A computational model for analytic column stores
with focus on columnar representations of data and composite compression schemes

Eyal Rozenberg¹

Abstract
This work presents an abstract model for the computations performed by analytic column stores or columnar query processors. The model is based on circuits whose wires carry columns rather than scalar values, and whose nodes apply operators with column inputs and outputs. This model allows expression of most of the architectural features of existing column-store DBMSes through columnar execution plans, rather than such features being implemented sui-generis, and without the column store maintaining significant out-of-plan data. A strict adherence to columnarity allows for a relatively simple and robust model; enabling extensive and intensive optimization of almost all aspects of query processing; and also enabling massive uniform parallelization of query process on modern hardware. Moreover, the computational model’s expressivity makes it useful also as an analytical tool for considering design aspects and features of existing column stores, individually and comparatively.

To achieve the model’s wide expressiveness, much of this work develops representation schemes of relevant data structures as combinations of plain columns, with columnar circuits used as scheme encoders and decoders. A particular focus is given to schemes which also compress the data, and their use in query execution — as an integral part of the computation: Subcircuits of larger columnar circuits, not black boxes. Decoder and encoder circuits are thus also composed to form more elaborate schemes. Such formulation allows both for an alternative view of well-known compression schemes, and for the development of new columnar compression schemes with useful features; these should be of independent interest irrespective of column store systems.

¹Bikurim Street 11/2, Haifa 3457609, Israel/Palestine. E-mail: eyalroz@technion.ac.il. Work done in part while at the DB architectures group at CWI, Science Park 123, Amsterdam 1098 XG, The Netherlands.
# Table of contents

Introduction 3

1 A model for columnar computation 7
   1.1 Model building blocks .................................................. 8
      1.1.1 Columns .............................................................. 8
      1.1.2 Circuit nodes: Computational operators .......................... 8
      1.1.3 Circuit layout: Port graphs ...................................... 11
   1.2 Columnar circuits ..................................................... 13
   1.3 Circuit composition and transformation ............................. 17

2 Columnar representation 21
   2.1 Encoding/decoding schemes for representation ...................... 21
   2.2 Subcolumns .............................................................. 23
      2.2.1 Explicitly-indexed columns ...................................... 23
      2.2.2 A subcolumn representation ..................................... 24
      2.2.3 Combining subcolumns ........................................... 25
   2.3 Segmented columns .................................................... 28
      2.3.1 Uniform segmentation and nearly-matrices ........................ 30
      2.3.2 Data replication within and across segments ................... 31
      2.3.3 Adapting operators to column segmentation .................... 33
   2.4 Representing additional constructs .................................. 34
      2.4.1 Index subsets ....................................................... 34
      2.4.2 Partitions of columns and index sets ............................ 36
      2.4.3 Element decomposition, interleaving, and type-punning ....... 37
      2.4.4 Variable-width columns and string representation ............ 40
      2.4.5 Nullable and union-type columns ................................. 41

3 Column compression 43
   3.1 Basic schemes .......................................................... 43
      3.1.1 Output-index-based compression ................................. 44
      3.1.2 Column support compaction ..................................... 45
   3.2 Patterns of scheme composition and transformation ................ 47
      3.2.1 Elementwise composition and “noise-signal”-like separation ... 48
      3.2.2 Discrete differentiation and integration ...................... 49
      3.2.3 Segmentization .................................................... 49
Introduction

Why bother spending time on formalizing a computational model for columnar data processing systems? Column stores have, after all, already become well-established and widely-used. Isn’t this pursuit several decades too late, and thus mostly irrelevant?

It is the author’s belief that nothing could be further from the truth, and that the dearth of theoretical underpinnings is an obstacle to the progress of more practical research into column stores or columnar DBMSes and their further acceleration. The reason for this, in a nutshell, is that column stores designers have failed to realize and exploit the full potential of the column as a fundamental concept and focus of computation. A formal computational model would:

- make the plethora of architectural features column stores typically support [ABH+13, §1] be taken care of as integral parts of the execution of a plan, without special treatment by feature-specific code. Consequently,

- bring as much of a column store’s computational work when processing a query into the sphere of consideration and influence by execution plan optimization phases.

- facilitate the conception and specification of a rich space of composite compression schemes, fully and seamlessly integrated into execution plans.

- form a basis for a further theoretical endeavor: A grammatic formalism for execution plan transformation/optimization rules — rather than such rules constituting mostly ad-hoc unconstrained code.

- inspire a more streamlined, flexible architecture for future column store software systems, especially those focused on emerging parallel-execution hardware, lending itself to better utilization of their capabilities as well as easier extensibility.

The non-history of column store computational models

It is somewhat peculiar that research into column stores has not produced any formalized computational models long ago already. An (essentially) columnar storage model for DBMSes was proposed as such already in 1985, in the form of the Decomposition Storage Model (DSM) [CK85]; columnar file storage format (“transposed files”) was suggested even earlier, in [Bat79]; and work in which a columnar model is implicit goes as far back as half a century ago, to the late 1960s [EB69]. Columnar DBMSes as full-fledged products have also long been available: KDB and Sybase IQ [Fre97] were released in the 1990s; MonetDB [Bon02] and C-Store [SAB+05] were made available in the 2000s as free software — the former is in active development under a free license even today.
In fact, by the time of writing, all major commercial DBMSes which consider analytics to be a use-case sport columnarity in some form or another — even if only as an auxiliary index for a row store: See [ABH+13, §2.2, §3.3] for the state of affairs as of 2013, which has since progressed further with systems by Amazon [GAT+15], Microsoft [LBH+15], Oracle [LCC+15] and so on. But these advances in practice have not manifested themselves in theoretical research; that is still wholly dominated by efforts inspired by, and geared towards, row-oriented DBMSes, unordered relations, and/or transactional work. The author was unable to find a single paper evaluating the computational models — formal or informal — used by the prominent column stores of recent years.

The more abstract fields of the theory of computation generally, and parallel computation theory particularly, also offer no panacea in terms of a relevant model. The closest computational models formalized and studied in the literature (see, e.g., [Sav98, Chapter II] for a textbook overview and [KSS18] for a recent and more domain-specific publication) seem to fall into one of the following categories:

1. A network of many computing nodes, acting independently and communicating with each other, where the parallelism is expressed by multiple nodes working on potentially different elements of a column/array).
2. A data-parallel computer, supporting complex control flows, exhibiting either SIMD (Single-instruction multiple-data) or SPMD (Single-program, multiple-data) behavior.
3. Circuits / circuit families, carrying scalar values on the wires, using a small fixed set of very-simple operator nodes (boolean algebra, or arithmetic operations on rational numbers). Parallelism may expressed by multiple nodes processing multiple elements of the input (in our case, the column).

The network abstraction of the first category is not useful for us to adopt: Networking may be a significant part of implementing a system as it scales, but the parallelism does not inherently depend on this network structure, nor is the parallel nature based on it. The second model is too powerful for our use: Allowing loops and such control structures would make it rather non-meaningful to distinguish between individual operators and execution plans more generally. Finally, the circuit model, in itself, does not express the concept of the column, or of the processing of an entire column at once. What we would be after is some kind of middle-ground between the two last options, as we shall elaborate in this document.

Yet another subfield of computer science in which we find some context for our desired computational model is that of programming languages. Several languages have been devised over the years intended to manipulate arrays as fundamental types; and some of these involve “dataflow graphs” in one form or another (e.g. [FCO90]). Unfortunately, the author is not well-versed in the scholarly work in this subfield enough to authoritatively draw on such work; but it would seem previous research efforts have not fleshed out or formalized a proper computational model which one may adapt for use with column stores. Some inspiration can be drawn from the the monadic and diadic operators of the APL language [Ive62], formalized with vector and matrix arguments in mind (as per the relatively late but more comprehensive set of definitions in [Mic]).

Finally, the reader should bear in mind that existing column stores do not actually share a single identical computational model or common set of structural features it relates to. Differences can be quite significant: Proper-DSM-based column stores couple each column of data with a keying column
relating it to the rest of its table; while others operates on tuples in multi-column projections. Or, taking another aspect of the model, some column stores support execution plans with strictly acyclic dependencies between operators — circuit or DAG-like plans; others allow full-fledged imperative programming, with loops and all (even if these are rarely used in practice).

What existing column stores have left out

Typically (if not universally), the columns appearing a column store’s execution plan are those found in a database schema — columns within tables to which users can make queries — and intermediate results of operators within the plan. A wealth of auxiliary data is kept separate from the execution plans: NULL indication; column statistics; index structures; partitioning, sharding or cracking information, and more yet. The column store code involving this auxiliary data is not reflected in the execution plans, but rather hard-coded to apply in certain cases and in certain ways. Also, such data is held in structures of idiosyncratic design; and while they might be well-designed for their specific function, they are not themselves columnar — another reason for their necessitating out-of-plan code.

Whatever is done outside of an execution plan cannot benefit from repeatedly-applied and contextually-applied plan optimizations and transformations; it can also hardly benefit, if at all, from the use of just-in-time compilation. Also, such idiosyncratic use of auxiliary data may impose constraints on the actual plan, such as additional materialization points of intermediate results, or less flexibility in scheduling.

Compression, data layout, and ‘pushed-down’ execution

Recently, the author was working on implementing lightweight decompression on a GPU. True to the principle mentioned above, several well-known schemes — Dictionary, FrameOfReference and so on — were concretized using distinct, fixed-width-type columns in memory. As implementation progressed, it turned out that large parts of the code were used by more than one scheme; and that operators originally written for executing queries on uncompressed data were being used as-is in decompression.

These occurrences in practice naturally induce two theoretical questions:

- Perhaps it would be useful to decompose compression schemes into simpler transformations (which do not, individually, decompress anything), and consider them separately as well as in combination?

- Is there really a distinction between the computational operators we employ during query execution “proper”, and the computational work performed for decompression?

The potential for composition of compression schemes, as opposed to their decomposition is rather obvious, both to a query processing engine designer as we as to the end user. For example, if we have a time series with points at intervals whose lengths only have few possible values, then an effective compression scheme could be: Storing the differences between samples rather than the values (Delta compression), and using less bits for the differences (NullSuppression), or perhaps storing indices into a dictionary of possible interval lengths (Dict).

The decomposition of compression schemes becomes useful mostly in the larger context of executing an actual query, when we can sometimes avoid some of the decompression work, and act
on a partially-decompressed form, or on partial data. Let us take the previous example of time-series data: Suppose we wish to perform some aggregation on a feature of events in the time-series which occur after long intervals. In this case, we would like to be able to skip the reconstitution of timestamps from differences, followed by taking differences again to filter out the irrelevant data. But to do so, we must be able to do away with the Delta decompression of the composite scheme.

Existing column stores supporting compressed data do not support composition or decomposition of compression schemes (e.g. Vectorwise [ZHNB06] and C-Store/Vertica [LMV12]); and the same seems to be true for high-performance row-stores (e.g. HyPerDB [Neu11, LMF16]). Some of them do have a few useful composite schemes baked-in; and some have some scan functionality ‘pushed-down’ to operate on the compressed data as it is being decompressed. But the use of each of these is idiosyncratic; and query-specific optimizations and work-avoidance be applied. Specifically, the two examples given above cannot be realized these systems.

**Structure of this document**  This work progressively presents concepts, constructs, definitions, based on previously-exposited ones; but as far as motivation is concerned, the document’s direction is mostly in reverse. We thus begins with the formulation of a computational model (Chapter 1 with rather limited justification for the choices of its specifics; we continue with an exploration of the expressivity of combinatorial structures within the computational model, as columns (Chapter 2; it is only then that we “apply” these representation schemes, in formulating common and less-common compression schemes in columnar terms (Chapter 3). A final chapter Chapter 4, prospectively sketches out aspects of the possible design of an analytic column store implementing the computational model fully and directly.

**Acknowledgements**

Part of the work on this document was conducted while the author was a post-doctoral researcher at CWI Amsterdam (https://www.cwi.nl), in the Database Architectures group; the author wishes to thank all group members, and the group’s founder, Prof. Emeritus Dr. Martin Kersten, for an enlightening stay. The author particularly wishes to thank Prof. Dr. Peter Boncz, for his guidance and insights during that time. Finally, the author wishes to thank Profs. Drs. Boncz and Urbani, as well as Aris Koning of MonetDB Solutions B.V., for useful discussion and comments on earlier versions of this work.
1

A model for columnar computation

In this chapter we formalize a model of the computations performed by column stores. In more concrete terms, we extend the folklore computational model of \textit{combinatorial boolean logic circuits} to columns.

A boolean logic circuit consists of nodes, or vertices, which are logic gates, such as \texttt{And} or \texttt{Not}; and of directed edges, corresponding to physical wires between logic gate outputs and inputs, each “carrying” a one-bit (scalar) value. The vertices and edges form a DAG (directed acyclic graph). Circuit inputs are conceptualized as wires with only their destination attached to gate input, while outputs are wires with only their source attached to a gate output. The function computed by a circuit is computed inductively, starting from an assignment of a sequence of bits to the inputs, and propagated through the vertices to deeper and deeper internal edges — until the final gate or gates which compute the circuit’s outputs — in correspondence to how setting 0/1 values on the input wire propagates through the physical nodes over the wire, eventually setting the output wire charges. We will not restate any of this formally; the reader may consult a detailed textbook treatment in [KJ10].

In the context of a column store, with most activity being (ideally) uniform over large columns, instead of forming gigantic circuits with single-bit or single-scalar-value edges, we instead have each input, and each edge, carry a \textit{column} with elements of some data type. This is where the similarity between the model we formalize and circuits reaches its end, both in the technical details of the definition and with respect to the richness and expressivity of the model.

\textbf{Notational conventions} In the following, \texttt{Img}(f), \texttt{Dom}(f), \texttt{Rng}(f) denote the image, domain and range, respectively, of a function \( f \). Composition of functions is denoted \( f \circ g \) where \((f \circ g)(x) = f(g(x))\). The natural numbers, \( \mathbb{N} \) are considered to include 0. All combinatorial structures we define are finite unless otherwise stated. Diagrams presenting two-dimensional arrays are column-major unless otherwise stated or indicated by element indices. We shall refer to “types” in the sense of programming language type systems: Types have domains — sets of possible values; they have a representation in physical computer memory; and they have associated operators, which may be applied to values in their domains. We’ll assume a fixed set of allowed types, which includes various integers and floating-point types with representations of fixed sizes in bits. This set of types will specifically include \( \tau_{\text{int}} \), an integer type large enough to represent the indices of the column being considered; as well as \( \tau_{\text{unit}} \), the unit type (with a single possible value); and \( \tau_{\bot} \), is a type with an empty domain (having no values). When we refer to “any type” or to an “arbitrary type” — this is
with respect to the global set of allowed types (which will will otherwise not be concerned with).

1.1 Model building blocks

1.1.1 Columns

We begin with a choice of a column store’s fundamental, most basic, data structure:

**Definition 1.1 (Column).** A *(plain) column* is a function \( c : \{0, \ldots, n - 1\} \to \text{Dom}(\tau) \) for some fixed-representation-size type \( \tau \) and \( n \in \mathbb{N} \). \( n \) is the column’s *length* (denoted \(|c|\)), \( \tau \) is its *element type* and the size of \( \tau \)'s physical representation is the column’s *width*.

A column may be *empty* in case \( n = 0 \), or it may interpreted as a *scalar*, a single value of type \( \tau \), for the case of \( n = 1 \). Both cases are valid, though the latter will be explicitly useful in this work, while the former will be mostly ignored.

Our definition of a column is not entirely concrete: It does not specify a particular physical layout for column data. In our computational model, columns will in fact have multiple possible physical layouts, developed at length in subsequent chapters. Still, as a concrete starting point, we choose a default physical layout in which columns manifest: The *standard representation* of a column is the sequence of its evaluations over its domain, by the natural order of integers, i.e. \((c(0), \ldots, c(n - 1))\). While also defined abstractly, this representation translates immediately into the physical world — a sequence of values in a computer’s memory.

In this work we often conflate columns in the abstract with their standard representation. Also, in light of this physical expression, this work uses the array subscripting syntax of C-like programming languages to denote column elements, e.g. for some \( i \in \mathbb{N} \), \( c[i] \) denotes \( c(i) \), the function \( c \) applied to \( i \).

**Definition 1.2 (Element frequency distribution / probability mass function).** Let \( c \) be a column of element type \( \tau \). The *(element) frequency distribution* or *frequency function* of \( c \) is the function \( \text{freq}_c : \text{Dom}(\tau) \to \mathbb{N} \) counting \( \tau \)-value appearances in \( c \), i.e. \( \text{freq}_c(x) = \frac{1}{|c|} |c^{-1}(x)| \).

Normalizing this function by \(|c|\), we obtain the *probability mass function* for the uniform distribution over the elements of \( c \).

Now, the support of \( \text{freq}_c \) (i.e. \( \text{freq}_c^{-1}(\mathbb{R}^+) \)) is of significance — the set of all \( \tau \) values appearing in \( c \) — while the support of \( c \) as a function is simply \( \{0, \ldots, |c| - 1\} \) regardless of the column’s contents. Thus, abusing notation, we refer to the support of \( \text{freq}_c \) as “the support of column \( c \),” denoting it \( \text{supp}(c) \).

1.1.2 Circuit nodes: Computational operators

In a boolean logic circuit, and with nodes limited to zero, one or two inputs and one output, there can be no more than \( 2^{2^0} + 2^{2^1} + 2^{2^2} \) possible kinds of nodes. In fact, only a small subset of them is necessary to define the computational model, since each of the functions can be computed using a small gadget-circuit using only the small subset: Just *And*, *Or*, and *Not* suffice, or even just *Nand* or *Nor*, alone [KJ10, §3.2]. The situation is entirely different when circuit node inputs are *columns* — the variety of computational operators becomes infinite. We do not, therefore, limit our discussion
to some fixed set of operators, and instead consider arbitrary operators inspecifically. A column store system may limit its operators to a finite set, or support defining them dynamically (as many DBMSes, including column stores, do).

**Definition 1.3 (Signature).** A (single-direction) computational signature is a tuple \((\text{labels}, \text{types})\): A set of labels, and a family of types \(\text{types}(\sigma) = \{\tau_\ell\}_{\ell \in \text{labels}(\sigma)}\) corresponding to each of them. A bidirectional/in-out computational signature is a pair \((\sigma_{\text{in}}, \sigma_{\text{out}})\) of an input and an output single-direction signatures, with disjoint sets of labels.

For an in-out signature \(\sigma\), we denote \(\text{labels}_{\text{in}}(\sigma)\) for \(\text{labels}(\sigma_{\text{in}})\) and \(\text{labels}_{\text{out}}(\sigma)\) for \(\text{labels}(\sigma_{\text{out}})\) and similarly \(\text{types}_{\text{C,in}}(\sigma)\) and \(\text{types}_{\text{C,out}}(\sigma)\). When some entity \(X\) has an associated (in-out) signature \(\sigma_X\), we denote \(\text{labels}_{\text{in}}(X) = \text{labels}_{\text{in}}(\sigma_X)\) and similarly for output labels and for types. Finally, we occasionally assume an implicit order on \(\text{labels}\) sets.

**Definition 1.4 (Operator).** A computational operator is a tuple \((M, \sigma)\) comprising some computational automaton \(M\) with finite description and a corresponding signature \(\sigma\). Omitting details, the automaton — a deterministic Turing machine [Sip12, §3.1] unless otherwise specified — takes as input the standard representations of columns of types \(\text{types}(\sigma_{\text{in}})\), and if it halts, produces the standard representations of columns of types \(\text{types}(\sigma_{\text{out}})\).

We will abuse this definition by referring to operators performing the “same” computation for different input types, as though they were the exact same operator (e.g. elementwise addition and multiplication, defined for multiple possible column element types).

For an operator \(\text{Op}\), we denote by \(\varphi_{\text{Op}}\) the partial function computed by \(\text{Op}\): A partial function from (a subset of) \(\prod_{\ell \in \text{labels}_{\text{in}}(\sigma_{\text{Op}})} \text{Dom}(\tau_\ell)\) to \(\prod_{\ell \in \text{labels}_{\text{out}}(\sigma_{\text{Op}})} \text{Dom}(\tau_\ell)\). We denote by \(\varphi_{\text{Op},\ell}\) the output column of \(\varphi_{\text{Op}}\) with label \(\ell\) (the projection of the output on \(\ell\), if you will). If \(\text{Op}\) has a single output labeled \(\ell\), we may conflate \(\varphi_{\text{Op}}\) with \(\varphi_{\text{Op},\ell}\). The arity of \(\varphi_{\text{Op}}\) is also referred to as the arity of \(\text{Op}\) itself.

As the first few examples of actual computational operators, suppose we wish to have the “same” logic operators of traditional circuits in columnar circuits as well — by having them apply to multiple tuple of inputs, elementwise. Choosing \text{And} as an example, we define the following example operator:

ElementwiseAnd

---

Figure 1.1: Illustration of a computational operator as a circuit node: Elementwise addition of two (equal-length) columns of type \(\tau\). The (grayed-out) arrows indicate how potential graph edges may connect to its ports.
The output satisfies \( \text{result}[i] = \text{lhs}[i] \land \text{rhs}[i] \).

The \( \sigma_{\text{in}} \) and \( \sigma_{\text{out}} \) signatures of this operator are the labels and types on the lines marked “Input” and “Output” above, respectively. As for the length designations, these are actually a constraint on the lengths for which the operator produces an output — as, unlike in boolean circuits, operators are partial rather than complete functions. In this case, \texttt{ElementwiseAnd} will only produce output for pairs of input columns having the same length.

Note. In practice, storing bits often involves issue such as machine byte-alignment. We do not account for alignment issues in this text, except to note that they can be overcome using mechanisms developed in Chapter 2 below.

