prev = cur;
  }
  PACK[b](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 called *predication* [12]: the current position is always copied to the end of the list, and the list pointer is incremented with a boolean.

Predication 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 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):

```
/* 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 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.

3.1.4 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 $32/128 = 0.25$ bits per value. Note

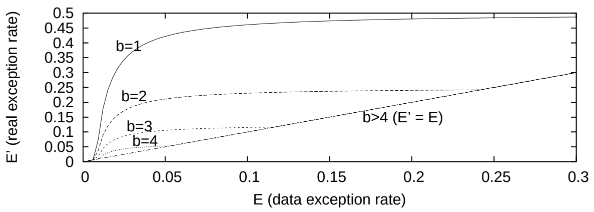

Figure 6. How compulsory exceptions increase the real exception rate E' for $b \leq 4$.

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:

```
ANY finegrained_decompress(int x,
  int*__restrict__ code,
  ANY*__restrict__ exception,
  entry_t*__restrict__ entry,
) {
  int i = entry[x>>7].start_list + x & ~127;
  int j = entry[x>>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 ($< 128 * 0.3/2 = 21$) 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.

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

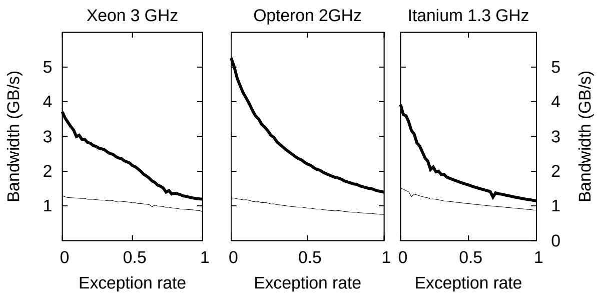

Figure 7. RAM-RAM (thin) versus RAM- Cache PFOR decompression (thick).

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 $E' = MAX(E, \frac{128E-1}{128E}2^{-b})$. Figure 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.

3.1.6 RAM-RAM vs. RAM-Cache Decompression

Figure 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

TPC-H query	compression ratio		Opteron 2GHz 4-disk RAID, 4GB RAM					Pentium4 Xeon 3GHz 12-disk RAID, 4GB RAM						8 x P4 Xeon 2.8GHz 142 disks, 16GB
	DSM	PAX	dec.speed MB/sec	DSM		PAX		dec.speed MB/sec	DSM			PAX		IBM DB2 UDB 8.1
				unc.	$M \Rightarrow C$	unc.	$M \Rightarrow C$		unc.	$M \Rightarrow C$	$M \Rightarrow M$	unc.	$M \Rightarrow C$	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
03	3.04	1.66	2546	35.0	11.3	183.5	113.6	2306	8.9	6.0	7.1	45.5	27.0	15.1
04	8.15	1.82	3018	18.2	2.4	115.5	65.9	3709	4.8	1.8	2.3	30.2	16.5	12.5
05	3.81	2.24	2119	54.3	15.3	300.1	155.9	2421	17.2	16.2	16.7	81.2	36.7	84.0
06	4.39	2.25	2031	48.2	10.7	232.7	104.3	2200	10.8	4.6	6.1	51.0	22.5	17.1
07	1.71	2.01	1251	119.8	72.0	614.2	349.4	1457	34.4	40.8	48.3	158.0	76.5	86.5
11	2.14	1.08	3225	27.0	14.6	180.9	162.2	4084	18.8	18.5	19.4	38.8	35.6	19.5
14	1.91	1.94	2888	23.7	12.2	90.6	46.9	3688	5.8	4.9	5.4	22.1	11.5	10.9
15	2.70	2.13	2464	44.9	22.4	209.8	97.1	2584	30.3	31.2	31.3	49.6	40.0	21.6
18	3.56	2.75	3833	181.9	50.6	1379.7	704.9	4315	38.9	13.6	21.3	419.9	151.5	318.2
21	4.11	2.12	2520	197.6	46.6	1423.5	759.2	2600	43.2	24.2	32.1	338.0	157.6	374.9

Legend: *unc.* – uncompressed data, $M \Rightarrow C$ – memory-to-cache decompression, $M \Rightarrow M$ – memory-to-memory decompression

Table 2. TPC-H SF-100 experiments on MonetDB/X100 (except DB2 results, taken from `www.tpc.org`)

in the RAM-RAM results from Figure 7.

