Decomposing and Re-Composing Lightweight Compression Schemes – And Why It Matters

Conference Paper · April 2018

DOI: 10.1109/ICDE.2018.00207

1 author:

Eyal Rozenberg
Centrum Wiskunde & Informatica
12 PUBLICATIONS **34** CITATIONS

SEE PROFILE

Decomposing and re-composing lightweight compression schemes – and why it matters

Eyal Rozenberg [#1]

[#] *DB Architectures Group, CWI*
Science Park 123, Amsterdam 1091 CD, The Netherlands
[1] E.Rozenberg@cwi.nl

***Abstract*—We argue for a richer view of the space of lightweight compression schemes for columnar DBMSes: We demonstrate how even simple schemes used in DBMSes decompose into constituent schemes through a columnar perspective on their decompression. With our concrete examples, we touch briefly on what follows from these and other decompositions: Composition of alternative compression schemes as well as other practical and analytical implications.**

I. Introduction

Schema data compression features in most leading analytic DBMSes (among others: [1], [2], [3]). Compression allows for a larger working set (if not the entire DB) to reside in RAM rather in secondary storage. It also helps counter the increasing imbalance between the available processing power and the bandwidth at which data reaches the processor: Decompression requires more computational effort per element of data, but increases the rate at which these elements are effectively received. This second aspect of the utility of compression also limits the "strength" of compression schemes in use: Overly-demanding decompression would slow down the speed of processing data below what the incoming bandwidth allows. As maximum compression ratios are not the goal, the compression schemes in frequent use are "light-weight": Discarding redundant bits (NS; storing the differences from a reference value (FOR); storing the difference between elements rather than the actual values (DELTA); encoding runs of identical values with their respective length (RLE); using small dictionaries (DICT) and so on.

These compression schemes may obviously be *composed* to useful effect. For example: A table holds shipped order details, with a date column. Data accrues over time, so the dates forms a monotone-increasing sequence with long runs for the orders shipped every day. Applying an RLE scheme to the dates, then applying DELTA to the run values,achieves a much stronger compression ratio than any single scheme individually.

What is less obvious is the potential for *decomposing* such simple schemes. It is not immediately clear how this is even feasible, or has any meaning. The following examples should do away with such a perception.

II. Decomposition examples

A. Decomposing Run-Length Encoding (RLE)

The RLE compression scheme is intended for columns with long sequences of identical elements; each such "run" is replaced with a pair: The run's length in elements, and the uniform value of these elements. In columnar terms, a single column `col` of values is compressed into a pair of corresponding columns, `lengths` and `values`, whose length is the number of runs in `col`.

This columnar view of the compressed form, stripped bare of implementation-specific "adornments" we often see in practice (fixed-length blocks, headers, cache-friendly padding and so on), lends itself to manipulation by straightforward columnar operations (see, e.g. [4], [5], [6]). In fact, just very few of these are already enough to express a *decompression algorithm* for RLE: Algorithm 1.

Algorithm 1 RLE decompression

Input: Equal-length columns `lengths`, `values`

1: `run_positions` ← PrefixSum(`lengths`)
2: n ← `run_positions`$[|$`run_positions`$|-1]$
3: `run_positions'` ← PopBack(`run_positions`)
4: `ones` ← Constant$(1, |$`run_positions'`$|)$
5: `zeros` ← Constant$(1, n)$
6: `pos_delta` ← Scatter(`ones`, `run_positions'`)
7: `positions` ← PrefixSum(`pos_delta`)
8: **return** Gather(`values`, `positions`)

Now suppose that rather than a `length` column, we were instead to hold `run_positions`. Clearly, We could reproduce the uncompressed column by applying Algorithm 1, sans its first operation; and no ambiguity is introduced. This constitutes a different compression scheme: *Run Position Encoding*, or RPE for short (see [7, §7.2]).

The transition from RLE to RPE requires the integration of the run lengths — the differences between consecutive values of `run_positions`. If we ignore the `values` column for a moment, we are left with what is simply the decompression of a DELTA-compressed form of `run_positions`. If we denote by ID the "compression scheme" of not applying any compression, we have:

$$\text{RLE} \equiv (\underbrace{\text{ID}}_{\text{for values}}, \underbrace{\text{DELTA}}_{\text{for run\_positions}}) \circ \text{RPE}$$

Lessons learned 1:

- Decompression can often be implemented using the same columnar operations which show up in query execution plans, on uncompressed data. Thus,

- There is no clear distinction between decompression and analytic query execution.
- Partial decompression of the compressed form of one scheme — when considered in terms of operations on columns — often itself corresponds to another compression scheme, which trades away some of the potential compression ratio of the composite scheme for ease of decompression.

B. Decomposing Frame-of-Reference (FOR)

The FOR compression scheme relies on compressed data having limited local variation despite of potentially larger global variation. It is also geared towards the practice of breaking up compressed columns into segments, chunks or blocks. For every such segment, a baseline or frame-of-reference value is provided, and the rest of the compressed form are offsets relative to this FOR. Due to the locality feature, the offsets can be narrower than would be necessary for representing an arbitrary value in the column (or even an offset from a global frame-of-reference). Note that it need not necessarily be the case that the first column element in the segment is also the frame-of-reference value.