Our example elementwise operator immediately generalizes to any function \( f : \tau_{\text{in}} \rightarrow \tau_{\text{out}} \) with \( \tau_{\text{in}} = (\tau_1, \ldots, \tau_{k_{\text{in}}}) \) and \( \tau_{\text{out}} = (\tau'_1, \ldots, \tau'_{k_{\text{out}}}) \):

\[
\text{Elementwise}_f
\]

| In/out? | Label      | Type | Length | Description                   |
|---------|------------|------|--------|-------------------------------|
| Input   | arguments1 | \( \tau_1 \) | \( n \) |                               |
| Input   | ...        |      |        |                               |
| Input   | arguments_{k_{\text{in}}} | \( \tau_{k_{\text{in}}} \) | \( n \) |                               |
| Output  | results1  | \( \tau'_1 \) | \( n \) |                               |
| Input   | ...        |      |        |                               |
| Input   | results_{k_{\text{out}}} | \( \tau'_{k_{\text{out}}} \) | \( n \) |                               |

Letting \( x_j = \text{arguments}_j[i] \) and \( y = f(x) \), the output satisfies \( \text{results}_j[i] = y_j \) for all \( j \in \{0, \ldots, k_{\text{out}} - 1\} \).

When \( k_{\text{in}} = 1 \) (or, respectively, \( k_{\text{out}} = 1 \), the single argument (respectively, result) column is referred to as \texttt{arguments} (respectively, \texttt{results}), dropping the index.

This kind of elementwise operators will be put to considerable use throughout this document.

We present one other example of simple operator on columns, with forethought of a columnar computation example later on, in Sections 1.2 and 1.3.

\texttt{Replicate}

| In/out? | Label | Type   | Length | Description               |
|---------|-------|--------|--------|---------------------------|
| Input   | value | \( \tau_{\text{int}} \) | 1      | A scalar value to replicate |
| Input   | factor| \( \tau_{\text{int}} \) | 1      | The number of times to replicate |
| Output  | replicated | \( \tau \) | factor | Copies of \texttt{factor} |

Produces a column with the uniform value \texttt{value} at each index \( \{0, \ldots, \text{factor} - 1\} \).

In the description of this operator we’ve used \texttt{value} and \texttt{factor} as scalars rather than columns; this should be interpreted as a use of \texttt{value}[0] and \texttt{factor}[0] — the single values in these length-1 column.
Side note: On the strength of circuit operators

The choice of Turing machines for Definition 1.4 above is motivated by actual column stores our computational model abstracts from — principally MonetDB [IGN+12], C-Store [SAB+05] and Vectorwise [ZB12]: Most column stores can and do run user-defined functions (UDFs) in one or more high-level language; and these functions may well be Turing-complete. Even if one only considers the operators built-in to the column store — they may involve arbitrary code, and are not restricted apriori in their complexity.

On the other hand, in a typical column store (including the abovementioned ones), UDFs are used sparingly if at all (depending on the application domain); and most of the built-in operators, require no more $O(n)$ time for length-$n$ inputs-plus-outputs, and $O(1)$ space. Even the more involved built-in operators are not terribly complex (e.g. $O(n^2)$ time for a Join in the most unforgiving figuring). Furthermore — and recalling our computational model being formalized with a mind to enable parallelization and distribution of computation — we expect most (not all) operators to fall into an even weaker complexity class: Computations which may be extremely-effectively paralellized to $n$ machines, or cores, or threads. Even $\text{NC}$ [AB09, §6.7.1] would be far too much. Without delving into details, these should have constant parallel running time and be decomposable to very small fixed-size traditional (non-columnar) circuits. This requirement is similar to the “pipelinability” property of operators defined by HyperDB [Neu11], but not quite identical, as may become apparent to a careful of subsequent sections and chapters. Further discussion of this point is beyond the scope of this document.

1.1.3 Circuit layout: Port graphs

Next, a combinatorial structure for our circuits. Boolean logic circuits could make do with a directed (acyclic) graph, as the Or and And binary operators are symmetric, so there is no need to distinguish between their two inputs; and the edge directionality distinguishes inputs from outputs. For operators on columns (and even for asymmetric operators on scalars), inputs are not interchangeable (and neither are outputs). Thus, if we think of them as vertices in a DAG, each such vertex must have different ‘ports’ which edges start from or arrive at. This inspires the following definition.

**Definition 1.5 (Port graph).** A (simple) directed port graph (or (simple) port digraph for short) is a tuple $G = (V, \Lambda, E)$ of vertices $V$, per-vertex port label sets $\Lambda$ and edges $E$, such that:

- $\Lambda$ is a family of per-vertex port label sets: $\Lambda = \{ \Lambda(v) \}_{v \in V}$.

- The edges in $E$ connect ports (i.e vertex-label tuples) rather than vertices. In other words: Denoting the ports of each vertex $v$ by $P_G(v) = \{ v \} \times \Lambda(v)$, and the overall port set of $G$ by $P(G) = P_G(V) = \bigcup_{v \in V} P_G(v)$ — we have $E \subseteq P(G) \times P(G)$.

Non-simple port digraphs and undirected port graphs are defined similarly, mutatis mutandis.

A port $p$ in a port digraph is said to be engaged if it is an endpoint of an edge in $G$ (and disengaged otherwise); if the origin port of its engaging edge is $p'$, $p$ is said to be engaged by $p'$. Two vertices $u, v$ in a port digraph are said to be (directly) connected if the graph has an edge between a port of $u$ and a port of $v$. A port $p'$ is said to be reachable from port $p$ if there is a path of edges beginning at $p$ and ending at $p'$; reachability from a vertex means port reachability from any of its ports; the reachability of a vertex means port reachability of any of its ports.
Vertex set \( V = \{A, B, C\} \)
Port label sets \( \Lambda(A) = \{\text{blue, cyan, red}\}, \Lambda(B) = \{\text{yellow, cyan}\}, \Lambda(C) = \{\text{red, green, blue}\} \)
Edge set \( \{(A, \text{red}), (B, \text{cyan}), (A, \text{cyan}), (C, \text{green}), (C, \text{blue}), (A, \text{blue}), (C, \text{green}), (A, \text{cyan}), (C, \text{green}), (B, \text{yellow})\} \)

Figure 1.2: A 3-node port digraph

Definition 1.6 (Induced sub-port-digraph). Let \( G = (V, \Lambda, E) \) be port digraph with \( V' \subseteq V \). The subgraph of \( G \) induced by \( V' \) is the port graph \( G|_{V'} = (V', \Lambda', E') \) where \( \Lambda' = \Lambda|_{V'} \) and \( E' = E \cap (P_G(V') \times P_G(V')) \).

We denote by \( d_{\text{in}}(p) \) (respectively, \( d_{\text{out}}(p) \)) the input (respectively output) degree of port \( v \) in a port graph \( G \). The input (respectively output) degree \( d_{\text{in}}(v) \) (respectively \( d_{\text{out}}(v) \)) of a vertex \( v \) in a port graph is the sum of all of its ports input (respectively output) degrees. For a pair of port sets \( P_1, P_2 \subseteq P(G) \), \( d(P_1, P_2) \) denotes the number of edges from ports in the first set to ports in the second; and for a pair of vertex sets \( V_1, V_2 \subseteq V_G \), \( d(V_1, V_2) \) denotes \( d(P_G(V_1), P_G(V_2)) \).

Generally, a port graph vertex can have edges coming in and going out via the same port. As the ports are a method of qualifying incoming and outgoing edges, it makes sense to consider those cases in which a given port, a given label, may only be associated with one direction — either incoming or outgoing. More formally, same port in a port graph can have edges coming in and going out

Definition 1.7 (Port orientation). Let \( G \) be a port digraph. A partition \( \Pi = (P_s, P_t) \) of \( G \)'s ports is an orientation (w.r.t. \( G \)) if \( G \)'s edges only connect ports in \( P_s \) to ports in \( P_t \). If \( G \) is port graph for which such a partition exists, it is said to have orientable ports.

In conclusion of this section, bear in mind that the port (di)graph is an abstraction of mere convenience. One could very well ‘encode’ ports as gadgets within a single binary relation, either more trivially with an auxiliary unary relation, or with some gadgetry and higher overhead in graph size, but no auxiliary relations — all using trivial model-theoretical techniques. It would also be possible to have labels as special vertices, and use oriented 4-uniform hyperedges with two regular vertices and two port labels. Our choice is geared towards the definition of circuits, below, since we’ll want to maintain the mental picture of a physical circuit layout, where a graph edge is a
physical wire, a graph vertex is a component placed on a PCB (printed circuit board), and a port is one of the holes into which a component’s pins fit.

1.2 Columnar circuits

We have already indicated columnar circuits are to be similar to boolean logic circuits, except for having columns instead of bits on the wires. Still, the orientation of the edges will necessitate a proper definition, somewhat more rigorous and verbose. Before presenting it, however, let us consider example circuit, Figure 1.3: It counterposes a program-listing-style execution plan (with syntax and structure similar to those used in MonetDB) to a columnar circuit for processing the same query. The program-listing-style plan cannot itself be the equivalent of a circuit, as its instructions/operators are in a total order, while in the circuit there is merely a partial order imposed by the data dependencies. One may thus think of the circuit as being counterposed to the equivalence class of all reorderings of the program-listing-style plan, in which instructions only use variables computed in previous instructions.

A columnar circuit for the query plan, drawn with height determined by dependency depth. The constants are considered to be inputs although they aren’t quite that.

A program in a MonetDB-MAL-like static-single-assignment language, corresponding to the columnar circuit (with element type information).

Figure 1.3: A possible query execution plan for query Q6 of the TPC-H benchmark [Tra17]

Not every port digraph can serve as the underlying structure of a columnar circuit. A port digraph $G$ is said to have circuit layout w.r.t. a partition $\Pi$ of its ports if $G$ is acyclic, $\Pi$ is a port orientation for $G$ (as per Definition 1.7), and no port of $G$ is the target of more than one edge (i.e. $\Delta_{in}(G) \leq 1$). If, given a port digraph, such a partition $\Pi$ exists, $G$ is said to admit circuit layout if
there exists a partition $\Pi$ with respect to which it has circuit layout. (If it seems odd that a graph in “circuit layout” cannot actually have circuits, i.e. cycles — recall that boolean logic circuits are also acyclic and not literally circuits; it’s the electricity driving their operator physically that runs in circuits).

A last stepping-stone before finally defining a columnar circuit is the induction of the ports. Indeed, given a finite set $V$, a function $\mathbf{op}$ from $V$ into some set of available operators would induce a family of input and of output ports for each element of $V$:

$$\Lambda_{\text{in}}(V, \mathbf{op}) = \left\{ \{v\} \times \text{labels}_{\mathbf{op}(v)} \right\}_{v \in V} \quad \Lambda_{\text{out}} = \left\{ \{v\} \times \text{labels}_{\mathbf{op}(v)} \right\}_{v \in V}$$

and with these two defined, we denote by $\Lambda(V, \mathbf{op})$ their elementwise union, i.e. $\Lambda(V, \mathbf{op})$ are all the induced ports for each vertex. This completes the machinery necessary for:

**Definition 1.8 (Columnar circuit).** A tuple $C = (\sigma, V, \mathbf{op}, E, \pi)$ — a computational signature $\sigma$ (as per Definition 1.3), a set of vertices $V$, a mapping of vertices to operators $\mathbf{op}$ (which induces the ports of each vertex), a set of edges $E$, and an “interface mapping” $\pi$ of circuit signature labels to ports — constitutes a columnar circuit if the following hold:

1. The tuple $G_C = (V, \Lambda_{\text{in}}(V, \mathbf{op}), E)$ constitutes a port digraph — the layout graph or structure graph of $C$.

2. The structure graph $G_C$ has circuit layout with port orientation $\Pi = (\bigcup \Lambda_{\text{in}}(V, \mathbf{op}), \bigcup \Lambda_{\text{out}}(V, \mathbf{op}))$.

3. Edges only connect ports with the same associated element type, i.e. if $((u, a), (v, b)) \in E$ then $(\text{types}_{\mathbf{op}(u)}, \text{out})_a = (\text{types}_{\mathbf{op}(v)}, \text{in})_b$.

4. The interface $\pi$ maps of the circuit input ports to disengaged vertex input ports, and every disengaged input port has exactly one circuit input port mapped to it; in other words, $\pi|_{\sigma_{\text{in}}}$ is a bijection between $\text{labels}_{C,\text{in}}$ and the disengaged input ports of $\Lambda_{\text{in}}(G_C)$.

5. The interface $\pi$ maps circuit output labels to output ports in $\bigcup \Lambda_{\text{out}}(G_C)$.

Note. This definition does not support “pass-through” wires in a circuit, connecting an input to an output directly with no operator used; without a pass-through capability, circuits may require the use of an idempotent (Elementwise-identity-applying) operator to achieve the same functionality — relaying its input to its output. Similarly, unused input ports are not supported, but could be simulated using $\text{NoOp}$ nodes: If a circuit input is mapped by $\pi$ to the input of one of these, it can be considered unused.

**Definition 1.9 (Circuit input and output ports).** The input ports of a computational circuit $C$ is the set $P_{\text{in}}(C) = \pi_C(\text{labels}_{C,\text{in}})$; as per the constraint above, these are exactly all disengaged ports in $C$. The output ports of $C$ are $P_{\text{out}}(C) = \pi_C(\text{labels}_{C,\text{out}})$.

We come now to the central definition of our computational model, relating the combinatorial objects to computed functions:
**Definition 1.10 (Circuit-computed function).** Let $C = (\sigma, V, \text{op}, E, \pi)$ be a columnar circuit, let $v \in V$ with corresponding operator $\text{Op} = \text{op}(v)$ and input arity $r = |\sigma_{\text{Op}, \text{in}}|$ (which may be 0) and let $p = (v, a) \in P_C(v)$. We now inductively define a family of functions, $\{\varphi_p\}_{p \in P(C)}$, each taking input variables $X = \{x_\ell\}_{\ell \in \text{labels}_{C, \text{in}}}$ (the circuit $C$’s overall inputs); our definition is inductive:

$$
\varphi_p(X) = \begin{cases} 
  x_{\pi-1}(p) & \text{if } p \text{ is a disengaged input port} \\
  \varphi_{p'}(X) & \text{if } p = (v, a) \text{ is an engaged input port with } (p', p) \in E \\
  \varphi_{\text{Op}, a}(Y) & \text{if } p = (v, a) \text{ is an output port and } Y = \{\varphi_{(v, \ell)}(X)\}_{\ell \in \text{labels}_{C,\text{in}}(\sigma_{\text{Op}})} 
\end{cases}
$$

In other words: The inputs are cascaded through the columnar circuit, just like in a boolean logic circuit; nodes (operators) apply their associated function to their inputs, and the resulting outputs are “carried” by wires on to input ports of subsequent nodes.

Finally, The function computed by $C$, denoted $\varphi_C$, is $X \mapsto \{\varphi_{\pi(\ell)}(X)\}_{\ell \in \text{labels}_{C,\text{out}}}$.  

![Diagram of a columnar circuit](image)

**Figure 1.4: Resolution of the function computed by a columnar circuit**

To make the above definition more concrete, let us work out the function computed by a specific circuit: The one appearing in Figure 1.4. For brevity, we use EA, EM, R, L, S2 and S3 in the following instead of ElementwiseAdd, ElementwiseMultiply, Replicate, Length, Scalar 2 and Scalar 3 respectively, and the column names res and val instead of result and value.
\[ \varphi_C(\text{col}) = \varphi_{C, (\text{v7, res})} (\text{col}) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{C, (\text{v7, lha})}, \varphi_{C, (\text{v7, rha})}) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{C, (\text{v5, res})}, \varphi_{C, (\text{v6, res})}) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\varphi_{C, (\text{v5, lha})}, \varphi_{C, (\text{v5, rha})}), \varphi_{R, \text{res}} (\varphi_{C, (\text{v6, val})}, \varphi_{C, (\text{v6, length})})) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{C, (\text{v3, res})}), \varphi_{R, \text{res}} (\varphi_{C, (\text{v4, val})}, \varphi_{C, (\text{v4, length})})) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (\varphi_{C, (\text{v3, val})}, \varphi_{C, (\text{v3, length})})), \varphi_{R, \text{res}} (\varphi_{C, (\text{v2, val})}, \varphi_{C, (\text{v2, column})})) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (\varphi_{C, (\text{v1, val})}, \varphi_{C, (\text{v2, length})})), \varphi_{R, \text{res}} (\varphi_{S3, \text{val}} (\cdot), \varphi_{L, \text{res}} (\varphi_{C, (\text{v2, column})}))) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (\varphi_{S2, \text{val}} (\cdot), \varphi_{L, \text{res}} (\varphi_{C, (\text{v2, column})}))), \varphi_{R, \text{res}} (3, |\text{col}|)) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (2, \varphi_{L, \text{res}} (\text{col}))), \varphi_{R, \text{res}} (3, |\text{col}|)) \]

\[ = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (2, |\text{col}|)), \varphi_{R, \text{res}} (3, |\text{col}|)) \]

and applying this to a single arbitrary column element, we get

\[ \varphi_C(\text{col})[i] = \varphi_{EA, \text{res}} (\varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (2, |\text{col}|)), \varphi_{R, \text{res}} (3, |\text{col}|))[i] \]

\[ = \varphi_{EM, \text{res}} (\text{col}, \varphi_{R, \text{res}} (2, |\text{col}|))[i] \]

\[ = \varphi_{R, \text{res}} (3, |\text{col}|)[i] \]

\[ = \text{col}[i] \cdot \varphi_{R, \text{res}} (2, |\text{col}|)[i] + 3 \]

\[ = \text{col}[i] \cdot 2 + 3 \]

this functions computed on each port are also noted in Figure 1.4 itself.

An implementation of the computational model we’ve defined is a system (a program, a machine or a multi-node cluster of machines) which, given a columnar circuit and an assignment to its variables, evaluates \( \varphi_C \) as in the above example, on its entire domain, i.e. materializes the columns produced by circuit \( C \) when provided with columns as assignments to its inputs.

**Definition 1.11 (Decision circuit).** A (columnar) decision circuit is a columnar circuit \( C = (\sigma, V, \text{op}, E, \pi) \) with a single output type \( \tau_{\text{bit}} \), i.e. with \( \text{types}(\sigma_{C, \text{out}}) = \{ \tau_{\text{bit}} \} \). A decision circuit is said to accept input columns \( \{ c_{\tau} \}_{\tau \in \text{types}(\sigma_{C, \text{in}})} \) if \( \varphi_C (\{ c_{\tau} \}_{\tau \in \text{types}(\sigma_{C, \text{in}})}) = \text{TRUE} \), and to reject the input if \( \varphi_C (\{ c_{\tau} \}_{\tau \in \text{types}(\sigma_{C, \text{in}})}) = \text{FALSE} \). If \( \varphi_C \) is a complete function, i.e. \( C \) accepts or rejects all inputs, \( C \) is said to be a full decision circuit.

**Side note: Circuits vs instruction sequences**

The circuit model is the more fitting characterization of the kind of computation column stores execute, as there is no ordering among the vertices except through their data dependencies. Instruction sequences, the much more commonly used formalism for specifying computations, are ordered; and are also amenable to control flow denotations such as loops, conditionals and jumps.
— none of which are supported by the computational model we have presented. This being said, in sequential, non-vectorized/columnar execution of programs, out-of-order execution which does not interfere with data dependencies is common practice. And we could just constrain instructions to merely Single-Static-Assignment applications of columnar operators, defining a program to be an equivalence class of instruction sequences under permutations ensuring that all instructions producing the inputs of an operator application appear before this application.

1.3 Circuit composition and transformation

A traditional boolean circuits is not typically generated from scratch, but rather composed of subcircuits, computing simpler functions. The same should be the case with columnar circuits as well, for column store constructing execution plans. Such plans undergo repeated transformations after their initial generation — involving removal, addition and replacement of individual operators and entire subplans. This section presents some formal machinery for applying such transformations to columnar circuits. We will also be making extensive use of them for further conceptualization work in later chapters.

Definition 1.12 (circuit union). Let \( C_1 = (\sigma_1, V_1, \text{op}_1, E_1, \pi_1) \), \( C_2 = (\sigma_2, V_2, \text{op}_2, E_2, \pi_2) \) be two circuit, and assume \( V_1, V_2 \) are disjoint, as are \( \text{labels}_{1,\text{in}} \) and \( \text{labels}_{2,\text{in}} \). The (disjoint) union, or union circuit, of \( C_1 \) and \( C_2 \) is the circuit \( C_{\cup} = (\sigma_{\cup}, V_{\cup}, \text{op}_{\cup}, E_{\cup}, \pi_{\cup}) \) where

\[
\sigma_{\cup} = (\sigma_{1,\text{in}} \cup \sigma_{2,\text{in}}, \sigma_{1,\text{out}} \cup \sigma_{2,\text{out}}) \\
V_{\cup} = V_1 \cup V_2 \\
E_{\cup} = E_1 \cup E_2 \\
\text{op}_{\cup} = \text{op}_1 \cup \text{op}_2 \\
\pi_{\cup} = \pi_1 \cup \pi_2
\]

For the case of non-disjoint vertex or signature label sets, one may differentiate them by applying \( x \mapsto (1, x) \) for one circuit and \( x \mapsto (2, x) \) for the other.

If we order the input and output labels of circuits \( C_1 \) and \( C_2 \), the latter after the former, their union can also be thought of as a concatenation: It takes a concatenation of inputs to each of them and produces a concatenation of outputs from each of them (when the outputs are defined, of course).