3.1.7 Choosing Compression Schemes

The table materialization operator in MonetDB/X100 should automatically decide which compression method to use for each disk chunk, and with what parameters. The idea is to first gather a sample (e.g. s=64K 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 chunk. When it strongly deteriorates, we could re-run the compression mode analysis to adapt the parameters for the next chunk or even choose another compression scheme. The complexity 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);
}
```

[3]In the near future, we plan to add new super-scalar compression algorithms targeted at floating point data and text.

We simply invoke this function for all relevant bit-widths b and choose the setting that yields best compression, i.e. $1 \leq b < 8 * \texttt{sizeof}(\texttt{V})$ where $b + E_{PFOR(b)} * 8 * \texttt{sizeof}(\texttt{V})$ 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* that is used during PDICT compression to compute the integer codes for values that must be compressed. Discussion of the particular hashing technique used is left out of scope here due to space limitations. In all, it achieves PDICT compression bandwidth of $> 1GB/s$ on all our three test platforms.

4 TPC-H experiments

Table 2 shows the performance of our compression algorithms in MonetDB/X100 running the TPC-H benchmark [15] 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.

	PFOR-DELTA			carryover-12			shuff		
	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s
INEX	1.75	679	3053	2.12	49	524	2.45	3.5	82
TREC fbis	3.47	788	3911	4.26	98	740	5.11	190	164
TREC fr94	3.12	682	3196	3.49	84	689	4.65	149	154
TREC ft	3.13	761	3443	3.47	84	704	4.89	178	157
TREC latimes	2.99	742	3289	3.30	79	683	4.61	164	153

Table 3. PFOR-DELTA on Inverted Files.

We used the same data clustering and index structures as in the previous in-memory MonetDB/X100 TPC-H SF-100 experiments [3]. 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 [6], we also present the results for PAX storage [1]. 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. We urge not to draw any further conclusions from these numbers other than that the MonetDB/X100 results are in the high-performance ballpark.

The TPC-H data was compressed using PFOR, PFOR-DELTA and PDICT (enum) compression schemes. The second and third columns of Table 2 show the compression ratios achieved per query. Note, that since "comment" fields could not be compressed with our algorithms, the PAX queries achieve significantly lower compression ratios. Columns 4 and 9 show that in most cases we reach our decompression speed target of $>$ 2 GB/s.

On the Opteron system, the speedup for most of the DSM queries is in line with the compression ratio. As the left part of Figure 8 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 8 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 by 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. Column 12 of Table 2 shows that such decompression from memory into memory is significantly slower than the fine-grained decompression between RAM and the CPU Cache, presented in column 11. 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 8, 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.

5 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 [20]. In this area, there is a trend to use lightweight-compression schemes rather than the classical storage-optimal schemes [16, 2].

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 [2], 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 3 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 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 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 3 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.

6 Conclusions and future work

In this paper, we 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

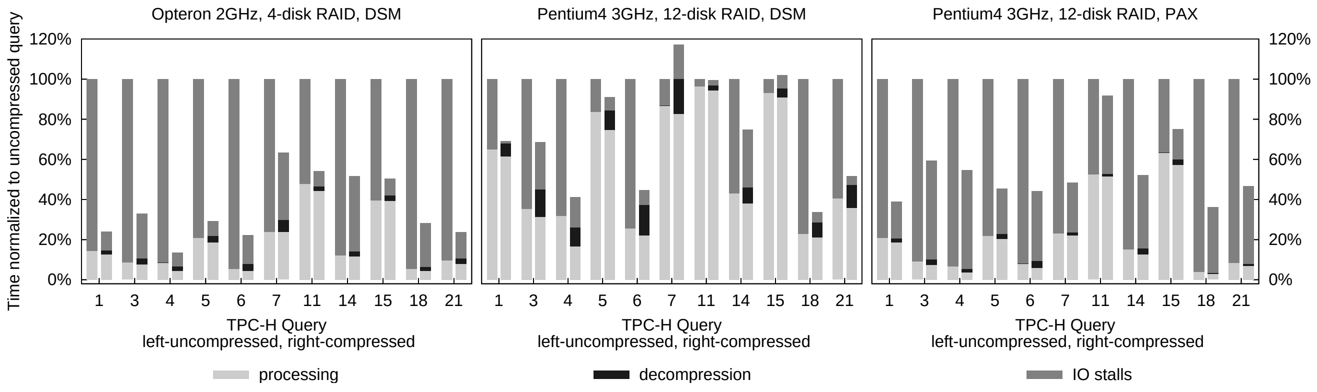

Figure 8. TPC-H SF-100 results, split in decompression time, other CPU time, and I/O time.

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. These extensions will be driven by our intent to use the MonetDB/X100 system to analyze huge repositories of multimedia and scientific (e.g. astronomical) data.

Finally, we believe that with the upcoming families of multi-core CPUs, the demand for high-performance data delivery will rapidly increase. With this increasing (parallel) CPU processing power, highly data-intensive applications might suffer not only from disk but also from a main-memory bandwidth bottleneck. Preliminary results show that our high-performance decompression routines can already improve this bandwidth on parallel architectures.

References

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, Rome, Italy, 2001.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[4] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[5] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. *SIGMOD Rec.*, 30(2):271–282, 2001.

[6] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, pages 268–279, Austin, TX, USA, May 1985.

[7] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. ICDE*, Orlando, FL, USA, 1998.

[8] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, 1991.

[9] D. Huffman. A method for construction of minimum redundancy codes. volume 40, pages 1098–1101, 1952.

[10] NCR Corporation. Teradata Multi-Value Compression V2R5.0. 2002.

[11] M. Pöss and D. Potapov. Data Compression in Oracle. In *Proc. VLDB*, Berlin, Germany, 2003.

[12] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[13] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.

[14] Sybase Inc. *Sybase IQ*. `http://www.sybase.com`.

[15] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002. `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.

[16] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[17] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[18] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, September 2000.

[19] R. N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

[20] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.