Again, let us conceive of the compressed form as "pure" columns: For an uncompressed column `c` of length n, the FOR-compressed form with segment length ℓ would be the value ℓ, a `refs` column of length $\lceil n/\ell \rceil$, and an `offsets` column of length n, in which elements $i \cdot \ell \ldots (i+1) \cdot \ell - 1$ are the offsets for compressed segment i, corresponding to $\texttt{refs}[i]$. Thus, again, the columnar representation allows for a columnar decompression of FOR: Algorithm 2.

Algorithm 2 FOR decompression

Input: Columns `refs`, `offsets`, segment length ℓ

1: `ones` $\leftarrow$ Constant$(1, |\texttt{offsets}|)$
2: `id` $\leftarrow$ PrefixSum$(|\texttt{ones}|)$
3: `ells` $\leftarrow$ Constant$(\ell, |\texttt{offsets}|)$
4: `ref_indices` $\leftarrow$ Elementwise$(\div, \texttt{id}, \texttt{ells})$
5: `replicated` $\leftarrow$ Gather$(\texttt{refs}, \texttt{ref_indices})$
6: **return** Elementwise$(+, \texttt{replicated}, \texttt{offsets})$

When decomposing RLE, we focused on the last steps of the decompression algorithm; here, let us do the opposite — keep the initial steps, and ignore the addition. In this case it, is as though all offsets are 0, and the decompressed data is a step function — having the constant value $\texttt{refs}[i]$ on the entire i^{th} segment. A compression scheme of fixed-segment-length step functions is not very useful as a stand-alone scheme for use within a DBMS, as it captures a tiny fragment of potential columns — but it is quite useful conceptually, allowing for the following formulation of the FOR scheme:

$$\text{FOR} \equiv (\textsc{StepFunction} + \text{NS})$$

since the offsets are, indeed, nothing but a narrow column, which relative to the original column's width we compress with NS. In other words, FOR captures *all columns which are L^∞-metric-close to the evaluation of a step function* (with the distance determined by the allowed width of the `offsets` column, i.e. the width parameter of the NS scheme).

The rough correspondce of the column data to a simple (i.e. much-less-than-n-dimensional) model can be used to speed up selections (e.g. range queries) and joins, or in the context of approximate or gradual-refinement query processing. One is also tempted to enrich the space of low-dimensional models, or to replace the L^∞ metric with another one:

- For the L^0 metric, i.e. $d(\vec{x}, \vec{y}) = |\{i < n \mid x_i \neq y_i\}|$, we could add *patches* to the basic model (see [1], [8]); this would represent columns whose data is "really" a step function, but with the occasional divergent arbitrary-value element.
- Let $d(x, y) = \lceil \log_2 |x - y| + 1 \rceil$ for $x \neq y$, i.e. the number of bits necessary for representing $|x - y|$. For the product metric $d(\vec{x}, \vec{y}) = \sum_i d(x_i, y_i)$, we could use a variable-width encoding for the `offsets` column (ignoring the encoding of offset widths for simplicity of presentation).

As for enriching the model, It is appealing to consider piecewise-linear functions, i.e. keep an offset from a diagonal line at some slope rather than the offset from a horizontal "step"; more generally, we would replace step functions with stepwise low-degree polynomials, or splines. Of course, this makes compression more of a challenge, as it would now require non-linear curve fitting rather than taking the minimum or the middle of the range of values.

Lessons learned 2:

- Some compression schemes separate a simpler, coarser, inaccurate representation of the data from finer, local, noise-like complementary features.
- Compression schemes of algebraic element types often have algebraic aspects themselves.
- The algebraic aspects of (composite) compression schemes underscore an algebraic view of columns as tuples, or evaluated functions.
- Generalizing/refining a compression scheme often means generalizing/refining one or more of its subschemes.

References

[1] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM–CPU cache compression," in *Proc. ICDE*. IEEE, 2006, pp. 59–59.

[2] D. J. Abadi, S. R. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proc. SIGMOD*. ACM, 2006, pp. 671–682.

[3] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper, "Data blocks: Hybrid OLTP & OLAP on compressed storage using both vectorization and compilation," in *Proc. SIGMOD*, 2016, pp. 311–326.

[4] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proc. VLDB*, vol. 3, no. 1-2, pp. 670–680, 2010.

[5] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *Trans. DB Sys.*, vol. 34, no. 4, pp. 21:1–21:39, Dec. 2009.

[6] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo — a vector algebra for portable database performance on modern hardware," *Proc. VLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.

[7] H. Plattner, *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer, 2014.

[8] E. Rozenberg and P. A. Boncz, "Faster across the PCIe bus: a GPU library for lightweight decompression," in *Proc. DaMoN*, 2017, pp. 8:1–8:5.