Definition 1.13 (Circuit input assignment). Let \( C = (\sigma, V, \text{op}, E, \pi) \) be a columnar circuit whose signature contains the label \( \ell \), let \( p_1 = (v, b) = \pi_{C}(\ell) \) be the vertex input port to which \( \ell \) is mapped in \( C \), and let \( p_2 = (u, a) \) be a vertex output port within \( C \), with the same associated element type as \( p_1 \). The circuit resulting from the assignment of vertex output \( p_1 \) to input \( \ell \) in \( C \), or
the assignment of the $a$ output of $u$ to the $\ell$ input in $C$ is the circuit $C' = (\sigma', V', E', \text{op}', \pi')$ where

$$\sigma' = (\sigma'_\text{in}, \sigma'_\text{out})$$
$$\sigma'_\text{in} = \langle \text{labels}_{\text{in}}(\sigma'), \text{types}_{C,\text{in}} \rangle$$
$$\text{labels}_{\text{in}}(\sigma') = \text{labels}_{\text{in}}(\sigma) \setminus \{a\}$$
$$\text{types}_{C,\text{in}} = \text{types}_{C,\text{in}}|_{\text{labels}_{\text{in}}(\sigma')}$$
$$V' = V$$
$$E' = E \cup \{(p_2, p_1)\}$$
$$\text{op}' = \text{op}$$
$$\pi' = \pi \setminus \{a, p_1\}$$

The above definition is “intra-circuit”; but one may wish to engage a circuit’s input with the output of another circuit, rather than another vertex in the same circuit. This can be done by reducing the inter-circuit case to the intra-circuit case by taking the union circuit of the two circuits to be connected.

**Definition 1.14 (Induced subcircuit).** Let $C = (\sigma, V, \text{op}, E, \pi)$ be columnar circuit and let $V' \subseteq V$ (with $V'' = V \setminus V'$). The subcircuit of $C$ induced by $V'$ is a circuit with the layout of the sub-port-graph of $G(C)$ induced by $V'$, the ports of $C$ mapped to $V'$ by $\pi$, plus an artificial circuit port for every port $p$ of a $V'$ vertex whose connection from/to $V''$ is broken by taking the induced subgraph. We label these new artificial circuit ports using the $PC$ ports to which they had been connected, assuming without loss of generality that these labels will be disjoint from $\text{labels}(\sigma)$. More formally, the induced subcircuit, denoted $C|_{V'}$ is the tuple $(\sigma', V', \text{E}', \text{op}', \pi')$ with

$$E' = E|_{V'}, $$
$$\text{op}' = \text{op}|_{V'}, $$
$$\sigma' = (\sigma'_\text{in}, \sigma'_\text{out})$$
$$\sigma'_\text{in} = \langle \text{labels}_{\text{in}}(\sigma'), \text{types}_{C,\text{in}} \rangle$$
$$\text{labels}_{\text{in}}(\sigma') = L_{\text{retained}} \cup L_{\text{exposed}}$$
$$L_{\text{retained}} = \{\ell \in \text{labels}_{\text{in}}(\sigma) \mid \pi(\ell) \in P_{C,\text{in}}(V')\}$$
$$L_{\text{exposed}} = \{\ell_{(v,a)} \mid P_{C,\text{in}}(V') \text{ and } d(V'', (v,a)) > 0\}$$
$$\text{types}_{C,\text{in}} = \begin{cases} 
\text{types}_{C,\text{in}} & \ell \in L_{\text{retained}} \\
\text{types}_{C,\text{in}} & \ell = \ell_{(v,a)} \text{ for some } (v,a) \in P_{C,\text{in}}(V') 
\end{cases}$$
$$\sigma'_\text{out} \text{ defined similarly to } \sigma'_\text{in}$$
$$\pi'(\ell) = \begin{cases} 
\pi(\ell) & \ell \in L_{\text{retained}} \\
p & \ell = \ell_p \text{ for some } p \in P_{C,\text{in}}(V') 
\end{cases}$$
Induced subcircuit replacement  Let $C$ be columnar circuit, let $V' \subseteq V$ (and let $V'' = V \setminus V'$) and let $C_r$ be a replacement circuit for subcircuit $C|_{V'}$. The replacement of $C|_{V'}$ with $C_r$ in $C$ can be thought of in the same terms as replacing, say, an audio extension card in a personal computer, physically: The card has a line in and line out ports — part of the computer case’s overall ports facing the outside world — and the replacement card’s similar ports replace those. Additionally, the card has a a sequence of specifically-positioned metal strips which fit into the computer’s motherboard — its internal connections to rest of the computer system: While a card is connected these are irrelevant, or unexposed; when we disconnect the old card these spring into effective existence, and the new card must match these connections in order to physically fit in to the system.

More formally, let $\rho : P(C_r) \setminus P(C) \to P(C|_{V'}) \setminus P(C)$ be a bijection between the intra-$C$ ports of the replaced and replacement circuits. The circuit $\hat{C}$, resulting from the replacement, is defined by

$$\sigma_{\hat{C}} = \sigma_C$$
$$V_{\hat{C}} = V'' \cup V_{C_r} = (V_C \setminus V_{C'}) \cup V_{C_r}$$
$$\rho_{\hat{C}} = (\rho_{C} \setminus \rho_{C'}) \cup \rho_{C_r}$$
$$E_{\hat{C}} = E_C|_{V''} \cup E_{C_r} \cup \{(p_r, p'') \mid (\rho(c_r), p'') \in E_C\} \cup \{(p'', p_r) \mid (p'', \rho(c_r)) \in E_C\}$$

$$\pi_{\hat{C}}(\ell) = \begin{cases} \pi_C(\ell) & \pi_C(\ell) \in P_C(V'') \\
\rho(\pi_C(\ell)) & \text{otherwise} \end{cases}$$

Figure 1.5: An induced subcircuit of the columnar circuit from Figure 1.4
We have taken some pains to formally define this transformation of columnar circuits, due to its relevance to execution plan optimization in column stores; it is a general form of the key step in a typical plan execution optimizations in a column store: Find a subplan which is amenable to improvement; then replace it with with a more useful subplan.

**Operator lifting** A special case of induced subcircuit replacement with a different motivation for interest is the case of one of the circuits having just one node — one computational operator: Let $\text{Op}$ be a columnar operator. The tuple $C = (\sigma, V, E, \text{op}_C, \pi_C)$ with

$$\begin{align*}
\sigma &= \sigma_{\text{Op}} \\
V &= \{v_{\text{Op}}\} \\
E &= \emptyset \\
\text{op}_C &= \{(v_{\text{Op}}, \text{Op})\} \\
\pi_C &= \{(\ell, (v_{\text{Op}}, \ell)) \mid \ell \in \text{labels}_{\text{in}} C' \cup \text{labels}_{\text{out}} C'\}
\end{align*}$$

is a circuit, referred to as the lifting of operator $\text{Op}$ into a circuit $C_{\text{Op}}$.

Note that any circuit with a single vertex is, in fact, a lifted operator, up to a renaming of its external ports. Now, consider a larger circuit $C$, in which we perform a subcircuit substitution of a single-node subcircuit $C'$ with another circuit $C_r$. This is essentially the replacement of the operator lifted into $C'$ with a possible implementation, $C_r$. In programming language terms, it is reminiscent of the inlining of a function at its call site (and a transformation one expects to be followed by various local optimizations such as avoiding redundant copies, operators followed by their inverse and so on).

The opposite kind of subcircuit replacement — where a larger subcircuit is replaced by a lifted operator — is referred to as subcircuit fusion, and is of key importance for efficient implementation of the columnar circuit model; see Section 4.1 for details.
2

Columnar representation

A column store produces and utilizes a lot of data which isn’t merely plain-vanilla columns —
whether it be meta-data for schema columns; indexing structures; lower-dimension projections;
or intermediate results during the processing of a query. Having kept our computational model
simplistic and uniform for the purposes of defining the basic computational model, this section
will expand the expressive power using simple columns as building-blocks of representing these
structures, thus promoting them to “first-class” entities within a column store.

2.1 Encoding/decoding schemes for representation

Throughout this section fix some unidirectional signature $\sigma$, let $\mathcal{D} = \prod_{\tau \in \text{types}(\sigma)} \text{Dom}(\tau)$, and fix
$I \subseteq \mathcal{D}$ — a set of families of (labeled) columns of interest. Also, let $\sim$ be an equivalence relation on
$I$; without delving into the details — think of $\sim$ as the equivalence between different representations
of the same data.

Definition 2.1 (Decoding scheme). Let $(C_d, C_v)$ be a pair of columnar circuits — the decoder
and the encoded form verifier — computing functions $\varphi_d, \varphi_v$ respectively. Such a pair constitutes a
concrete decoding scheme for $I$ w.r.t. $\sim$ if:

1. The decoder and (encoded form) verifier have the same input signature (e.g. any possible
input to the decoder is also an input of the verifier and vice-versa).

2. The verifier is a full decision circuit (i.e. $\varphi_v^{-1}(\{\text{true, false}\}) = \mathcal{D}$).

3. If a family of columns is accepted by the verifier then the decoder produces output when on
those same columns as input (i.e. $\varphi_v(\text{Dom}(\varphi_d)) = \{\text{true}\}$).

4. The decoder only produces outputs in $I$ (specifically, the outputs have types $\vec{\tau}$).

5. The decoder produces output on all valid encoded forms, i.e. $\varphi_v^{-1}(\{\text{true}\}) \subseteq \varphi_d^{-1}(I)$.

6. Every column tuple $\vec{x} \in I$ has a valid encoded form which decodes into $\vec{x}$, or into a $\sim$-equivalent
tuple (i.e. $\varphi_d$ hits every equivalence class of $\sim$).
A decoding scheme for \( I \) w.r.t. \( \sim \) is a function \( f_d \), for which there exists a concrete decoding scheme \((C_d, C_v)\) where the domain of \( f_d \) are the valid encoded forms for the concrete scheme, and \( \varphi_d \) is identical to \( f_d \) on those valid encoded forms (i.e. \( \text{Dom}(f_d) = \varphi_v^{-1}\{\text{TRUE}\} \) and \( f_d = \varphi_d|_{\text{Dom}(f_d)} \)).

\[ \text{Note.} \] The verifier circuit in Definition 2.1 is an artifice allowing us to mostly circumvent the issue of decoder input validity, and describe decoding circuits which presume their inputs are valid — significantly simplifying their structure and the operators they are made up of. Thus Indeed, the computational complexity of deciding input validity may be higher than actually performing the decoding on valid inputs (as our model is circuit-based, not timed-machine based). Verifiers will not be of much significance in the remainder of this document, which is not concerned with invalid inputs and their handling.

A decoding scheme in which the decoder produces a single column \((k = 1)\) is a column decoding scheme.

**Definition 2.2 (Encoding scheme).** A columnar circuit \( C_e \), computing function \( \varphi_e \) constitutes a concrete encoding scheme for \( I \) w.r.t. \( \sim \) if:

1. The (encoder) circuit’s input signature is \( \sigma \).
2. \( I \subseteq \text{Dom}(\varphi_e) \).
3. If \( X, Y \in I \) satisfy \( \varphi_e(X) = \varphi_e(Y) \) then \( X \sim Y \).

An encoding scheme for \( I \) w.r.t. \( \sim \) is a function \( f_e \), for which there exists a concrete encoding scheme \( C_e \) with \( f_e = \varphi_e|_I \).

**Definition 2.3 (Codec).** A concrete codec for \( I \) w.r.t. \( \sim \) is a tuple \((C_e, C_d, C_v)\) such that \( C_e \) is a concrete encoder and \((C_d, C_v)\) is a concrete decoder (w.r.t. \( I, \sim \)), and the following hold:

1. All encoder outputs for \( I \) are valid encoded forms, i.e. \( \varphi_e(I) \subseteq \varphi_v^{-1}\{\text{TRUE}\} \). (The converse statement is valid for any concrete decoder and encoder for \( I \).)
2. Decoding the encoding of a tuple of columns in \( I \) produces an equivalent tuple of columns (i.e. letting \( f = \varphi_d \circ \varphi_e \), we have \( f(X) \sim X \) for every \( X \in I \)).

A codec is a pair \((f_d, f_e)\) of a decoding and an encoding scheme satisfying the above conditions (substituting \( f_d \) for \( \varphi_d \) and \( f_e \) for \( \varphi_e \)).

**Definition 2.4 (Column representation scheme).** A representation scheme for columns of type \( \tau \) is a codec for columns (i.e. with \( k = 1 \)), with \( I \) being all columns of this element type and with \( \sim \) being the equality relation.

The definitions above allow for multiple valid encoded forms (or representations) for a single uncompressed column-tuple. Occasionally, we will be interested both in such “lax” schemes and their “narrowing”, by the imposition of further constraints on the encoded forms:

**Definition 2.5 (Sub-scheme).** A decoding scheme \( f \) constitutes a sub-decoding-scheme, or sub-scheme, of another decoding scheme \( f' \), if \( \text{Dom}(f) \subseteq \text{Dom}(f') \) and \( f'|_{\text{Dom}(f)} = f \).

We similarly define a sub-codec and a sub-representation scheme.
2.2 Subcolumns

2.2.1 Explicitly-indexed columns

Before proceeding to define and model subcolumns, we first take a detour to introduce an additional representation scheme for (complete) columns. Its measure of redundancy will make it easy to adapt it, further below, to subcolumns.

Consider the following operators:

**Iota**

| In/out? | Label | Type | Length | Description |
|---------|-------|------|--------|-------------|
| Input   | n     | τ    | 1      | Intended length |
| Output  | result | τ    | n      | Generated column |

Produces the identity column of a specified length, i.e. \( \text{result}[i] = i \).

**Permute**

| In/out? | Label | Type | Length | Description |
|---------|-------|------|--------|-------------|
| Input   | permutation | τ    | n      | Permutation of \( \{0, \ldots, n-1\} \) |
| Input   | data   | τ    | n      | Column to permute |
| Output  | permuted | τ    | n      | |

Produces the result of applying the permutation encoded in \( \text{old_pos} \) to the data in \( \text{data} \), i.e. a column satisfying \( \text{permuted}[i] = \text{data}[\text{permutation}^{-1}[i]] \).

**Length**

| In/out? | Label | Type | Length | Description |
|---------|-------|------|--------|-------------|
| Input   | col   | τ    | n      | |
| Output  | result | τ    | n      | Set to \( n \), i.e. \(|\text{col}|\) |

Produces the length of the input column \( \text{col} \) as a scalar.

If we **Permute** a column, the original order of elements is lost, and the original column cannot be restored without known the permutation used. However, if we were to add a second column alongside our original one, and make it indicate positions — the permutation would leave us with sufficient information to undo them. In fact, the result of permuting the output of **Iota** is the permutation. We thus define:

**Indexed**

| Label | Type | Length | Description |
|-------|------|--------|-------------|
| pos   | τ    | n      | element positions in the column |
| data  | τ    | n      | element values |

The scheme induced by the concrete decoder being the lifting of **Permute**, and the concrete encoder mapping a column \( \text{c} \) to \((\text{Iota}(|\text{c}|), \text{c})\).

*Note.* This representation corresponds to the Decomposition Storage Model (DSM) [CK85]. For many years it was also the default internal representation scheme for columns in MonetDB (albeit not using the formalism in this paper), referred to as a BAT (Binary Association Table); \((\text{pos}, \text{data})\) here corresponds to the BAT “head” and “tail” parts [MKB09, §3]. Recently, MonetDB has dropped this scheme [Mul16] except for some vestigial aspects of the API for columns. In C-Store (and likely
Vertica), columns are not materialized this way — but iteration over position-and-value pairs is a fundamental part of the interface for all columns [AMDM07, §2].

![Standard representation](image1)

![Indexed representation of the same column — result of encoding the column](image2)

![Indexed representation — a valid encoded form not obtained by applying the encoder](image3)

**Figure 2.1: The Indexed column representation scheme**

This representation scheme is highly redundant. It does, however, preclude the need to consider the index of corresponding elements within the two column, i.e. one can make do with just the tuple set \{(indices[i], data[i]) | 0 ≤ i < n\}, decoding each tuple independently.

### 2.2.2 A subcolumn representation

**Definition 2.6 (Subcolumn).** A subcolumn is a partial function \(sc : \mathbb{N} \rightarrow \tau\) whose domain (of definition) is a finite subset \(D \subseteq \mathbb{N}\).

A subcolumn \(sc\) is said to be contiguous if its domain is a contiguous subset of \(\{0, \ldots, \max(D) - 1\}\), and incontiguous otherwise.

**Observation 2.7.** A column can be thought of as the special case of a contiguous subcolumn whose domain contains 0 (hence, with domain \(D = \{0, \ldots, n - 1\}\)). This reflects our choice to define subcolumns independently of any specific super-column, or even a specific-size supercolumn domain.

Let \(sc, sc'\) be two subcolumns of the same element type which agree as partial functions. \(sc\) is said to be a subcolumn of \(sc'\) if their domains satisfy \(\text{Dom}(sc) \subseteq \text{Dom}(sc')\), and they agree on \(\text{Dom}(sc)\); this is denoted \(sc \subseteq sc'\).

As in the case of columns, we choose one representation scheme as the standard scheme; any other representation scheme is defined by a codec relative to the standard one.

**Subcolumn** (Standard representation of a subcolumn)

| Label | Type | Length | Description |
|-------|------|--------|-------------|
| pos   | \(\tau_{\text{int}}\) | \(n'\)  | Non-negative values |
| data  | \(\tau\) | \(n'\)  |        |

The represented subcolumn \(sc\) is the partial function \(x \mapsto \text{data}(\text{pos}^{-1}(x))\), i.e. \(sc[i] = \text{data}[\text{pos}^{-1}[i]]\). Its domain is \(\text{Img}(\text{pos})\).
Observation 2.8. For Dom(sc) = {0, ..., n − 1}, the full domain, a standard representation of a subcolumn is a representation of it as a column using the INDEXED encoding scheme; as in that scheme, there are multiple standard representations for each subcolumn (one for every of the |D|! permutations of the domain elements).

Figure 2.2: The standard subcolumn representation scheme

Definition 2.9 (Canonical representation of subcolumns). The canonical representation of a subcolumn is its standard representation in which the elements of pos are monotone increasing.

Observation 2.10. A subcolumn sc is contiguous if and only if the pos column in its canonical representation is contiguous.

Definition 2.11 (Subcolumn decoding & encoding schemes). A subcolumn decoding (respectively, encoding) scheme is a decoding (respectively, encoding) scheme (see Definition 2.1 and the following text) for pairs of columns, of types (τ_{int}, τ) — an integer type τ_{int} and an arbitrary type τ — with I being the set of all standard subcolumn representations, and ∼ being the equivalence relation among standard subcolumn representations. A subcolumn codec is a codec for the type pair (τ_{int}, τ).

As in the case of columns, decoding schemes for the set of all subcolumns of a type τ induce representations of type-τ subcolumns in addition to the standard one.

2.2.3 Combining subcolumns

Definition 2.12 (Subcolumn overlay). Let sc₁, sc₂ be two n-subcolumns with domains D₁, D₂ respectively. The overlay of sc₁ by sc₂, denoted sc₁ ← sc₂, is the partial function:

\[ sc(x) = \begin{cases} sc_1(x) & x \in D_1 \setminus D_2 \\ sc_2(x) & x \in D_2 \end{cases} \]
Two subcolumns are said to be *compatible* if they agree (as functions) on all elements of the intersection of their domains.

**Observation 2.13.** The following are equivalent regarding two subcolumns $sc_1$, $sc_2$ of the same type:

1. $sc_1$ and $sc_2$ are compatible.
2. There exists a column $c$ such that $sc_1 \subseteq c$ and $sc_2 \subseteq c$.
3. $sc_1 \leftarrow sc_2$ is a super-column of both $sc_1$ and $sc_2$.

Two subcolumns being compatible makes for a special case of the above definition:

**Definition 2.14 (Subcolumn union).** Let $sc_1$, $sc_2$ be two compatible subcolumns. The *union subcolumn* of $sc_1$, $sc_2$ is the subcolumn $sc_1 \cup sc_2 = sc_1 \leftarrow sc_2$.

**Observation 2.15.** The relation $\subseteq$ is a partial order over all subcolumns of a fixed column; moreover, the set of such subcolumns and the $\subseteq$ partial order form a lattice w.r.t. intersection and union — isomorphic to the lattice of partial sets of a fixed set.

The above modes of combining columns and subcolumns are immediately useful in defining representation schemes:
**SubcolumnUnion**

| Label   | Type | Length | Description                      |
|---------|------|--------|----------------------------------|
| pos₁    | τ_int| n'     | First subcolumn, element positions|
| data₁   | τ    | n'     | First subcolumn, element data     |
| pos₂    | τ_int| n''    | Second subcolumn, element positions|
| data₂   | τ    | n''    | Second subcolumn, element data    |

A subcolumn representation scheme constituting standard representations of two compatible subcolumns.

**DisjointSubcolumnUnion**

is a restriction of SubcolumnUnion to pairs of disjoint subcolumns. This is still a representation scheme, since the restriction is of the representations (that is, encoded forms), not of the represented subcolumns.

**SubcolumnOverlay**

is the subcolumn representation scheme similar to SubcolumnUnion, except that the two subcolumns need not be compatible, and the decoding result is the overlay of the first subcolumn by the second one.

In fact, some of these representation schemes inspire corresponding (full-)column representation schemes:

**ComplementingSubcolumns**

| Label   | Type | Length | Description                      |
|---------|------|--------|----------------------------------|
| pos     | τ_int| n'     | Element positions                |
| data₁   | τ    | n'     | First subcolumn, element data    |
| data₂   | τ    | n − n' | Second subcolumn, element data   |

An adaptation of DisjointSubcolumnUnion to the full-column case: pos₁ and pos₂ are each other’s complement, so only one of them (say, the first) is encoded; and the decoding produces a standard representation of a column, not a subcolumn.

**ColumnByOverlay**

Same as ComplementingSubcolumns, except that the two subcolumns need not have disjoint domain, and the decoding result is the overlay of the first subcolumn by the second one.

**OverlaidColumn**

| Label            | Type | Length | Description            |
|------------------|------|--------|------------------------|
| data             | τ    | n      | Main column            |
| overlay_pos      | τ_int| n'     | Subcolumn positions    |
| overlay_data     | τ    | n'     | Subcolumn data         |

Similar to ColumnByOverlay, except that instead of two subcolumns we now have a full column and an overlaying subcolumn. This scheme can be decoded by the lifting of the Scatter operator (see below). (Also referred to as Patched — the individual position-datum pair are like patches applied to the fabric of the data column).

These schemes can all be shown to have concrete codecs. To illustrate what some of the decoders are like, consider the following operators:
Concatenate

| In/out? | Label | Type | Length | Description |
|---------|-------|------|--------|-------------|
| Input   | col_1 | τ    | n_1   |             |
| Input   | col_2 | τ    | n_2   |             |
| Input   | ...   |      |       |             |
| Input   | col_k | τ    | n_k   |             |
| Output  | result| τ    | ∑_{j=1}^{k} n_j |             |

Produces the concatenation of the two input columns, i.e. the column

\[
\text{result}(x) = \begin{cases} 
\text{col}_1(x) & 0 \leq x < n_1 \\
\text{col}_2(x - n_1) & n_1 \leq x < n_1 + n_2 \\
\vdots & \\
\text{col}_k(x - \sum_{j=1}^{k-1} n_j) & \sum_{j=1}^{k-1} n_j \leq x < \sum_{j=1}^{k} n_j 
\end{cases}
\]

This operator does not concatenate individual elements into larger elements — it’s only columns that get concatenated.

Scatter

| In/out? | Label | Type | Length | Description |
|---------|-------|------|--------|-------------|
| Input   | col   | τ    | n     | Target to scatter onto |
| Input   | pos   | τ    | n'    | Positions in target into which to scatter |
| Input   | data  | τ    | n'    | Data to place into target |
| Output  | result| τ    | n     | A copy of col with the positions specified by pos overriden by the corresponding elements in data. |

Produces the column

\[
\text{result}[i] = \begin{cases} 
\text{data}[\text{pos}^{-1}(i)] & i \in \text{Img}(\text{pos}) \\
\text{col}[i] & \text{otherwise}
\end{cases}
\]

Examples of the use of these three operators appear can be seen in Figure 2.5.

![Concatenate Diagram](image1)

![Scatter Diagram](image2)

Figure 2.5: Decoders for representation schemes involving subcolumn combinations

## 2.3 Segmented columns

**Motivation** Much of the processing in concrete column stores is applied limited-size chunks, segments or blocks: Pages on a magnetic disk drive or in DRAM; the data fitting inside one
of a CPU’s cache levels; a GPU multiprocessor’s shared memory; an FPGA’s bus width and its synthesized circuit data width; and so on. Many, if not most, existing column stores have chunks or segments as a fundamental abstraction, and partitions into chunks as a fundamental system feature (Some examples: Vectorwise [ZBNH05, ZHNB06], Google PowerDrill [HBB+12, §2.3], C-Store/Vertica [LMV+12]; a notable exception is MonetDB [Mona]). Yet, the computational model presented in Chapter 1 is based on a uniform, unbroken definition of a column. We have opted for this simpler abstraction, as both the model itself and the representation and compression schemes this work presents would have been very unwieldy, had they beed saddled with segmentations of each column.

At the same time, we do intend to model existing systems, so we cannot simply ignore column segmentation. Segmentations are also inherent to several key data compression schemes, a subject which Chapter 3 will explore in more depth. We therefore presented segmented columns through the use of non-segmented ones, rather than the other way around.

First, suppose we merely wish to regard a column’s indices — the sequence \( \{0, \ldots, n - 1\} \) — in contiguous segments. These can be represented as follows:

**Segmentation**

| Label         | Type | Length | Description                              |
|---------------|------|--------|------------------------------------------|
| start         | \(\tau\) | \(m\)    | ...of the first element of each segment  |
| length        | \(\tau\) | \(m\)    | Non-negative number of elements in each segment |

There are no gaps between the segments, that is, \(\text{start}[i] = \text{start}[i - 1] + \text{length}[i]\) for every \(i \in \{1, \ldots, m - 1\}\). It must also hold that \(\text{start}[0] = 0\) and \(\text{start}[m - 1] + \text{length}[m - 1] = n\).

**UniformSegmentation**

| Label            | Type | Length | Description                      |
|------------------|------|--------|----------------------------------|
| segment_length   | \(\tau\) | 1      | Length of the segmented columns  |
| overall_length   | \(\tau\) | 1      | Length of the segmented columns  |

A degenerate scheme for the special case of segment length being uniform; the number of segments is \(\lceil \text{overall_length}/\text{segment_length} \rceil\).

One may verify that these two qualify as representation schemes, with the represented columns being the identity functions of any length. Also, the length values may be 0 for the non-inform-length case; when this occurs, the segmentation is said to be degenerate.

Now let’s apply these segmentation schemes to columns of actual data:

**Segmented**

| Label             | Type | Length | Description                          |
|-------------------|------|--------|--------------------------------------|
| data              | \(\tau\) | \(n\)    | actual column data                   |
| segment_start_pos | \(\tau\) | \(m\)    | of the first element of each segment  |
| segment_length    | \(\tau\) | \(m\)    | Non-negative number of elements in each segment |

A standard representation of a column and an accompanying Segmentation.

**UniformlySegmented**

| Label             | Type | Length | Description |
|-------------------|------|--------|-------------|
| data              | \(\tau\) | \(n\)    | actual column data |
| segment_length    | \(\tau\) | 1      | segment length |
A standard representation of a data column; only one of the scalars in a **UniformSegmentation** is additionally necessary, as the data column length substitutes for the overall_length of a **UniformSegmentation**.

The two kinds of column segmentation — fixed- and variable-length segments — differ quite significantly not just in their representation, but in their motivations for use; but — that discussion will wait until Subsection 3.2.3, below. The rest of this section will focus on the interpretation and manipulation of segmented columns (mostly with fixed-segment-length).

### 2.3.1 Uniform segmentation and nearly-matrices

**Definition 2.16 (Segmented view).** Let \( c \) be a column of length \( n \) and \( \ell \in \mathbb{N} \). The \( \ell \)-segmented view of \( c \) is the function \( (i, j) \mapsto c[i \cdot \ell + j] \), defined whenever \( i \cdot \ell + j < n \).

With this function in mind, and when \( \ell \) is obvious from the context, we abbreviate \( c[i \cdot \ell + j] \) as \( c[i, j] \).

The segmented view is the conceptualization of the break-up into segments defined above: Consecutive, aligned, length-\( \ell \) segments are perceived as short columns in a \( \lceil n/\ell \rceil \)-column **nearly-matrix**. This view is matrix-like; but it cannot generally correspond to a proper matrix, since the last shorter column — the last segment — will be missing some elements when \( \ell \nmid n \).

![Figure 2.6: The UniformSegmented representation scheme, \( \ell \)-segmented views and nearly-matrices](image)

Still adhering to the matricial interpretation, a 1-segmented view of a column can be thought of as its transposition — a single ‘row’ of length \( n \) (but bear in mind it is merely the replacement of \( i \mapsto c[i] \) with \( (1, i) \mapsto c[i] \)). More generally, transposition is defined as follows:

| In/out? | Label            | Type   | Length          | Description               |
|---------|------------------|--------|-----------------|----------------------------|
| Input   | segment_length   | \( \tau_{\text{int}} \) | 1               |                            |
| Input   | col              | \( \tau \)             | \( \text{segment_length} \cdot k \) |                            |
| Output  | transposed       | \( \tau \)             | \( \text{segment_length} \cdot k \) |                            |
| Output  | transposed_segment_length | \( \tau_{\text{int}} \) | 1               |                            |
Produces a permutation of \( \text{col} \) such that \( \text{transposed}[i,j] = c[j,i] \) for every \( i,j \in \{0, \ldots, k-1\} \times \{0, \ldots, \text{segment\_length} - 1\} \). The transposed segment length is exchanged with the number of segments.

Transposition requires \( \text{col} \) to a full matrix (i.e. have a full-length last segment), as otherwise its result does not form a nearly-matrix, i.e. there would be gaps in the resulting ‘column’ due to their origin elements under transposition missing. This contrasts with a a segment length change (a re-segmentation if you will), which is possible for any column length.

![Figure 2.7: Rearranging data in a segmented view: Segment length change vs. transposition](image)

### 2.3.2 Data replication within and across segments

Columnar circuits for decoding and encoding schemes occasionally involve replicating the values of short columns or scalars into longer columns; and this is the case also for column segments (as will become apparent in Subsection 2.3.3). Additionally, the matrix-like segmented views lend themselves to thinking of two kinds of replication of data, corresponding to the two dimensions of the segmented-view ‘matrix’: Column-wise and row-wise replication:

| In/out? | Label         | Type  | Length                              | Description                                                                 |
|---------|---------------|-------|-------------------------------------|-----------------------------------------------------------------------------|
| Input   | \( \text{col} \) | \( \tau \) | \( n = \text{segment\_length} \cdot k \) | Equal to the input of the same name.                                       |
| Input   | \( \text{segment\_length} \) | \( \tau_{\text{int}} \) | 1 |                                                                             |
| Input   | \( \text{factor} \) | \( \tau_{\text{int}} \) | 1 |                                                                             |
| Output  | \( \text{replicated} \) | \( \tau \) | \( \text{factor} \cdot n \) |                                                                             |
| Output  | \( \text{segment\_length} \) | \( \tau_{\text{int}} \) | 1 |                                                                             |

Produces a UniformlySegmented column with the same segment length as the input but with \( \text{factor} \cdot k = \text{factor} \cdot (n/\text{segment\_length}) \) segments, with column \( i \) in the segmented view of \( \text{replicated} \) being equal to column \( i/\text{factor} \) in the segmented view of \( \text{col} \). In other words, each segment is replicated into \( \text{factor} \) consecutive copies of itself in the output.

Replicate
 Produces a UniformlySegmented column with longer segments than the input, but the number of segments remains the same. The new segment length is factor times the old length; each element in each of these segments is replicated factor times, consecutively, within its segment, so that the jth element in the result segment equals the (j/factor)th element of the original segment.

Note that the Replicate operator, defined in Subsection 1.1.2 for unsegmented columns of length 1, agrees with this definition — it is the special case of this definition where the input has just a single segment — the entire column. See also Subsection 2.3.3 below regarding similar generalizations.

| In/out? | Label       | Type | Length   | Description                  |
|---------|-------------|------|----------|------------------------------|
| Input   | col         | τ    | n        | \( n = \text{segment\_length} \cdot k \) |
| Input   | segment\_length | τ\_int | 1       |                               |
| Input   | factor      | τ\_int | 1       |                               |
| Output  | replicated  | τ    | factor \cdot n | = factor \cdot \text{segment\_length} |
| Output  | segment\_length\' | τ\_int | 1       |                               |

2.3.2.1 Uniform segmentation and subcolumns

We would like to have a meaningful concept of “taking a subcolumn” of an ℓ-segmented view, rather than a plain column; naturally, this means respecting the segmentation when discarding some of the data. To be more explicit: A permutation of \{0, \ldots, n − 1\} is said to respect ℓ-segmentation if the image of every ℓ-segment of the domain is an ℓ-segment as well (e.g. \( ℓ \cdot i, \ldots, ℓ \cdot (i + 1) − 1 \) becomes \( ℓ \cdot i', \ldots, ℓ \cdot (i' + 1) − 1 \)); and the final sub-ℓ-length segment, if it exists, remains in place.

A subcolumn sc is said to respect the ℓ-segmentation of a supercolumn column c if each (short) column in c’s ℓ-segmented view is either entirely within Dom(sc) or does not intersect Dom(sc) at all.

**Observation 2.17.** An ℓ-segmentation-respecting permutation of the pos column of an ℓ-segmentation-respecting subcolumn can be uniquely determined with just the first of every ℓ contiguous elements.

For our well-behaved subcolumns and permutations, this observation leads to a more space-efficient representation:
Instead of a straight-up permutation of the entire index set, this scheme describes a permutation of \( n' \) elements. The positions are in units of \( \ell \) elements, i.e. if \( k \in \text{Img}(\text{segment pos}) \) then the decoded subcolumn’s domain contains \( k\ell, k\ell + 1, \ldots, k\ell - 1 \).

\[
\begin{array}{c|c|c}
\text{segment length} & \text{segment pos} & \text{data} \\
\hline
3 & 2 & 0 \ 3 \\
2 & 3 & 0 \ r \ h \ d \\
1 & 1 & \end{array}
\]

\[
\begin{array}{c|c|c}
\text{segment length} & \text{segment pos} & \text{data} \\
\hline
3 & 2 & 0 \\
1 & 1 & \end{array}
\]

\[
\begin{array}{c|c|c}
pos & \text{data} \\
\hline
6 & 0 \\
7 & r \\
8 & 1 \\
9 & h \\
1 & e \\
2 & 1 \\
9 & d \\
\end{array}
\]

**Figure 2.9:** The subcolumn from Figure 2.6 (“helloworld” without the “low”) encoded with the \text{SegmentedSubcolumn} scheme.

**Figure 2.10:** Decoder for the \text{SegmentedSubcolumn} representation scheme

2.3.3 Adapting operators to column segmentation

To actually utilize segmented columns — to apply columnar circuits to them — we require operators which respect such segmentation (either fixed- or variable-length). Specifically, we would want to adapt operators we have already defined for non-segmented columns to take segmented column inputs, and produce outputs for each of the input column segments.

**Definition 2.18 (Simple, fixed-length operator segmentation).** Let \( \text{Op} \) be a columnar operator taking an input column labeled \( c \) of type \( \tau \) and producing a single output labeled \( c' \), whose length is a function of \( |c| \). The segmentation of \( \text{Op} \) is an operator \( \text{Segmentize}(\text{Op}) \), taking a \text{UniformlySegmented} input and producing a \text{UniformlySegmented} output. \( \text{Segmentize}(\text{Op}) \)
applies \( \text{Op} \) to every one of the \( m = \lceil |c|/\text{segment\_length} \rceil \) segments of the segmented view of \( c \). The resulting would-be output columns (denote them \( c'_1 \) through \( c'_m \)) are concatenated together into the \text{data} component of \( c' \). By our constraint on \( \text{Op} \), all segment outputs are of the same length, which is used as the \text{segment\_length} component of \( c' \).

This definition can be generalized to the case of multiple input and output columns:

**Definition 2.19 (General operator segmentation).** Consider a signature \( \sigma = (\sigma_{\text{in}}, \sigma_{\text{out}}) \), let \( L_{\text{shared}} \subseteq \text{labels}_{\text{in}}(\sigma_{\text{Op}}) \) be a subset of inputs to be shared among all segments, let \( L_{\text{segmented}} = \text{labels}_{\text{in}}(\sigma) \setminus L_{\text{shared}} \), and let \( \text{Op} \) be a columnar operator with signature \( \sigma \). The segmentation of \( \text{Op} \) is an operator \( \text{Segmentize}(\text{Op}) \) with the following characterization. \( \text{Segmentize}(\text{Op}) \) has the same input labels as \( \text{Op} \) with labels in \( L_{\text{shared}} \), but for every input column labeled \( c \in L_{\text{segmented}} \), \( \text{Segmentize}(\text{Op}) \) takes a segmented column \( c' \) (in the SEGMENTED or UNIFORMLYSEGMENTED encoding scheme) of the same data element type. \( \text{Segmentize}(\text{Op}) \) produces output only if the its inputs from \( L_{\text{segmented}} \) all have the same number of segments. When that is the case, for each \( c_{\text{out}} \in \sigma_{\text{out}} \), \( \text{Segmentized}(\text{Op}) \) produces a SEGMENTED column, with the data being concatenation of \( \text{Op}'s \) \( c_{\text{out}} \) outputs for each of the input sets for the various segments, and a segmentation corresponding to these outputs’ lengths.

In formulating this definitions, one notices how fixed-length segmentation in columns lacks robustness: If an operator is applied whose output length depends on input \text{data} rather than just input \text{lengths} (say, a range filter) — different output column segments will have different lengths; thus the outputs must be defined as variable-length segmented. For this reason the definition above does not use UNIFORMLYSEGMENTED outputs; for those to be possible, we would need a constraint on the operator involved to conserve length uniformity.

An application of the unsegmented operator

| \text{data} | \text{permutation} | \text{permuted} |
|---|---|---|
| 6 | 2 | 7 |
| 4 | 3 | 7 |
| 1 | 3 | 7 |
| 0 | 1 | 7 |

An application of the segmentized operator — multiple permutations of the same data; the input is presented in 4-segmented view.

**Figure 2.11:** A segmentation of the 
\text{Permune} \text{ operator with } L_{\text{shared}} = \{\text{data}\} \text{ and } L_{\text{segmented}} = \{\text{permutation}\}.

### 2.4 Representing additional constructs

#### 2.4.1 Index subsets

Recall that our standard representation of a subcolumn contains two columns: \text{positions} and \text{values}. Now consider the result of dropping the \text{values} column: Doing so will leave us with the information regarding which indices are the subcolumn’s domain; and since the order of indices in this columns is incosequential — what we are left with is the specification of an \text{index subset}. As in the definition of subcolumns, we must choose whether to be ignorant of the overall index set size
(resp. subcolumn domain); and for index sets we make the opposite choice, marking the domain size by default:

**SparseIndexSet** (Standard representation of an index subset)

| Label      | Type   | Length | Description                                      |
|------------|--------|--------|-------------------------------------------------|
| full_length| int    | 1      | size of the full set of indices                 |
| elements   | int    | up to full_length | values of elements in the index subset, in any order |

If the `elements` column is also sorted, the subset representation is said to be *canonical*.

The standard representation is sparse in the sense of the underlying assumption that elements not specified in it are missing from the set. Alternatively, we can choose to have an explicit indication of presence for every potential element — a *dense* representation — using the following:

**Definition 2.20 (Set characteristic column).** Let $S \subseteq \{0, \ldots, x - 1\}$. The *characteristic column* of $S$ (a.k.a. the *characteristic function* of $S$) is the function $\chi_S : \{0, \ldots, n - 1\} \rightarrow \{\text{true, false}\}$ with

$$
\chi_S(x) = \begin{cases} 
\text{true} & x \in S \\
\text{false} & \text{otherwise}
\end{cases}
$$

The set of characteristic columns for all $S \subseteq \{0, \ldots, n - 1\}$ is in bijection with the power set $\wp(\{0, \ldots, n - 1\})$ itself; and thus we choose the characteristic columns — the evaluated characteristic functions — as the **DenseIndexSet** representation scheme for subsets of $\{0, \ldots, n - 1\}$. A concrete encoder and decoder for this scheme appear in Figure 2.12.

![Diagram](image)

**Figure 2.12: The DenseIndexSet representation scheme**

The decoding and encoding circuits we’ve presented for **DenseIndexSet** use two yet-undefined operators. These are:

**SelectIndices**

| In/out? | Label     | Type            | Length | Description |
|---------|-----------|-----------------|--------|-------------|
| Input   | characteristic | $\tau_{\text{bit or \{true, false\}}}$ | $n$    |             |
| Output  | indices   | $\tau_{\text{int}}$ | up to $n$ |             |

Produces a column containing the indices of all elements of *dense* with value *true* (or 1 if we’re using the numeric type). The output is not necessarily sorted in increasing order.
Length

Given a column of type $\tau$, produces its length as a scalar (i.e. a column of length 1).

Columns of bits corresponding to a subset a column’s (or rather, DBMS table’s) indices have long been in use in analytic query processing, well before the advent of column stores (an early example is the Model 204 DBMS by Compute Corporation of America [O’N89], which predates even the era of relational DBMSes and SQL).

The characteristic function of a subset is often computed in practice using some criterion or predicate on a column’s values. This motivates the following scheme, in which the application of the criterion is “lazy” (i.e. only applied when decoding):

$\text{ImplicitDenseIndexSet}_p$

| Label                  | Type | Length | Description                                      |
|------------------------|------|--------|--------------------------------------------------|
| predication_subject    | $\tau$ | $n$    | data to which the predicate $p$ is applied       |

$p$ must be a function which is not uniform on the domain of $\tau$. Specifically, let $a, b$ be values of type $\tau$ with $p(a) = \text{false}$ and $p(b) = \text{true}$). This scheme is decoded by applying $\text{Elementwise}_p$, then applying the decoder for $\text{DenseIndexSet}$. Encoding begins with a $\text{DenseIndexSet}$ encoder, followed by the elementwise mapping of $\text{false} \mapsto a$ and $\text{true} \mapsto b$.

Finally, we define highly concise representations for subsets of $\{0, \ldots, n - 1\}$ which are either contiguous, or are a union of contiguous ranges.

$\text{ContiguousIndexSet}$

| Label | Type | Length | Description                                      |
|-------|------|--------|--------------------------------------------------|
| start | $\tau_{\text{int}}$ | 1      | Index of the first element in the set            |
| length| $\tau_{\text{int}}$ | 1      | Number of elements in the set                    |

The represented set is $\{\text{start}[0], \ldots, \text{start}[0] + \text{length}[0]\}$.

2.4.2 Partitions of columns and index sets

In Section 2.3 we presented representation schemes for the break-up of columns into contiguous segments; let us now generalize this to arbitrary partitions — in which the parts are not necessarily contiguous.

If the number of parts is fixed (say, $k$), we could simply adjoin the (standard) representations of $k$ subcolumns:

$\text{Partitioned}_k$

| Label  | Type | Length | Description                                      |
|--------|------|--------|--------------------------------------------------|
| $\text{pos}_1$ | $\tau_{\text{int}}$ | $n_1$   | Element positions of the first subcolumn            |
| $\text{data}_1$ | $\tau$        | $n_1$   | Element data of the first subcolumn                 |
| ...                     |              |         |                                                   |
| $\text{pos}_k$ | $\tau_{\text{int}}$ | $n_k$   | Element positions of the $k^{th}$ subcolumn         |
| $\text{data}_k$ | $\tau$        | $n_k$   | Second subcolumn, element data                     |

The position columns are distinct, and the union of their support sets is the entire index set, i.e. $\bigcup_i \text{supp}(\text{pos}_i) = \{0, \ldots, n - 1\}$.
And if we only consider the index sets in this scheme, we have an index set partition representation scheme: \texttt{PartitionedIndexSet}_k.

\textit{Note.} Given our constraints, only \( k - 1 \) of the position columns are actually necessary. For the case of \( k = 2 \), dropping the redundant complementing position column results in the \texttt{Complementing-Subcolumns} scheme.

Having separate columns for each part in the partition becomes rather unwieldly with many parts, and essentially irrelevant when wishing to represent a number of parts proportional to the column length. We can do with something simpler than \( k \) subcolumns, when we consider set-theoretic definition of a partition: A function from the set of indices to the set of parts. And a column, after all, is merely an evaluated function from indices to the domain of the element type; hence:

\textsc{Partition} (Standard representation of a column index set partition)

An alias for the standard representation of a column, with the domain being a set of part identifiers.

A representation of a partition in the \textsc{Partition} scheme is said to be \textit{canonical} if its image (as a function) is \( \{0, \ldots, |\text{Rng}(k)| - 1\} \) for some \( k \in \mathbb{N} \). With a canonical representation, there are no implicitly-empty parts.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{chart.png}
\caption{Materialization of an index set partition, from the standard scheme (\textsc{Partition}) into the \textsc{Partitioned}_k scheme.}
\end{figure}

\textit{Note.} The circuit in Figure 2.13, when adapted for \( k = 2 \), transforms a \texttt{DenseIndexSet} instance (after a \( \tau_{\text{bit}} \) to \( \tau_{\text{int}} \) upcast) into an index subset in the default representation, along with its complement.

\subsection*{2.4.3 Element decomposition, interleaving, and type-punning}

More generally, let \( \vec{\tau} = (\tau_1, \ldots, \tau_k) \); any injective partial function \( f : \vec{\tau} \rightarrow \tau_c \) may be considered as a method of ‘losslessly’ composing elements of the types of \( \vec{\tau} \) into a element of type \( \vec{\tau} \) — and
thus as a representation of the element into which they compose. If $f$ is also surjective, it defines a representation scheme:

**Components**

The compressed form of a column $\text{col}$ is its image under $f^{-1}$, i.e. $k$ columns of types $(\tau_1, \ldots, \tau_k)$, each of length $|\text{col}|$. Decompression is performed by applying the $\text{Elementwise}_f$ operator (see Subsection 1.1.2).

Despite not having constrained $f$ generally, we would typically think of the concatenation of the elements of a product type, the data members of a “struct” or “class” (in a C/C++/Java-like programming language), or of the bytes or bits of a wider numeric type. Particularly interesting is the case of concatenating identical types, i.e. the case of $\tau_i = \tau_d$ for all $i \in \{1, \ldots, k\}$: When the types are distinct, we have to keep them in separate columns; but if $f$ concatenates multiple values of the same type, it makes sense to just have them all in the same column, used as input for the decoding function. Below are two alternatives for doing just that:

**ComposeSegments**

| In/out? | Label        | Type  | Length                  | Description                        |
|---------|--------------|-------|-------------------------|------------------------------------|
| Input   | segment_length | $\tau_{\text{int}}$ | 1                         |                                    |
| Input   | components    | $\tau_d$ | $k \cdot \text{segment_length}$ | Element-index-major                |
| Output  | composed      | $\tau_c$ | segment_length          |                                    |

In a $\text{segment_length}$-segmented view of the input, the $k$ columns ($k$ segments) undergo the equivalent of $\text{Elementwise}_f$, i.e. the output satisfies $\text{composed}[i] = f(\text{components}[1,i], \ldots, \text{components}[k,i])$.

**Assemble**

| In/out? | Label        | Type  | Length                  | Description                        |
|---------|--------------|-------|-------------------------|------------------------------------|
| Input   | segment_length | $\tau_{\text{int}}$ | 1                         |                                    |
| Input   | components    | $\tau_d$ | $n \cdot \text{segment_length}$ | Component-index-major              |
| Output  | composed      | $\tau_c$ | $n$                     |                                    |

In a $\text{segment_length}$-segmented view of the input, each of the segments has exactly the number of elements for applying $f$ to their values. The output is 1 element per every input segment, and satisfies $\text{composed}[i] = f(\text{components}[i,1], \ldots, \text{components}[i,k])$.

These operators both have a two-dimensional input, representing several distinct columns; but while $\text{ComposeSegments}_f$ applies the composing function $f$ to elements at strides of $\text{segment_length}$ (i.e. it expects a concatenation of the columns), $\text{Assemble}$ applies $f$ to consecutive elements of the $\text{components}$ column (i.e. it expects the columns to be interleaved). Thus each of these operators can be obtained from the other by applying it after a $\text{Transpose}$.
Figure 2.14: Difference between the Assemble and ComposeSegments operators, with the elementwise composition function $f$ being simple addition.

The two operators translate into two compression schemes — two specializationbs of Components for the case of a uniform elementwise function argument type:

**ConcatenatedComponents**
A segmented column decompressible by ComposeSegments; the input column length must be divisible by the segment length, i.e. the segmented view of the must be a full matrix.

**Shattered**
A segmented decompressible by Assemble; the input column length must be divisible by the segment length.

### 2.4.3.1 Bitwise decomposition

The “maximum-resolution” case of ConcatenatedComponents is breaking up a column of a $w$-wide element type into $w$ 1-bit-wide component columns. This approach has seen some research attention under the name of “bitwise-decomposed storage”. Specifically, when I/O savings are of utmost importance, one may want to go ahead with some computational work even with just the highest bits of a column of values (e.g. for CPU-GPU co-execution of queries, as explored in [PMK14]).

Another kind of decomposition into bits — less orthogonal and space-frugal — is per-value indicator bit columns. For example, suppose $\tau_c$ is a type with domain $\{a, b, c, d, e\}$ instead of using 3 one-bit component columns (with which values will correspond to $000, 001, 010, 011, 100$), we would use 5 one-bit indicator columns, such that for every index $i$, exactly one of them is set (i.e. $00001, 00010, 00100, 01000, 10000$).

| ValueIndicators |
|------------------|
| **Label** | **Type** | **Length** | **Description** |
| domain_size | $\tau_{int}$ | 1 | 
| bitmaps | $\tau_{bit}$ | $n \cdot$ domain_size |

$num$ bits concatenated columns (via a segmented view); the $j^{th}$ $n$-bit column is a dense subset representation of those indices within $\{0, \ldots, n-1\}$ having value $n-1$. (Inaccurate but common shorthand name: BITMAP; also referred to as “Bit-vector encoding” [ABH+13, 422].)

While this representation scheme is exponentially inefficient, it has the advantage of offering what are essentially pre-computed results for constant-column equality operator applications, and requiring access to only a small part of the column when only a small subset of values are of interest.
2.4.3.2 Type punning and member access

The bitwise decomposition described in Subsubsection 2.4.3.1 is a specific example of using the SHATTERED for reinterpreting, rather than transforming, data. In our computational model, there is no “reinterpretation”, nor access to a part-of-an-element — a column is never replaced, reconceived, or trimmed of uninteresting data; but in actual programming, a single value of an aggregate or product type may be perceived as the sequence of its constituent types, in order of their appearance in physical memory. For example: a complex number could be its real component followed by its imaginary components, separately). Descending a level further, non-aggregate numeric values may be perceived as sequences of bytes, or bits, in accordance to the idiosyncracies of their representation in computer memory. This practice is known as type punning; applying the SHATTERED scheme compression with an appropriate function on an input column produces the equivalent of such punning (and will likely result in actual punning in an implementation of the computational model).

2.4.4 Variable-width columns and string representation

Our basic definition of a column (Definition 1.1) was chosen for its simplicity, which in turn simplifies the formalism for our computational model. An important aspect of this choice is the limitation of columns to have a fixed width, i.e. element types with a fixed-size representation. This ensures the memory representation of a column is perfectly straightforward, and any elements is accessible with no read-after-read dependencies. A column store must, however, also support variable-width data — so as to hold columns of strings, arbitrary-precision numbers or just opaque sequences of octets. Variable-width data also features extensively in many CPU-targeted lightweight compression schemes (see, e.g. [DHHL17]); most of which are usable or in-use in column stores, further motivating consideration. To do so, we will not amend the definition of a column, but rather make a separate one:

Definition 2.21. A variable-width column is a function \( \text{vwc} : \{0, \ldots, n-1\} \rightarrow \text{Dom}\left(\bigcup_{k \in \mathbb{N}} \prod_{j=0}^{k} \tau\right) \) (where the empty product of \( \tau \)'s is \( \tau_{\text{unit}} \)) for some fixed-representation-size element type \( \tau \) and non-negative integer \( n \). \( n \) is the column’s length, and \( \tau \) is its base element type. The length (in units of \( \tau \)) of the \( i \)th element of \( \text{vwc} \), also referred to as the width of \( \text{vwc} \) at index \( i \), is the value \( k \) such that \( f(i) \in \text{Dom}\left(\prod_{j=0}^{k} \tau\right) \).

A variable-width column is not a column by our definition (Definition 1.1); it may only accidentally constitute a column if it so happens that all of its elements have the same length; it will therefore not be decodable into the standard column representation, nor could we encode it in other column representation schemes. The concatenation of the \( \tau \)-units of all column elements, however, is a plain column; and the additional information it needs is the association of contiguous subsequences of \( \tau \) elements with their corresponding variable-width element indices. That is exactly a partition into segments, discussed in Subsection 2.4.2; and our representation scheme will combine a SEGMENTATION form with the raw data:

(Standard representation of a variable-width column)
**Label** | **Type** | **Length** | **Description**
--- | --- | --- | ---
start_position | $\tau_{\text{int}}$ | $n$ | offsets into data
length | $\tau_{\text{int}}$ | $n$ | ... of the sequences of $\tau$ elements for each element of the variable-width column; of non-negative value.
data | $\tau$ | ? | individual $\tau$ values whose concatenations make up the different elements of the variable-width columns

**start_position** and **length** correspond to **start** and **length**, respectively, in scheme SEGMENTATION. For an index $i$, the $i^{\text{th}}$ element of the variable-width column occupies the range of $\tau$ elements $\text{data}[i], \ldots, \text{data}[i + \text{lengths}[i] - 1]$ (and specifically, occupies nothing if $\text{lengths}[i] = 0$).

This “verbose” choice of standard representation has more to it than meets the eye, as we have not constrained **start_positions** to even be monotone increasing. When it also holds that $\text{start_positions}[i] = \sum_{j<i} \text{lengths}[j]$, things are much simpler, and we refer to this case as a canonical representation of a variable-width column.

A special case of variable-width columns are those columns whose width is not fixed, but is at least bounded. In the SQL standard, \textsc{varchar(k)} columns are of this kind: strings with a per-column maximum length. These columns have a more straightforward, though wasteful, representation if we expand all elements to actually take up that maximum width:

**CappedWidth**

**Label** | **Type** | **Length** | **Description**
--- | --- | --- | ---
max_length | $\tau_{\text{int}}$ | 1 | Cap on single element length in units of $\tau$
lengths | $\tau_{\text{int}}$ | $n$ | ... of the sequence of $\tau$ elements for each element of the variable-width column
data | $\tau$ | $n \cdot \max\_length$ | Sequences of $\tau$ elements for each element of the variable-width column, and arbitrary filler values.

Considering a **max_length**-segmented view of **data**, each column is a small buffer holding a single variable-size element, and **lengths** indicates how much of the buffer is actually taken up by the element; the few remaining type-$\tau$ elements may have any value and are ignored.

**Note.** The above is insufficient to fully model the representation of strings, as it does not go into the character set encoding — which often introduce another layer of variability-of-length: Each single character within a string can have variable length (e.g. when the popular UTF-8 encoding is used). While this work covers composite compression schemes in Chapter 3, String representation is not explored further.

### 2.4.5 Nullable and union-type columns

In SQL tables, a column may be allowed to have NULL values at some positions rather than a value of the actual type. This can be interpreted as expanding the column type $\tau$ by a single value, i.e. the replacement of $\tau$ by the union type $\tau \cup \tau_{\text{unit}}$ (with $\tau_{\text{unit}}$ having a single possible value); and this interpretation motivates considering the case of NULL’s as a special-case of disjoint-union types: $\tau_1 \cup \tau_2$. How would we represent a column with such an element type?

Three approaches come to mind:

1. Actually expand the representation of $\tau_1$ values to accomodate all $\tau_2$ values as well; this often means increasing column elements’ width in bits.

41
2. Hold two complementary subcolumns, one of $\tau_1$’s and the other of $\tau_2$’s.

3. Hold a column of just one of the types (without loss of generality, $\tau_1$), and an overlay subcolumn of elements of the other type, which are to be \textit{patched} into the base column.

The first approach is typically the most burdensome: In addition to the width increase, one must also pay up-front the price of extra condition checks and reinterpretation work to obtain a usable $\tau_1$ or $\tau_2$ from the combined domain. However, by definition, this is our standard approach: Ignoring how the type was formed and yielding a standard representation (in the sense of Subsection 1.1.1). The second and third approaches correspond, respectively, to the \textsc{ComplementingSubcolumns} and to \textsc{Patched}.

Now let us consider these approaches for the specific case of NULL’s. The first approach isn’t often used in DBMSes in practice (in part due to the shortcomings listed above). Some DBMSes do, however, artificially restrict many of their data types to ensure at least one “invalid” value, which is then used to represent NULL. Two concrete examples from MonetDB, which takes this approach [Theb]: In the case of fixed-width integer types, the lowest or highest representable value is sacrificed to serve as the encoding of NULL; and in a UTF-8 encoded string, one of the invalid sequence of octets is used (as not any sequence of octets is a valid encoded UTF-8 string).

Using the second approach for NULL’s, we would compose \textsc{ComplementingSubcolumns} with the \textsc{Constant} scheme used for the $\tau_2 = \tau_{\text{unit}}$ subcolumn, i.e. keep a second proper column only in principle: We would only need one actual column for the (consecutive) non-NULL values, and one column being the standard representation of the binary choice between NULL and the non-NULL value for each position.

Finally, the third option would also require just two columns due to the use of \textsc{Constant}. In this case, and for a column of basic type $\tau$, of length $n$ and with $m$ NULL values, the overhead over a length-$n$ non-nullable column would be $m \cdot |\tau_{\text{int}}|$, or $m/n$ in relative terms (and ignoring any further compression) — quite moderate.
Column compression

We have so far modeled columnar computations; and have used them to represent several fundamental combinatorial structures in columnar schemes. This is sufficient machinery for us to turn our attention to encoding schemes effecting data compression. The following sections are not intended to be an exhaustive re-formulation of existing academic work on compression schemes for DB columns; they do, however, cover the lightweight compression schemes most commonly studied academically and used in practice (see [ABH+13, §4.2] for a brief survey), formulating them in columnar terms. Other schemes developed below result from the application of the same principles and approach as in previous chapters, as well as the composition of existing schemes — and have mostly not been formulated in previously published research.

Definition 3.1 (Compression scheme). A column compression scheme is a codec \((f_e, f_d)\) for single columns, as per Definition 2.3. Respectively, a concrete column compression scheme is a concrete codec \((C_e, C_d, C_v)\) for single columns.

Column compression schemes are often not representation schemes for columns, in that only a subset of all possible columns is representable by a compressed form in a given scheme (other columns are “not compressible” by the scheme).

Definition 3.2 (Representation size). The size (in bytes) of a columnar representation of a column or other structure is the sum of the constituent columns’ sizes in bytes, which in turn is their element type’s fixed width in bytes, times the column length.

Definition 3.3 (Compression ratio). The compression ratio of a compressed form \(\text{col}'\) under a concrete compression scheme \((C_c, C_d)\) is the ratio of sizes of \(\varphi_d(\text{col}')\) and \(\text{col}'\) in bytes. The compression ratio of an uncompressed column \(\text{col}\) under scheme \((C_c, C_d)\) is the compression ratio of its compressed form \(\varphi_c(\text{col})\) (which is the ratio of sizes of \(\text{col}\) and \(\varphi_c(\text{col})\) in bytes).

3.1 Basic schemes

We begin with a formulation of relatively simple compression schemes, each exploiting a single aspect of redundancy or excess in the representation of the input: Predictability of values using the element index; or a support set much smaller than \(\text{Dom}(\tau)\). These schemes are also characterized by decompressed elements directly depending on a single element of the compressed column, at the most — so that the basic schemes are also the “most parallelizable”.

43
3.1.1 Output-index-based compression

Our first compression scheme applies in cases where the entire input column is simply an evaluation of a known function:

**GENERATED**

| Label    | Type | Length | Description                  |
|----------|------|--------|------------------------------|
| length   | τ    | 1      | ... of the represented column |

The decompressed column is the evaluation of $f$ over domain $\{0, \ldots, \text{length}[0] - 1\}$ (i.e. the result of applying the composition of `Elementwise` after `iota` to the compressed input).

Of course, there exists only one such column for each choice of $f$ at each length, which means such a scheme will not often be directly used in practice; still, auto-generated columns do occur in real-life databases (e.g. matrix row and column indices based on the overall record index). To slightly expand the expressivity beyond this scheme, we introduce some parameters to choose $f$ within a larger (though fixed-dimensional) space. For example, let it be just a single parameter for selecting among the space of uniform-valued functions from $\mathbb{N}$ to $\text{Dom}(\tau)$:

**CONSTANT**

| Label    | Type | Length | Description                  |
|----------|------|--------|------------------------------|
| value    | τ    | 1      | the single uniform value of all column elements |
| length   | τ    | 1      | ... of the represented column |

The decompressed column has `length` elements, all equal to `value`.

This is, in fact, more than a toy example; it is quite the useful plug-in for more complex compression schemes — where it might stand in for a full-blown component column. Examples of such schemes can be found in Subsection 2.4.5 and Subsection 3.4.1 below. It may also be used as the default representation scheme for newly-created non-NULL fixed-width columns, so as to avoid the work of actually initialize them.

For columns with a ring-like element type, admitting addition and multiplication (e.g. the integral types), the `CONSTANT` can be seen as a semi-degenerate of larger spaces of functions: The single value `value` is a scalar coefficient, for a single basis function $x \mapsto 1$. If we allow $k$ functions, rather than just one, as part of the definition of the scheme, we have:

**GENERATED**

| Label     | Type | Length | Description                  |
|-----------|------|--------|------------------------------|
| length    | τ    | 1      | ... of the intended column   |
| coefficients | τ | $k$   | coefficients of the linear combination of $\vec{f}$ forming the generating function |

The scheme is parameterized on a sequence of $k$ basis functions, $\vec{f} = (f_1, \ldots, f_k)$, so that the decompressed column is an evaluation of $\tilde{f} = \sum_{i=1}^{k} \text{coefficients}[i-1] \cdot f_i$ over the domain $\{0, \ldots, n - 1\}$.

**GENERATED Polynomial**

The `GENERATED` scheme, with the choice of increasing-power monomials for $\vec{f}$, i.e. $f_i(x) = x^{i-1}$. 

44
Of course, real-life data does not match a generated function perfectly — as there are almost always some discrepancies (see Subsection 3.2.4) or what could be described as noise (see Subsection 3.2.1).

### 3.1.2 Column support compaction

The support of a column may only constitute a small subset of the domain of its element type, motivating compression schemes which use less space to indicate which element of that support set is used. We consider an initial, simplistic taxonomy of small support sets, by order of increasing complexity.

1. Singleton set → **Constant**

2. Set having a low maximum within $\tau$ with respect to its physical (binary) representation, containing most values below that maximum → **DiscardHighBits**

3. Set with none of the above features (scattered in a non-trivial fashion within $\tau$) → **Dictionary**

**Singleton support sets** Single-value columns admit the **Constant** compression scheme from Subsection 3.1.1 above. **Constant** is a degenerate case of a small support set, requiring no per-element compressed data.

**Dense low-bounded support sets** A database column will often be declared to hold integers of some default (larger) width, while in fact its data could be represented with a lower width. This makes the top bits in the representation always-zero (or in the case of signed integers, a sign-extension). If it also happens that within the non-redundant bits, the column’s support set is ‘dense’ — containing more than half of the possible values — it does not make sense to try to represent an element of the support set other than by its lower bits. Generalizing from the case of unsigned integers, we define:

**DiscardHighBits**

Let $\tau, \tau'$ be types, $\preceq$ be the natural well-ordering of $\text{Dom}(\tau)$ and $b$ a positive integer, such that $\text{Dom}(\tau') \subsetneq \text{Dom}(\tau)$ and that $\max\{\text{Dom}(\tau')\}$ is lower by $\preceq$ than the $2^b$-th lowest element in $\text{Dom}(\tau)$. The compression by **DiscardHighBits** is the mapping of each element of type $\tau$ to itself, but as an element of type $\tau'$; and the decompression is the opposite. In “physical” terms, this is typically the dropping the high bits in the representation of each element. Abusing the above definition somewhat, we also refer to the discarding of high bits of signed-integer types as encoding with this scheme.

**NullSuppression**

A commonly used alternative name for **DiscardHighBits**; the scheme does not relate to the SQL **NULL** value. (Shorthand name: **NS**)
Explicitly-enumerated support sets  The support set may sometimes be small, but not limited mostly to the lower elements of an ordered type. While it may still fit some pattern or rule, it is often simple and effective to specify all elements of the set, using an evaluated function whose image covers the entire set. This is (inaccurately) referred to as a dictionary for the column, used in an eponymous compression scheme:

**Dictionary**

| Label   | Type   | Length | Description               |
|---------|--------|--------|---------------------------|
| dictionary | \( \tau \) | \( d \) | ... into dictionary       |
| indices     | \( \tau_{\text{int}} \) | \( n \) |                            |

Letting \( \text{col} \) denote the uncompressed column, \( \text{Img}(\text{dictionary}) \) is a superset of \( \text{Img}(\text{col}) \). The compressed form is the index in the evaluated dictionary instead of the original value, so that \( \text{dictionary} \circ \text{indices} = \text{col} \) (with \( \circ \) denoting function composition), and elementwise we have \( \text{col}[i] = \text{dictionary}[\text{indices}[i]] \).

Decompressing the **Dictionary** scheme involves performing essentially the opposite of the Scatter operator (see Subsection 2.2.3):

**Gather**

| In/out? | Label | Type   | Length | Description           |
|---------|-------|--------|--------|-----------------------|
| Input   | pos   | \( \tau_{\text{int}} \) | \( n_2 \) | positions in data    |
| Input   | data  | \( \tau \) | \( n_2 \) |                      |
| Output  | result | \( \tau \) | \( n_1 \) |                      |

Performs an indirect lookup into \( \text{data} \), i.e. gathers various elements of \( \text{data} \), so that \( \text{result}[i] = \text{data}[\text{pos}[i]] \).

### 3.1.2.1 De-duplication and use of dictionary indices as surrogates

With a compression scheme focused only on the support set, it is natural to wish to use the compressed-form elements (the representations of individual uncompressed elements) as surrogates for the original column elements: Avoid decompression, and apply columnar operators to the compressed form of the data. This is possible if the compression function is injective, or better still, when the order relation within the compressed domain commutes with the decompression function; and key operators we can apply to the compressed surrogates are:

- Equality and range filters, using a compression of a single match value or a pair of extrema to compare against;
- **Sort**, **Partition** or the obtaining of the support set for a column;
- **Join** of a pair of columns which share the same compression scheme.

This potential for operator “push-down” via surrogates was noticed already in early research into the use of compression in DBMSes [GS91, Page 4].

Use of surrogates is clearly possible with **DISCARDHIGHBITS**; but the **Dictionary** scheme does not guarantee, by definition, usable surrogates. To secure this guarantee we must impose additional constraints on the **dictionary** column of the scheme:
1. **Injectivity**: For surrogate use of the compressed elements in *exact-match comparisons*, $\text{dictionary}^{-1}$ must be a proper function, i.e. all dictionary entries must be distinct.

2. **(Strong) monotonicity**: For surrogate use of the compressed elements in *order relation comparisons*, $\text{dictionary}$ must strictly preserve that order, i.e. $i < j$ iff $\text{dictionary}[i] < \text{dictionary}[j]$. (Note that if $\text{dictionary}$ is known to be injective, this is equivalent to the additional constraint of weak monotonicity.)

By adding one or both of these constraints to the definition of $\text{Dictionary}$, we obtain a different — more restricted — compression scheme:

**UniqueDictionary**

A sub-scheme of $\text{Dictionary}$, where compressed forms have no duplicate elements in the $\text{dictionary}$ column. (Shorthand name: UDICT.) With this scheme, $\text{indices} = \text{dictionary}^{-1} \circ \text{col}$ (with $\circ$ denoting the composition of functions).

**MonotoneDictionary**

A sub-scheme of UniqueDictionary, where the $\text{dictionary}$ column is monotone-increasing with respect to an order relation $\preceq$ over $\text{supp(\text{dictionary})}$; we omit $\preceq$ if it is the default order relation for the uncompressed column’s element type. (Shorthand name: MDICT.)

De-duplication and sorting change the cost-benefit tradeoff of the unconstrained Dictionary scheme: We gain the use of surrogates, described above — at the cost of additional, less-parallelizable work in performing the compression. But there may be additional, secondary gains in using UDICT or MDICT: When a unconstrained dictionary contains many duplicate entries, their removal reduces its size. This, in turn, may allow for a smaller width of the dictionary indices as well. The compression ratio may thus improve somewhat; and a smaller dictionary is also likely to fit more of itself into a processor’s cache or local memory space. Also, a sorted dictionary is more likely to admit additional compression utilizing the similarity of consecutive elements (see Sections 3.2 and 3.6.1 below).

Finally, note that the choice of indices in a dictionary can be used to express an order relation among values of the encoded column, other than the default order on the uncompressed column’s element type. For example, the dictionary entries could be ordered by descending frequency within the uncompressed column, or even so that the dictionary index is a hash of the value (see also Section 3.3).

### 3.2 Patterns of scheme composition and transformation

A reader familiar with existing analytic DBMSes (columnar or otherwise), may be puzzled by the previous section: Very few schemes were presented as “basic”; most of them cannot even encode an arbitrary input column (regardless of compression ratio); and several schemes which, in other systems, are not considered ‘composite’, were not mentioned (such as RLE and FOR). The reason for this is that our computational model, together with the machinery for columnar representations of data, allow us to formulate all those additional schemes by composing the basic ones, or transforming them using a few additional operations in the encoding and decoding circuits.
Each such “composite compression scheme” is simply a codec, with the encoding and decoding partial-functions being compositions of several functions. A composite scheme may be concretized by composing the decoding and encoding circuits of its constituent schemes: Assigning outputs of one decoder to inputs of the other, and assigning encoder outputs in the opposite direction; or, in some cases, taking the union of these circuits; both of these transformations were described in Section 1.3 above.

One can compose compression schemes almost to no end, choosing schemes using knowledge of the intended input distributions. This section will describe some patterns for composing schemes — which will be put to use in concrete composite schemes of particular interest, throughout the rest of this chapter. Each of the patterns presented here exemplifies some concept worthy of explicit description — and are commonly useful in terms of compression ratio gains.

### 3.2.1 Elementwise composition and “noise-signal”-like separation

In signal processing, a fundamental task is restoring or isolating an original transmitted signal from (literal) noise it picks up as it is transmitted over physical media. The signal carries limited and well-formed information; the noise is unconstrained with the encoding or modulation of the signal, but is of limited power. In our setting, there is ‘noise’ per se — information can discard; but the analogy still inspires a decomposition of an unencoded column: \( \text{col}[i] = \text{signal}[i] + \text{noise}[i] \), with \text{signal} admitting a more effective compression (typically better than, say, \text{DiscardHighBits}), and \text{noise} being of limited magnitude. These can be compressed separately, with \text{noise} using \text{DiscardHighBits}.

The trivial — though useful — example of this kind of scheme would follow through with the analogy to signal processing:

| Label      | Type | Length | Description                          |
|------------|------|--------|--------------------------------------|
| coefficients | \( \tau \) | \( k \) | coefficients of the linear combination of \( \vec{f} \) forming the generating function |
| noise      | \( \tau_{\text{noise}} \) | \( n \) | elementwise additive noise           |

The type \( \tau_{\text{noise}} \) is a subtype of \( \tau \), typically with fewer bits in its representation. The decompressed column is the addition of the noise, up-cast from \( \tau_{\text{noise}} \) to \( \tau \), to the generated function.

The more general scheme above scheme may well be used in practice to store samples of some real-world signal, with signal-processing techniques employed to determine the basis coefficients.

Generalizing the ‘signal-noise’ decomposition, elementwise addition may combine arbitrary schemes. Specifically, \text{GENERATED} with \( k \) base functions can be perceived as the application of \( k - 1 \) compositions-by-addition of \text{GENERATED}_{\vec{f}} \) schemes for the different basis functions.

\textit{Note.} While lossy compression of information is out of scope for this document — \text{NOISYGENERATED}_{\vec{f}} \) schemes could be used for lossy compression floating-point data, which is of inherently limited accuracy: The \text{noise} values might be encoded at lower precision, seeing how after their addition to \text{signal}, those less-significant bits may fall below the floating point precision threshold.
3.2.2 Discrete differentiation and integration

Column stores processing queries, and performing decompression, frequently need to consider both the positions (or indices into a column) of elements and lengths or distances between elements (within a column), or to switch between these two kinds of information. When compressing, we can make do with only one kind: We may store the positions and take their derivative as necessary, or store only lengths, and integrating over them. To be more concrete, circuits involving such compressed data will use one of the following operators:

### Derivative

| In/out? | Label       | Type | Length | Description          |
|---------|-------------|------|--------|----------------------|
| Input   | col         | τ    | n      |                      |
| Output  | differences | τ    | n − 1  |                      |

Sets \( \text{differences}[i] \) to \( \text{col}[i + 1] - \text{col}[i] \) — the discrete derivative.

### PrefixSum

| In/out? | Label   | Type | Length | Description          |
|---------|---------|------|--------|----------------------|
| Input   | data    | τ    | n      |                      |
| Output  | sums    | τ    | n      |                      |

Sets \( \text{sums}[i] \) to \( \sum_{j=0}^{i} \text{data}[j] \). (a.k.a. Integral, InclusivePrefixSum)

### ExclusivePrefixSum

| In/out? | Label   | Type | Length | Description          |
|---------|---------|------|--------|----------------------|
| Input   | data    | τ    | n      |                      |
| Output  | sums    | τ    | n      |                      |

Sets \( \text{sums}[i] \) to \( \sum_{j=0}^{i-1} \text{data}[j] \). Identical to PrefixSum, except that the upper limit of summation is not used.

3.2.3 Segmentization

In Section 2.3 we formalized representations for segmented columns, and the process of segmentation of operators. Now, given some compression scheme for columns — it can be applied separately to consecutive segments of an (uncompressed) column, with the compressed forms’ constituent columns for every segment being concatenated, to form a compressed form for the column overall. Doing so is applying a segmentized version of the original scheme. Segmentized schemes are of two varieties, depending on the nature of the segmentation:

**Fixed-length segments** As mentioned in Section 2.3, segmented columns with fixed segment length are prevalent in many column stores regardless of compression; and they therefore often choose to compress distinct segments separately (possibly with some column-level data) — to avoid inter-segment dependencies during decompression. Provided segments are not too short, this may help compression ratio: The overhead on per-segment data is often lower than the savings due to localization: A smaller range of variation to cover; and no need to tolerate or account for anomalies throughout the entire column.
Variable-length segments  Letting segment lengths vary is a costly choice, regardless of the compression scheme. First, there is the space necessary to specify segment lengths. There’s also a cost in parallel decompression time, due to read-after-read dependencies: A decompressed column element’s segment index cannot be determined in advance (and, in fact, often requires multiple reads into a SEGMENTATION); and only when it is known can the per-segment data be read. This choice is therefore the result of necessity rather than convenience:

- The column data itself may require adapting segment length to capture abrupt local changes within a column. Alternatively, the data may exhibit gradual, but non-uniformly-progressing, changes in behavior; in this case the break-up into segments would be rather artificial, but required for limiting divergence between the actual data and some localized model. As an example, consider the columns obtained by the following random process: Elements are uniformly sampled from a small-support local distribution, and this distribution changes at a non-uniform pace).

- In some cases, it is the decompression process which necessitates variable-length segmentation. We observed in Subsection 2.3.3 that segmentized operators do not preserve uniformity: The dependence of operator output length on the input data causes uniformly-segmented data to be transformed into non-uniform-length output segments. Thus, even if a variable-length segmentation is not immediately apparent in a description of the compressed form, a decoder may have to introduce it within a decoding circuit.

3.2.4 Patching and outlier removal

Definition 3.4 (Hamming distance [Ham50]). Let $\text{col}_1, \text{col}_2$ be columns of type $\tau$ and length $n$. The Hamming distance between $\text{col}_1$ and $\text{col}_2$ is the number of indices on which $\text{col}_1$ and $\text{col}_2$ disagree, i.e. $\left|\{i \in \{0, \ldots, n-1\} | \text{col}_1(i) \neq \text{col}_2(i)\}\right|$. The Hamming distance between a column $\text{col}$ and a set $S$ of columns of the same length is the infimum of distances between $\text{col}$ and every column in $S$.

Suppose some compression scheme $\text{MainScheme}$ (that is, some codec $(f_d, f_e)$) has certain desirable features, such as high compression ratio or well-parallelizable decompression. Now suppose we wish to compress a column $\text{col}$ which is not in $\text{Dom}(f_e)$; rather, it is within a small Hamming distance from it: If not for a few outlier values, the column could be compressed using $\text{MainScheme}$. Let $\text{col}'$ denote a column in $\text{Dom}(f_e)$, closest to $\text{col}$; instead of giving up on the efficient compression of $\text{col}$, it is intuitive to consider using the compressed representation of $\text{col}'$ instead, with some information allowing for post-decompression correction of $\text{col}'$ into $\text{col}$. The necessary information is in fact simply the subcolumn of $\text{col}$ on $\{i \in \{0, \ldots, n-1\} | \text{col}(i) \neq \text{col}'(i)\}$, which we adjoin in the standard representation (see Definition 2.6). The application of this subcolumn — the patching of $\text{col}'$ with $\text{col}$ values — is an application of the SubcolumnOverlay operation; and we have thus obtained a new compression scheme, being the composition of PATCHED (defined in Subsection 2.2.3) after $\text{MainScheme}$ on the main column.
Figure 3.1: An example of patching with standard-representation-coded inputs. The column contents is the phrase “the eels and the foxes” in Swedish, without spaces; most character UCS-2 codes fit into octe, while the patches use the full 16-bit width. The choice of 8-bit characters later overridden by patch values is arbitrary.

Figure 3.2: A decoder for patched columns, where most of the data is encoded with scheme `MainScheme` and the positions of outliers encoded with `PatchPositionsScheme` respectively.

The decoding example is immediately reminiscent the use of the Patched scheme for representing NULL values, in Subsection 2.4.5 Indeed, if the column element type is $\tau = \tau_1 \cup \tau_2$, and there are only a few elements from $\text{Dom}(\tau_2)$, the patches can have type $\tau_2$ and the column overall can be compressed as a purely $\tau_1$-type column.

The use of patched compression schemes in column stores was pioneered by Vectorwise, as presented in [ZHN06]. However, work published so far on this subject (as surveyed [DHHL17]) does not present patching as a meta-scheme, or a scheme transformation, applicable generically to arbitrary base schemes; rather, very specific patched schemes are presented. This stems from the orientation of existing work towards less-parallelized CPU-based decompression: These schemes “package” data from both the main column and the patches subcolumn into consecutive chunks of the (not immediately columnar) compressed form — so that it fits well into a few consecutive CPU cache lines. This is done with some finesse in carefully designing compressed block layout — which makes for idiosyncracy rather than generality. See a more detailed discussion of this point in [RB17].

Patching can increase compression efficiency not only by reaching the domain of a compression
scheme: It also allows selection of better-compressible column within a compression scheme’s domain. For example, suppose the RLE scheme is applied to a column with very long stretches of similar lengths. But suppose also that at a low frequency within those stretches, an interrupting value appears once or twice. The distance between these interruptions has high variance, making it difficult to estimate the index of the run covering a given position in the output; also, the interruptors take up close to half of the RLE compressed form in terms of bytes used. Patching away a single one of these interrupting values saves the space of two values and two run lengths in the main RLE compressed form, but costs one-or-two values and one-or-two position indices in patch information; this is not much of a saving, if at all (especially if the type for position indices is wider than the type of run lengths). However, the overall length of the compressed main column has now dropped significantly — as most runs had only existed separately because of the interruptors — and the variances in run lengths have also decreased. This may allow for faster decompression of the column overall, as well as faster processing of the compressed form: Computing, say, a sum of all column values may become faster due to the smaller number of runs, while the re-integration of the sum of patch values and the sum of the main column will have very low overhead.

Note. As part of a later discussion of composite compression schemes, a converse-of sorts of column patching will be presented in Subsection 3.5.3.

3.2.5 Alternating schemes within the same column

This section is concerned with the variation in the distribution and patterns of data within a single column to be compressed. We have already touched upon several approaches to handle such variation, under certain constraints: The use of (low-complexity) models; separate handling of outliers by patching; and the localization of the compression scheme’s parameters, by segmentatizing the scheme. But what if significant fractions of the column behave so differently, that forcing the same compression scheme on the entire column is extremely costly? Alternatively, what if two possible schemes are each better fit at different parts of the column, so that while none of them does poorly overall, each of them could do really well if it could just be limited to that part? Indeed, it is intuitively useful to be able to alternative between two (or more) different schemes in different parts of the column.

In existing column stores supporting compression, this is achieved as a side-effect of column segmentation. In SybaseIQ, for example, each page in magnetic disk storage can may have data in a different compression scheme [ZHNB06, §2.1]. In Vectorwise, there are (at least) two levels of segmentation — one spanning a certain number of secondary storage pages (“chunks”; a compression scheme is chosen at this resolution) and within each chunks, multiple segments which may either be compressed or uncompressed [ZHNB06, §3.1.1].

It so happens, that this capability is inherent already in the machinery developed so far in this and the previous chapters, for representation and compression schemes: For each of the compression schemes to be alternated, the elements of the column compressed using this scheme constitute a subcolumn; and these subcolumns constitute a partition of the full column. We have already discussed the representation of partitions, in Subsection 2.4.2; one simply has to adjoin the PARTITION representation scheme with the compressed forms for the concatenated elements of each of the compressed subcolumns.

Of course, the column representing the partition (holding compression scheme indices) may itself be compressible, reducing the overhead of scheme alternation. A specific case of interest is
alternation aligned with transitions between equi-sized pages or chunks on disk or other storage: In this common case, we the partitions are \( \ell \)-segmentation-respecting, for some appropriate \( \ell \). Given \( \ell \), and with \( k \) alternating compression schemes — instead of keeping a partition with \( \lceil \log(k) \rceil \) bits per each compressed column element, we would only require \( \lceil \lceil \log(k) \rceil / \ell \rceil \) bits per element to indicate which scheme is in use. For typical values of \( \ell \) in practice, this is rather negligible.

Using a single partition column is not always the best choice, especially for an \( \ell \)-segmentation-respecting partition. In the latter case, we may prefer to represent each of the subcolumns using \texttt{SEGMENTEDSUBCOLUMN} before each subcolumn’s specific compression scheme. While the space overhead is somewhat higher — \( \lceil \lceil \log(n/\ell) \rceil / \ell \rceil \) per element for a column of length \( n \) — this representation affords us immediate use of each subcolumn separately. Also, a concatenation of the different \texttt{segment_pos} columns will constitute a valid \texttt{SEGMENTATION} scheme instance.

Alternation using a \texttt{SEGMENTEDSUBCOLUMN}-based representation. Note at most one of the subcolumns has a non-empty slack segment.

Figure 3.3: Decoders for columns compressed with two schemes, \texttt{SCHEME-1} and \texttt{SCHEME-2}, alternating in an \( \ell \)-segmentation-respecting fashion.

Building up a rising pyramid of composite compression schemes, in the next several section we present schemes utilizing the composition patterns from Section 3.2, along with other compositions-of-composite schemes, exploring their uses, strengths and weaknesses.


### 3.3 Variable-width columns: Dictionaries and surrogates

In Subsection 3.1.2 above we explored the use of dictionaries for compressing individual elements independently, and how their use may be adapted so that dictionary indices are usable as surrogates for the entire value (Subsubsection 3.1.2.1). The benefits of achieving the same for variable-width columns are potentially even more dramatic: It is typical, for example, for a database to have long columns of strings, each of which is a word or a phrase taken from a small set; and users are notorious for performing Join’s on these columns.

Recall, that the standard representation of variable-width columns (defined in Subsection 2.4.4) already has a dictionary-like aspect: Access to element data for a given index in the column is indirect, going through a column of pointers / position indices into the raw data. If all elements of a variable-width column happen to have length 1, then its *lengths* column is redundant, and we in fact have a DICTIONARY-encoded regular column (with *start_positions* assuming the role of *indices* and *data* assuming the role of *dictionary*). It can thus be said that the standard representation variable-width columns *already incorporates* a dictionary, except that dictionary keys are (position, length) pairs rather than merely a position; and that no additional compression scheme need be defined.

If the support set is small enough that indicating the length of each element is too wasteful, the pair of columns *start_positions* and *lengths* may be compressed using the DICTIONARY scheme — with identical indices. The resulting compression scheme merits a proper named definition:

**VariableWidthDictionary**

| Label            | Type    | Length | Description                        |
|------------------|---------|--------|------------------------------------|
| *indices*        | τ_{int} | n      | enumeration indices of the original elements |
| *entry_start_positions* | τ_{int} | m      | offsets into *data* |
| *entry_lengths*  | τ_{int} | m      | variable-size “element” lengths in units of τ |
| *entry_data*     | τ       | \(\sum_{i=0}^{m-1} \text{lengths}[i]\) |

The \(i\)th element of the uncompressed (variable-width) column is the concatenation of the elements of *entry_data* at indices \(\text{entry_start_positions}[i]\) through \(\text{entry_start_positions}[i]+\text{entry_lengths}[i]−1\). (Shorthand name: VWDICT.)

---

54
Two final potential benefits of using dictionaries with variable-length data involve either the overlap or the spacing-out of element data within the data array.

Recall that there are no constraints — in either the standard representation or in VWDICT — on the positions within the raw data column of elements’ data; this is unlike in the canonical column representation, where data is a concatenation of the elements in order. Specifically, there is nothing preventing the stretches of individual \( \tau \) values for different elements of the variable-width column from overlapping each other, or even beginning at the same position: A variable-length element is determined also by its length. Thus with base data “abc”, we could have the dictionary consisting off “a”, “b”, “c”, “ab”, “bc”, “abc” and the empty string — 7 different elements. Overlap use still has a small price with VWDICT, where each such element costs another length value, but in the standard representation, it costs nothing.

Alternatively, we may want to use the dictionary indices to make searching for element more efficient. Suppose we wish to determine whether a column contains a certain variable-length value. If the column values are sorted (regardless of the compression or representation scheme), a binary search would work. If we were guaranteed the dictionary only holds extant values, and we were using the sorted sub-schemes of the default representation or of VWDICT, we could apply a binary search to the dictionary — which is likely much faster to do. But finally, suppose we have a reasonable hash function \( h : \bigcup_{k \in \mathbb{N}} \prod_{i=1}^{k} \tau \rightarrow \mathbb{Z} \); in this case, we could have the data for element \( \text{col}[i] \) appear in data starting at index \( h(\text{col}[i]) \) (ignoring collisions). If we then search for our

---

**Figure 3.4**: The VariableWidthDictionary compression scheme applied to the sequence of words "a", "fish", "and", "a", "cat", "and", "a", "fish", "and", "a", "cat". Note that in a realistic scenario, the indices column would be much longer than the dictionary columns.

With VWDICT defined, we now have a second alternative for a surrogate scheme for variable-width data. Either use the default representation’s (position, length) pairs, or just use VWDICT positions. And for each of these alternatives we can define the two sub-schemes in which the dictionary entries are unique and strictly-monotone respectively (see Subsubsection 3.1.2.1).

### 3.3.1 Creatively positioning variable-length element data

Two final potential benefits of using dictionaries with variable-length data involve either the overlap or the spacing-out of element data within the data array.

Recall that there are no constraints — in either the standard representation or in VWDICT — on the positions within the raw data column of elements’ data; this is unlike in the canonical column representation, where data is a concatenation of the elements in order. Specifically, there is nothing preventing the stretches of individual \( \tau \) values for different elements of the variable-width column from overlapping each other, or even beginning at the same position: A variable-length element is determined also by its length. Thus with base data “abc”, we could have the dictionary consisting off “a”, “b”, “c”, “ab”, “bc”, “abc” and the empty string — 7 different elements. Overlap use still has a small price with VWDICT, where each such element costs another length value, but in the standard representation, it costs nothing.

Alternatively, we may want to use the dictionary indices to make searching for element more efficient. Suppose we wish to determine whether a column contains a certain variable-length value. If the column values are sorted (regardless of the compression or representation scheme), a binary search would work. If we were guaranteed the dictionary only holds extant values, and we were using the sorted sub-schemes of the default representation or of VWDICT, we could apply a binary search to the dictionary — which is likely much faster to do. But finally, suppose we have a reasonable hash function \( h : \bigcup_{k \in \mathbb{N}} \prod_{i=1}^{k} \tau \rightarrow \mathbb{Z} \); in this case, we could have the data for element \( \text{col}[i] \) appear in data starting at index \( h(\text{col}[i]) \) (ignoring collisions). If we then search for our
arbitrary element at that exact position, the search would only require constant time (independently of the column length) on average [CLRS09, Chapter 11].

Of course, collisions can and do occur. One can construct a proper hash table to handle them (e.g. of the linear probing [Lit80] variety); but another option is to give up on hashing colliding elements, so that data is split into hashed_data and unhashed_data. MonetDB applies this approach to some extent for representing string columns: For string columns with small support size, the strings are placed inside a hash, once the hash exceeds some size, storage reverts to a more straightforward manner, which is not even a dictionary, so that later-inserted strings may appear multiple times. (This feature has not been described in published scholarly work, and would require examining the source code [Thea]).

Hash-based placement of element data also has uses when Join’ing columns. Of particular interest is the case of a pair of columns to be Join’ed, both of which share such a compression scheme: A hash-table layout of the dictionary can serve as an imperfect surrogate of a hash table of the entire column — saving most of the work of a hash-based Join implementation, except for those elements with hash collisions.

### 3.4 Segmentized schemes

#### 3.4.1 Uniform-value-run encoding

Database table columns may exhibit long sequences of identical values — a run — without being constant overall. The typical example are the columns of a multi-column key by which a table is sorted, with the exclusion of the last column; and these are quite prevalent in practice (as the authors of C-Store report, for example, in [AMF06, §4.3]). With our extensive previous discussion in Subsection 3.2.3, and recalling the basic schemes in Section 3.1, we can capture these runs by applying the CONSTANT compression scheme to each of them, and treating the concatenation of these stretches as a segmented column, i.e. segmentizing the CONSTANT scheme. The resulting scheme is:

**RunEncoding**

| Label   | Type  | Length | Description         |
|---------|-------|--------|---------------------|
| start_position | \(\tau\) | \(n\)  | \(n\) of each run  |
| length  | \(\tau\) | \(n\)  | \(n\) of each run  |
| value   | \(\tau\) | \(n\)  | The repeating value of each run |

The uncompressed column has the value value\([i]\) at all positions position\([i]\) \(\ldots\) position\([i]\) + length\([i]\) − 1.

This scheme achieves effective compression only if the run lengths are not too short on average. To quantify “not too short”, consider the specific case of \(\tau = \tau\) \(\text{int}\): The representation of each run requires one extra element in each column, which in our limited case is \(3 \tau\) \(\text{int}\) elements; thus, for a compression rate of 2, the average run length would have to be at least 6. This begs for some further space savings, so as to be useful with more modest average-length of runs. Reducing the overhead is indeed possible, since start_position and length columns are almost entirely redundant with each other: The former is a discrete integration of the latter. Dropping any one of these columns results in a different scheme:
RunPositionEncoding

RunEncoding without the `length` column, but with an additional scalar `overall_length`. Or, a standard sucolumn representation of the first elements of each run, plus the overall length necessary to bound off the last run. (Shorthand name: RPE.)

RunLengthEncoding

RunEncoding without the `start_positions` column. (Shorthand name: RLE.)

There is a trade-off in the choice of which of the redundant columns remains and which is discarded: If we keep the `start_position` column, we get simplified decompression — as the value of any element of the output is determined by the triplet of elements for a single run, without influence by any of the other runs; decompressing using the `length` requires discrete integration — a much more costly operation with deep dependency chains among elements. On the other hand, run lengths are typically much shorter than the entire column length, in which case we can further compress them with DiscardHighBits — with minimal decompression overhead; this motivates keeping `length` over `start_position`. In fact, we can ensure that this typical occurs always, with a slightly modification of the RLE scheme:

CappedRunLengthEncoding

A scheme similar to RLE, but with \( \text{Dom}(\text{length}) \subseteq \{0, \ldots, r - 1\} \) for some \( r \) provided as an extra scalar. When encoding, runs longer than \( r \) elements are broken up into consecutive runs of length \( r \), and a final run of up to \( r \) elements — all with the same run value. (Shorthand name: CappedRLE.)

Note. The RLE scheme is very well-known — albeit not in a columnar formulation — and in extensive practical use in a variety of applicative fields, dating back as far as analog compression of US Television broadcast signals in the 1960s [CKPB63, RC67]. The capping of run lengths, so as to limit the amount of information required to encode it, and the consequent introduction of additional runs of the same value, was discussed and put into practice already in those early days of digital telecommunications [CKPB63, §F]. The RPE scheme is not as frequently used; this is likely because it is less suited for communication streams, being anchored to some starting position. Also, the decompression penalty when using RLE only occurs in a parallelized setting; with serial decompression, integration is just as easy as differentiation (neglecting numeric stability issues), slanting the trade-off mentioned above in favor of RLE. The RPE scheme has, however, recently proved quite useful for linear algebra computations on compressed data (see [EBH+16, “Offset List Encoding” in §4]).
An implementation of the IsSameAsPrevious operator

Figure 3.6: Compressor and decompressor for the RunPositionEncoding compression scheme

Finally, we consider two variants of RunEncoding and its related schemes, involving further segmentation:

**Fixed-width pre-segmentation** A significant difference between the columnar formulation of RLE and its use in practice has to do with the common practice of holding columnar data in segments, “blocks” or “chunks” (see Section 2.3). Despite RunEncoding, RPE or RLE being inherently segmented, this is a variable-length and data-dependent segmentation; to better reflect existing systems, we need to consider the application of an artificial fixed-width segmentation to a column before applying one of the run-based schemes. The price of this pre-segmentatization; Non-negligible decompression overhead, and no chance for exceedingly-high compression ratios with extremely-long runs. But there is also a significant gain: In addition to the bounding of search lengths, the capping of run length and the fact that position offsets are relative to the segment,
so both RLE and RPERunPositionEncoding now require less bits per run — which changes the trade-off between them somewhat in favor of RPE.

Implicit sub-segmentation In this variant we stick to the single segmentation inherent in RunEncoding-like schemes: The runs. Instead of adding a second segmentation, we break up existing runs into smaller runs — so that it becomes also a sub-segmentation of a uniform $\ell$-segmentation; that is, whenever a run contains both the $(i \cdot \ell - 1)^{th}$ and the $i \cdot \ell^{th}$ element - we break it into two runs on the length-$\ell$ “segment” boundaries. The decompression circuits for RunEncoding/RLE/RPE can be used as-is for these schemes. Also note that applying this sub-segmentation to RLE column after this refinement, and adjoining $\ell$ as a scalar, results in an instance of the CappedRLE scheme mentioned above.

3.4.2 Splines: segmentized generated functions

We’ve just explored compression schemes using the segmentation of CONSTANT — RunEncoding, RPE and RLE — representing columns as consecutive uniform-value runs. Recalling that columns are simply functions, we’ve in fact been discussing representations of step functions. In Subsection 3.4.2, we generalized from constant functions to elements of linear spaces of basis functions, such as low-degree polynomials; and we now do the same in the segmentized context. Indeed, we will use the well-studied generalization of step functions (to low-degree polynomial functions): The spline.

In fields such as computer game graphics, industrial design and typography, one often wishes to model curves (or surfaces in 3-dimensional space) using polynomial functions. However, high-degree polynomials are harder to reproduce physically and slower to compute; also, they typically exhibit high-amplitude oscillation between their points of constraint, particularly near edges of the intervals of definition (Runge’s phenomenon). It is thus usually more convenient to subdivide the domain, and use multiple low-degree polynomial pieces, one per segment, rather than a single high-degree polynomial. Historically, their use arose from physical splines, pieces of wood used in craft design which would bend into the minimum-energy form, while being bound by physical constraints of position and angle at their endpoints. For an exposition of the historical development and varigated uses of splines, see [DB01].

A similar principle is relevant for data compression also: If the data has a certain shape locally, be it a gradually-changing shape or with points of discontinuity — this overall tendency or shape would be captured relatively well by a spline, at a certain resolution of segmentation (whether fixed or variable, and with or without continuity and smoothness constraints).

The proper segmentization of $\text{GENERATED}_f$ yields the following scheme:

**GeneralizedSpline**$_f$

| Label                | Type | Length | Description                                                                 |
|----------------------|------|--------|-----------------------------------------------------------------------------|
| segment_start_pos    | $\tau_{\text{int}}$ | $m$    | of the first element of each segment                                       |
| segment_length       | $\tau_{\text{int}}$ | $m$    | Non-negative number of elements in each segment                             |
| coefficients         | $\tau$                     | $m \cdot k$ | a concatenation of $m$ sequences of $k$ coefficients for the $f_i$, forming a different function for each segment |

**Spline**

The case of $\text{GeneralizedSpline}_f$ where the basis functions are monomials, i.e. $f_i(x) = x^i$. 

59
These schemes have a somewhat redundant representation — similarly to the case of RunEN-coding. We may compress it further by dropping one of the two columns of the segmentation. Here we spare ourselves the dilemma by following the traditional formulation of splines, which focuses on knots, the points delimiting the segments:

**KnottedSpline**

| Label       | Type  | Length | Description                                                      |
|-------------|-------|--------|------------------------------------------------------------------|
| knots       | \(\tau\) | \(m + 1\) | start/endpoints of segments                                      |
| coefficients| \(\tau\) | \(m \cdot k\) | a concatenation of \(m\) sequences of \(k\) coefficients for the polynomials \(x^0, \ldots, x^{k-1}\) |

The first knot is always 0 and the last is \(n - 1\), with \(n\) being the uncompressed column length.

**EquiknottedSpline**

| Label         | Type  | Length | Description                                                      |
|---------------|-------|--------|------------------------------------------------------------------|
| interval_length | \(\tau\) | 1       | the distance between consecutive knots                           |
| coefficients   | \(\tau\) | \(m \cdot k\) | a concatenation of \(m\) sequences of \(k\) coefficients for the polynomials \(x^0, \ldots, x^{k-1}\) |

The uniform-segmentation equivalent of KnottedSpline. The last interval may be shorter than the uniform length.

and we could similarly define knotted versions of GeneralizedSpline.

Spline functions are technically a generalization of step functions — which are captured by the segmentized CONSTANT scheme. And yet, in practical use, it is in fact the step-function-describing RPE and RLE schemes that are effectively the more general, or rather more widely applicable directly. This is due to their being much more “forgiving” towards diversions from the modeled function: If some elements do not fit the current (constant) step/run value - we may simply add a new run for it, with only local overhead — another run, and possibly breaking up an existing run in the middle. With splines, one needs quite a few coefficients. Also, in practice, one does indeed encounter long stretches of identical values; and possibly some long arithmetic sequences (degree-1 splines); but column data being perfectly quadratic or cubic over some stretch are rare occurrence. Thus, paradoxically, one would tend not to use the generalized scheme as-is, but rather combine it elementwise with small additive “noise” column (as discussed in Subsection 3.2.1).

**Smooth splines** Spline functions are often constrained to be continuous curves, i.e. constrained to be continuous at the internal knots. Stronger smoothness constraints are also sometimes introduced, so that the polynomial “pieces” have the same direction at the point of joining (e.g. to avoid jagged edges on the surface of a car or side of a ship, whose contour is set to a spline); and further constraints are sometimes placed on the higher derivatives. These constraints create linear dependencies between the coefficients of the polynomials (or outright equality constraints if we choose another polynomial base, as in B-splines [DB01, Chapter 10]) - reducing the number of parameters necessary per segment. One would typically still opt to keep the full sequence of redundant coefficients for each knot, as failing to do so requires reconstructing them with a pass over previous segments.

**Splines with non-integral knots** Monomials — the basis functions for the SPLINE scheme — can be applied to any positive real value rather than just to the (integral) column indices. The same would often be true for other choices of \(\tilde{f}\) base functions. With this in mind, one could specify
non-integral spline knots — using some fixed-width floating-point type for the knot positions — in the schemes above, or a non-integral interval length for equi-knotted splines and generalized splines. The column values are the evaluations at offsets which are integral relative to each other, but non-integral relative to the knot. This may reduce the complexity of fitting optimal splines to data.

**Frame-of-Reference**  The composition of EQUIKNOTTEDSPLINE₁ with elementwise additive differences (see “noise-signal” separation in Subsection 3.2.1). At every segment, the additive differences can be thought of as offsets from a reference value or a baseline value for that segment — the uniform value of the step function. This special case is commonly known as:

| Label       | Type | Length | Description               |
|-------------|------|--------|---------------------------|
| segment_length | τₖ₁  | 1      |                           |
| reference     | τ   | 1      |                           |
| offsets       | τ₀   | n      |                           |

The $i$th element of the uncompressed column is $\text{offsets}[i] + \text{references}[i/\text{segment_length}]$. (Shorthand name: FOR.)

The FOR has been in use by several analytic DBMSes (e.g. Teradata [Mor02] and Vectorwise [ZHNB06] composed with aposteriori patching).

### 3.4.3 Compressed Differences

Consecutive elements in real-world database table columns are often close to their predecessors: Timestamps; samples from real-world data sources with low energy in high frequencies; arbitrary data after having undergone a sort; and so on. In such cases, each element’s value is a fairly decent estimate for the value of the next one — even if a model for the entire column is not available or is complex. When these element-to-element differences are all integral (or of fixed precision), they may fit a smaller-size type, and the following scheme achieves meaningful compression:

| Label | Type | Length | Description                           |
|-------|------|--------|---------------------------------------|
| base  | τ    | 1      | Arbitrary value                       |
| delta | τₐ   | n      | Differences between consecutive uncompressed elements |

The first element of the delta column holds the difference between the first uncompressed element and the base value (so that the base value is not uniquely determined by the uncompressed column).
As defined, NAIVEDELTA is not a very practicable scheme: The computation of each decompressed element depends on all previous input elements — possibly millions or billions of them. It is at this point that segmentation comes into play: Instead of a change in any compression scheme parameters per se, we apply it to NAIVEDELTA to break the chain of dependencies; dependencies will now extend only to the segment boundary rather than all the way to the beginning of the column.

**Delta**

| Label          | Type | Length                  | Description                                      |
|----------------|------|-------------------------|--------------------------------------------------|
| segment_length | \(\tau\) | 1                       | one value per segment                            |
| base           | \(\tau\) | \([n/\text{segment}\_\text{length}]\) | Differences between consecutive uncompressed elements in a segment |
| delta          | \(\tau_\Delta\) | \(n\)                  |                                                  |

The first element of each segment, i.e. \(\text{delta}[\text{segment}\_\text{length} \cdot i]\) for the \(i\)th segment, holds the difference between the segment’s first uncompressed element and that segment’s base value, \(\text{base}[i]\).

An alternative to segmentizing NAIVEDELTA, is characterizing the behavior of the differences **globally** rather than **locally** — with model functions as in Subsection 3.4.2 above. But that is not, in fact, a different compression scheme — as a characterization of the derivative as a span of basis functions is the same as a characterization of the underived function, with the anti-derivative of those basis functions (up to a global constant). Thus, NOISYGENERATED may be a reasonable alternative to NAIVEDELTA; and while both can be segmented, it will be less essential to do so for the latter scheme; NOISYGENERATED would also be much easier to decompress, with its work being elementwise.

**Patching segmentized differences** The long dependency chains created by representing differences rather values remains somewhat problematic even with segmentatization. In addition to the limits of the parallelization of decompression, the use of (aposteriori) patching also becomes difficult. To illustrate, suppose \(\tau, \tau_\Delta\) are integral types and that \(\text{sup} (\text{Dom}(\tau_\Delta)) < M\). Now suppose we wish to compress a column made up of \(k\) occurrences of 0, followed by at least \(k\) occurrences...
of \((k+1)M\). One can show that at least \(k\) patches are necessary to bring all consecutive pairs of elements to within \(M\) of each other; consequently, at least \(k\) aposteriori patches must be made to a \textsc{Delta}-compressed form of this column. If, instead, we were to patch the compressed column — after casting it into \(\tau\) rather than \(\tau_\Delta\) values — this would require only a single patch at index \(k\).

The adjoining a subcolumn in standard representation to the compressed form of \textsc{Delta}, of patches to be applied as we’ve just described, defines another compression scheme — \textsc{PatchedDelta} scheme or \textsc{PDELTA} for short. A non-columnar equivalent of this scheme is defined in [ZHNB06, §3.1] as \textsc{PFOR-Delta}.

### 3.4.3.1 Segment-local dictionaries

Segmentation allows mid-column changes in a compression scheme’s parameters. In subsubsection above we described the localized modeling of a column in a space of low-degree functions — fitting a different model for each segment. We now turn to a different assumption of local “well-behavedness”: An assumption regarding the column support set.

For example, consider a column with a poor (discrete) modulus of continuity, for which one cannot provide a well-fitting model using typical base functions (e.g. polynomials), even if one were to allow for a narrow noise column. But suppose also, that despite its discontinuous nature, the column’s support set does exhibit locality: Taking neighborhoods around each position in the column, the local support size is a power significantly smaller than 1 of the column’s overall support size. In a nutshell: The support set is globally large, but locally small (and not dense).

To exploit this locality, we break up the column into contiguous segments, each of which with its own hopefully-small local dictionary: Not too large, so as to limit the size of the support set; but not too small, so as not to incur excessive storage overhead for the dictionaries, nor decompression speed overhead for reading them. Keeping a dictionary for each of these segments results in the following compression scheme:

**UniformSegmentDictionaries**

| Label                | Type   | Length                     | Description                      |
|----------------------|--------|----------------------------|----------------------------------|
| segment_length       | \(\tau\)\(_{\text{int}}\) | 1                           | with positive elements           |
| dictionary_entries   | \(\tau\) | \(d \cdot \text{segment\_length}\) | Entries for all dictionaries, concatenated into the respective segment’s dictionary |
| indices              | \(\tau\)\(_{\text{int}}\) | \(n\)                       |                                  |

Decoded similarly to \textsc{Dictionary}, except that the dictionary entry index is not \(\text{indices\_\{i\}}\), but rather \(\text{indices\_\{i\}} + (i/\text{segment\_length}) \cdot d\).

This local-dictionary scheme is used essentially as-is, albeit not in columnar formulation, in existing systems (e.g. Oracle [PP03] and Vertica [LMV+12]). It also has quite a few possible variations, or further developments:

**De-duplication**  We can constrain the per-segment dictionaries to have no duplicate values and/or to be sorted — allowing for the local-dictionary indices to serve as surrogates; these would be the same constraints described in Subsubsection 3.1.2.1 for the case of a single all-column dictionary. Note, though, that the surrogacy in the segment-dictionaries case would be limited to the relevant segment: The comparison of indices into dictionaries of different segments does not yield a meaningful result.
Patching  The segmented dictionaries scheme relies on complete disappearances of elements from support sets from segment to segment. In some real-life cases, this does happen; for example, a column sorted using another, categorical, column (e.g. day of the week), where the support for different categories is distinct. But if the column’s support has a more continuous behavior, disappearance will not be complete; Instead, we’re more likely to see supports with the vast majority of weight on a small set of values, and some weight on a large set of very-infrequent values. In other words, the segment support sets will not be small, but the frequency functions of values in the column will have almost all of their weights on a small set each. To be able to ignore the distributions’ “tails”, we may use patching. The patches can theoretically be themselves segmentized rather than being column-global, but it is not clear whether this segmentation is particularly useful, so all-column patching seems more attractive.

Variable-length segments  We’ve made an off-hand choice of using a uniform segmentation with the local dictionary, but we can also opt for a non-uniform one (see Section 2.3. Such a choice may be relevant if the changes in the local support set are themselves non-uniform, i.e. if there are long stretch where the support set hardly changes, and others where it changes at a faster pace. Remember, however, that instead of adding another level of indirection, one can also consider patching for some problematic parts of a column; increasing the dictionary size, effectively obtaining a slower change; or using a two-level dictionary (see below).

A particularly interesting scenario where the locality of support breaks is that of a discontinuity in the occurrence frequency function: After a certain element, the support set changes abruptly, perhaps even completely (and this will typically not be aligned with a segment end). Variable-length segments seemlessly addresses this case — but it can also be handled by a doubling of the dictionary size: If the discontinuities are at least segment_length elements apart, a double-sized dictionary could have both the current segment and the next segment’s support set covered, disappearing the discontinuity.

Two-level dictionaries  A drawback of UniformSegmentDictionaries, as defined, is the amount of space necessary to store all of the dictionaries, especially when the segment length is not very high relative to the dictionary size (due to relatively weak support set locality). Also, a global support set which is itself compressible, or when the original column has very large elements or variable width — adversely affect the storage of segment dictionaries in terms of complexity, size or both. Both these issues may be addressed by composing UniformSegmentDictionaries after an application of Dictionary to the uncompressed column. The first level of dictionary use — the global dictionary — results in fixed-width integers, regardless of the original element type; and pre-reduces their sizes in the second-level dictionaries – the segment dictionaries.

Such a two-level scheme (with de-duplication and sorting) is used in the Google PowerDrill column store [HBB+12, §2.3] (again, in a non-columnar formulation). Specifically, schema columns are physically partitioned into “chunks”, each one being much more complex than a plain column

Compression of the segment dictionaries  The overhead of storing a dictionary per each segment poses a dilemma: If our segments are short, we may need to store so many dictionaries that the space taken up by dictionary_entries is of the same order of magnitude as the space taken up by the indices column, compromising the compression benefit of the segment dictionaries in the first places. On the other hand, if segments are long, the dictionary for each segment must
be larger, as we lose some of the effect of locality; and while the dictionary’s size is offset by the lower number of dictionaries — the dictionary index width will increase as well, dragging the overall compression ratio down.

One can conceive of several ways of circumventing, or resolving, this dilemma:

• Expand the dictionaries slightly (hopefully without increasing the dictionary index size) beyond the size necessary for our fixed choice of segment length. Now, construct ‘union dictionaries’ of the dictionaries of consecutive segments, which can be used for stretches of several consecutive segments. We could then apply RUNENCODING or RLE to avoid duplicating these dictionaries.

• Make some effort to reduce the number of distinct dictionaries (e.g. by creating “segment union dictionaries” as in the item above). Avoid the repeated use of the same dictionary using a dictionary-of-dictionaries, i.e. adjoin a column with one \( \tau_{\text{int}} \) element per each segment of indices, indicating the index of the dictionary to use for that segment; the distinct dictionaries, having some fixed size, are concatenated in dictionary_entries.

• We’ve been treating the different dictionaries as perfectly distinct stretches of data, despite them being placed together in the same column. We could remove this constraint, and allow a dictionary to start at an arbitrary position within the dictionary_entries column. Thus less-frequent elements at the beginning of a dictionary may be discarded by moving its starting index up, but not all the way, so that many or most elements remain. The dictionary length will also be set so as to introduce the appropriate amount of new elements, and possibly also to re-introduce elements which were lost at the beginning but are still necessary; at the same time, we’ll need to avoid enlarging it too much to avoid widening the indices into it.

This approach complicates the compression somewhat, as one must tackle the combinatorial challenge of choosing a dictionary index width for which the exist a satisfying placement of elements: Such a placement must include enough redundant appearances of values to capture segments’ support, but not too many appearances so as to spoil the overall effect of dictionary overlap.

• Instead of storing entire dictionaries, store the differences between consecutive dictionaries (i.e. which elements were added and which removed); due to the locality property, and the choice of smaller segment length, consecutive dictionaries mostly overlap (if we ignore placement within the dictionary), on the average. Thus, on average, their differences should take up little space. For more on the use of differences in compression, see Subsection 3.4.3 above.

... and if this option reminds the astute reader of Intra-frames and P-frames in MPEG video compression [DL96] — the analogy is not without merit.

### 3.5 Exploiting non-uniformity in frequency distributions

In seeking features to exploit for compression, and particularly in considering a column’s support set, we have so far mostly ignored the frequencies of values appearing in the column: We’ve treated each of them as equally worthy of compression, with an equal potential benefit from compression; and the only allowance for infrequency we’ve presented is patching, which is useful removing
extremely-infrequent elements entirely. In other words, we’ve mostly ignored the non-uniformity of columns’ frequency distribution (as per Definition 1.2).

We begin with a classic (non-columnar) approach to exploiting such non-uniformity for coding — using elementwise variable-length coding of the originally fixed-length data, so that frequent elements use less bits. Then, in Subsection 3.5.2, we consider a conceptual of “transposition” of the above, where instead of having one variable-width column, we have a multiple, uniform-width columns, but of decreasing lengths, i.e. where some elements “end” in an earlier column. Finally, we focus on a single aspect, or step, of the multiple-column approach, involving a two-way partition of the frequency distribution function, that is useful in itself as a building block for compression schemes, and is in a sense the complement of patch removal; this is Subsection 3.5.3.

Note. In information-theoretic terms, in this section we exploit low (Shannon) Entropy [CT12, §2.1] of columns’ element distributions.

### 3.5.1 Huffman-like encoding of variable-width columns

Given a column with highly non-uniform element frequencies, it may be worthwhile to apply an elementwise approach of exploiting this non-uniformity: Having frequent elements be represented in less space and infrequent elements taking up more space than in the fixed-width representation. This is an extremely common technique in communication and (serial, non-columnar) compression, particularly using the Huffman code [Huf52] for representing elements; this code is optimal for symbol-for-symbol prefix codes (see [CT12, §5.8]. A similar choice, for the additional constraint of preserving the relative order of the unencoded values in the encoded form, is the Hu-Tucker code [HT71, Hu82].

But what is a straightforward in a serial context is not to be considered lightly in parallelism-focused computational computation: Variable-size rather fixed-size representation of data comes at a cost, of either extra space or extremely poor amenability to parallel decompression. With our standard representation of variable-width column, see Subsection 2.4.4) we pay in a significant amount of extra space, to indicate element positions or lengths. Without those indications, however, it would generally not be possible to start inspecting the variable-length data at some mid-point and determine where an uncompressed element starts or ends; decompression would therefore have very long dependency chains to follow over numerous compressed-form column elements.

Two approaches for achieving a middle ground between minimum size and more shallow data dependencies are:

- Use of periodic, rather than per-element, position/length indication (a-la FrameOfReference or Delta); between these indicators, decompression is almost impossible to parallelize, in that one cannot determine where elements start or end by examining them from some midpoint between the indicators.

- Interspersing synchronization elements between actual compressed column elements, which can be identified with certainty even when the input is read starting at an arbitrary position. Without delving into the details, applying this code-theoretic concept would be based on the exploration of the subject in [FR84, Tit97]; but would differ due to our ability to pre-transform the column to enforce certain features or remove certain artifacts, using patching; this may allow for simpler decompression, or for an improved upper bound on the compression ratio relative to the classic case.
Finally, note that compression using variable-width columns can exploit more than the mere element frequency distribution, but rather than frequency distribution of pairs, or longer sequences, in the uncompressed column. This would bring it somewhat closer to Lempel-Ziv-style compression schemes [ZL78].

3.5.2 Cascading dictionaries

We wish to benefit from using a shorter representation for more frequent elements — but perhaps we are loath to sacrifice so much of the uniformity and regularity of fixed-width columns, which is required for Huffman-like coding. In this case, we may adopt a compromise: Fixed-width columns, but several of them, or rather, several complementary subcolumns — each with a different width. Of course, the representation in each width would be a fixed-width number, or more specifically: An index into a Dictionary. Elements will be assigned indices to these subcolumns (and given a dictionary entry) based on the frequencies in the uncompressed column.

Consider the first, narrowest dictionary, $D_1$; its index width is $b_1$ bits, and it covers an $\varepsilon_1$ fraction of the column’s values. One of its entries, however — we’ll choose entry 0 for simplicity — is reserved for indicating that an element is not covered by the dictionary. We may think of the indices column of the DICTIONARY scheme also as a dense subcolumn representation (if we were to check it for equality with 0, elementwise). After these elements have been added to the dictionary and the problem resolved, we remain with the task of compressing merely the $1 - \varepsilon_1$ fraction of the values which do not appear in $D_1$, as a contiguous column. If the residual subcolumn still meets the criteria of this section (non-compactable support set, roughly obeys a power law) — the process may be repeated: Dictionary $D_2$, with $b_2 \geq b_1$ bits covering $\varepsilon_2$ of the subcolumn; and again, a new residual subcolumn is formed, to be compressed further.

| CASCADED.Dictonaries | Label   | Type | Length     | Description                          |
|----------------------|---------|------|------------|--------------------------------------|
| dictionary         | $\tau$ | up to $2^b_i$ | Entries of the $i^{th}$ dictionary |
| ...                 | ...    | ...  | ...        | ...                                 |
| dictionary         | $\tau$ | up to $2^b_k$ | Entries of the $k^{th}$ dictionary |
| indices            | $b_1$-bit integers | $n$ | into dictionary; an index of 0 indicates elements to be set in later phases. |
| ...                 | ...    | ...  | ...        | ...                                 |
| indices            | $b_k$-bit integers | $n_k$ | into dictionary. |

Decompression occurs in $k$ phases, each using two circuit inputs columns, dictionary, and indices, and an additional, intra-circuit, input residual, — one of the outputs of the previous phases. The phase produces two outputs: residual, for use in the next phase, and a sparse-representation subcolumn sc = (pos, data). The phases can be interpreted as successive application of SUBCOLUMNOverlay. The final phase’ residual output is empty, and is not used.

The $i^{th}$ phase gets the positions of those output column elements which have not yet finalized, from residual. It partitions this set of indices into two subsets: Indices to be finalized in this phase — pos — and indices left for later phases — residual. The partition uses indices: An index residual[i−1][j] is placed in residual if indices[i][j] = 0, and in pos otherwise. The phase concludes with DICTIONARY decoding of pos and dictionary.
An initial implicit column, \( \text{residual}_0 \), is the identity column for \( \{0, \ldots, n - 1\} \), i.e. initially, all elements are still not finalized. The disjoint subcolumn union \( \bigcup_{i=1}^{k} \text{sc}_i \) constitutes the decompressed column in default subcolumn representation.

A single phase of the decoder (see Subsection 2.4.2 regarding partition materialization.)

The full-blown \((k\text{-phase})\) decoder

**Figure 3.8:** Decoder and subcircuit implementations for the \text{CascadedDictionaries} compression scheme

We have not described an exact procedure for \textit{partitioning} the elements into \( k \) sub-sets. Naturally, the earlier dictionaries involve the more frequent elements, but there is still the choice of width for each dictionary to be decided. The more emphasis one places on shallower dependencies during decompression, the fewer dictionaries one would tend to use and wider they’ll be; and vice-versa — optimizing the compression ratio may involve more, narrower dictionaries.

\textit{Note.} The idea of partitioning columns by element frequency, to be able to use shorter-width dictionary indices for more frequent elements has already been utilized in lightweight compression in a DBMS context: IBM has used a “frequency partition”-based compression scheme in it’s BlinkDB \[RSQ^{+08}, \S III\]. The scheme is rather different from \text{CascadedDictionaries}: BlinkDB is a row store, so the scheme there is applied to tuples; also, it compresses without preserving tuple order, so that the relative position of the various tuples is not encoded. With this said — it is an interesting composite scheme also involving \text{Delta}, which could be expressed in terms of the basic schemes and composition patterns presented in this chapter; and the discussion of partition formation \[RSQ^{+08}, \S 3.b\] is also of interest.

### 3.5.3 Small-dictionary fitting

While the \text{CascadedDictionaries} scheme discussed in the previous section, is certainly of some utility, its definition involves something possibly even more useful: The single phase of its decompression, with a decoder as per Figure 3.8a. Instead of its repeated application on residual columns, let us apply it just once, and assume the decoded values for the residual are available explicitly:

\text{SubcolumnDictionary}
When decompressing, dictionary and indices are treated just as in the DICTIONARY scheme: decompressed ← dictionary ◦ indices — except for the elements of indices⁻¹(0); these have individual values provided, in order of their position, via the residual_data column.

We can indeed formulate CascadedDictionariesₖ as a repeated composition of SubcolumnDictionary with itself, k times — with the appropriate subcolumns and bit widths \( \vec{b} \), and dropping the empty final residual subcolumn.

SubcolumnDictionary alone will not yield a good compression ratio unless the dictionary covers almost all column elements. But even if it leaves out a sizable fraction, it is a prime candidate for composing with other encoding schemes, a procedure we refer to as small-dictionary fitting. Such a separation of a column into an “easier” and “harder” part immediately reminds us of the patching procedure (described in Subsection 3.2.4); in some senses, they can be seen as each other’s converse:

| Rationale for use          | Patching                  | Small-dictionary fitting |
|---------------------------|---------------------------|--------------------------|
| Intended scope of application | Remove the infrequent, “difficult” elements | Remove the frequent elements |
| Number of elements remaining to compress after application | The full column’s length | Maximum possible          |
| Compression ratio of affected data | Pretty bad (typically worse than uncompressed) | Pretty good (typically better than rest of data) |
| Overhead likely proportional to Repeat application | Number of exceptional elements | Total number of column elements |
| Scheme used for composition | SUBCOLUMNOVERLAY | SUBCOLUMNDIRECTORY |

Small-dictionary fitting can therefore take its place alongside the fundamental, or commonplace, scheme composition and transformation patterns in Section 3.2.

3.6 Further compression of scheme-generated auxiliary columns

3.6.1 Patch positions — sparse index subsets

Patching, described in Subsection 3.2.4 and used in several concrete schemes presented in this chapter, has a very steep space overhead per patched element: We need to store not only the full-size uncompressed element, but also an index into the uncompressed column. And the more effective the main compression scheme, and the longer the column — the more “painful” this extra overhead of specifying the patch position becomes. As a consequence, naive patching — a subcolumn of patched value in SPARSEINDEXSET representation — is only beneficial with very low fractions of values to patch in a column.

Luckily, index subsets which aren’t extremely sparse tend to be amenable to some compression, due to two characteristics: They can be arranged in monotone-increasing order; and the denser they
are, the lower the arithmetic differences between subsequent indices are on the average, even well below the density at which one might consider switching to a dense representation.

To simplify the exploitation of these two properties, we focus on long common prefixes of indices’ binary representation — since if subsequent indices are close together, then on the average (though with slightly lower probability) they share a long common prefix. Indeed, we can define a family of representation schemes concretizing the prefix-agreement by grouping together the representation of identical prefixes:

| Label      | Type              | Length | Description                                      |
|------------|-------------------|--------|-------------------------------------------------|
| prefix     | unsigned integer, \(p\) bits | \(m'\) | each to be prepended to multiple suffixes        |
| suffix\_count | unsigned integer, \((w - p)\) bits | \(m'\) | Number of suffixes for each common prefix       |
| suffix     | unsigned integer, \((w - p)\) bits | \(m\)  | each preceded by the appropriate prefix         |

With this scheme, (encodings of) indices appear in order of index value. As consecutive indices tend to share their higher (more-significant) bits, we apply a \(\text{Carve}_{w,p}\) operator, carving each \(w\)-bit integer into its higher (or prefix) \(p\) bits and its lower (or suffix) \((w - p)\) bits is performed. The column of \(p\) high bits of patch positions exhibits (hopefully long) sequences of identical values, so we apply RLE to it. The lower bits column remains untouched, completing the encoding. This procedure is clearly reversible, up to the sorting — but sorting maintains the equivalence of the standard representations of index sets, so it unnecessary when decoding.

An index subset representation in this family may take up as many as \((p + (w - p) + (w - p)) = 2w - p\) bits per element. This worst-case occurs if each element has a different prefix: It requires \(w\) for the actual carved-up element and \((w - p)\) bits for representing the length of the one-element subset. On the other hand, the more subset elements we have relative to the number of prefixes, the less the size-per-element of the \(\text{prefix}\) and \(\text{suffix\_count}\) (whose absolute size can never exceed \(2^p \cdot \text{paren}\) + \((w - p)\) = \(2^p w\)). Thus, for \(2^p = o(m)\) (both as functions of \(n\), we have a size per element of \((1 + o(1)) \cdot (w - p)\) — mostly just the suffix.

The former, extra-sparse case motivates choosing a higher value of \(p\), as for \(p = 0\) we again reach the sparse representation; but the less-sparse case motivates a lower value of \(p\). We can balance these contradicting considerations by using multiple distinct (and disjoint) subcolumns: Each encoded in \(\text{CommonPrefix}\) with the same value of \(w\) but different values of \(p\). Preferring simplicity over optimality for demonstrating the principle, let us choose, with foresight, two \(p\) values: 0 and \(w/2\) (assuming \(w\) is even). This results in the following compression scheme for index subsets:

\(\text{CommonUpperHalf}_{w}\)

Each element of the subset to be compressed is placed in one of two subcolumns, which are themselves represented using \(\text{CommonPrefix}_{w,0}\) and \(\text{CommonPrefix}_{w,w/2}\) respectively (assuming \(w\) is even for simplicity). The choice of placement for the different values depends on the density of the set’s intersection with each consecutive \(2^{w/2}\)-aligned sequences of \(2^{w/2}\) elements (i.e. the subsets of elements which all share the same \(w - w/2\) prefix): If a patch
value is the single one within its $2^{w/2}$-aligned sequence, it is placed in the 0-prefix subcolumn; and if there are multiple elements, they are all placed in the $w/2$-prefix subcolumn. The subcolumns are disjoint, so the decoding employs DISJOINTSUBCOLUMNUNION after decoding each of the individual subcolumns.

By our earlier analysis, the size per index subset element of a COMMONUPPERHALF representation is $\min(1, \frac{1}{2} + o(1)) \cdot w$ for $m = \omega(\sqrt{n})$.

Returning finally to the context of patching, suppose a column of length $n$, has $\varepsilon n$ of its values requiring patches. With COMMONUPPERHALF $\lceil \log(n) \rceil$, the amortized per-element overhead (in bits) of storing patch indices is $\varepsilon \cdot (\frac{1}{2} + o(1)) \cdot \lceil \log(n) \rceil$ — so that, asymptotically, the patches require about half as much space as their naive representation. To make this more concrete, suppose that $\varepsilon n > 10 \sqrt{n}$ (which, for $n > 10^6$, means $\varepsilon > 1\%$); the index representation size per patch element is then under $0.6 \lceil \log(n) \rceil$, rather than the full $\lceil \log(n) \rceil$ for naively-represented indices.

Note. Common prefixes are a sub-optimal form of clustering the indices together by their arithmetic distance; and so is the choice of only length-0 and length-$w/2$ prefixes. One could, instead, undertake the computation of an actually optimal choice of cluster reference points, and assignment of indices to the clusters. This would be a non-trivial special case of the min-max facility location problem: Even if the optimal number of clusters ($k_{opt}$) is known in advance, the best known upper bound on the time complexity seems to be $O(k_{opt} \cdot n^2)$ [WS11] — which is extremely prohibitive for long columns. If we also allow for the possibility of switching from a sparse to a dense regime locally in some cluster (as in Roaring bitmap compression [CLKG16]), the complexity increases even further.

3.6.2 Element length specifiers

Using a variable-length representation of a column’s elements often yields significant compression ratio improvement; but in the standard representation, and other compression schemes, this is offset by the significant overhead of storing elements’ start positions. In the most useful cases of variable-width columns, the frequent elements only use a few bits — in which case the starting positions takes up more space than the data itself. Thus, to effectively compress variable-width columns, they themselves must be compressed.

Fortunately, the element lengths, taken as a column, are often amenable at least to some of the simpler compression schemes, or a combination thereof: The range of extant element length is usually limited, and only as many bits as the logarithm of its size is required. Achieving this might requires using something like DICTIONARY in the general case, but often that’s not necessary, since the possible lengths are very dense in a certain range, thus FRAMEOFREFERENCE (see paragraph 3.4.2) followed by DISCARDHIGTBITS produce a near-optimal compression. The segmentization of the FRAMEOFREFERENCE scheme may also come in useful, if the length distribution changes gradually.

3.6.2.1 Shaping element lengths by padding

Beyond the straightforward approach above, there’s a special feature of some variable-length representations which may be utilized to further improve the compression of a column of their length indicators:

- For unsigned integers, in the ubiquitous base-2 bitwise representation, a representation can always be padded with 0’s as the top bits without changing the value.
• The case of signed integers with two’s-complement representation is similar, but using the sign bit for padding.

• Text strings are usually represented in character set encoding schemes which allow for indicating the end of the string with a special value; this is commonly the “null character”, or the numeric value of 0. It can be repeated past the end of the string for padding.

For these kinds of data, the representation of an element can always be extended to a longer length without altering the value. We can thus trade off extra bytes in the element data column for a better fit to compressible patterns in the element lengths column.

This upwards-flexibility of element lengths is exploited in several well-known, non-columnar lightweight compression schemes: The 4-Gamma and 4-Wise NS schemes element widths for consecutive sequences of 4 elements (to the maximum of the actual widths); SIMD-128 and SIMD-FastPFOR go as far as setting a single length for sequences of 128 elements [DHHL17]. These all incur a significant penalty in compression ratio, however, when the compressed data is not locally uniform in width. Combining this with the fact that variable-length representation only makes sense to begin with when the element distribution is somewhat power-law-like, one is led conclude that the approach of these schemes is somewhat questionable in terms of achievable compression ratio. As an example, consider a column in the distribution of length exhibits a Geometric distribution with parameter $p$ (with minimum value 1 occurring with probability $p$); and that we set the maximum length in bits every $k$ consecutive elements. Letting $W_{\text{max}}$ be the maximum width of some sequence $k$ elements and $W_1, \ldots, W_k$ be their individual widths, we have

\[
\begin{align*}
\text{Ex}[W_i] &= \text{Ex}[\text{Geom}(p)] = \text{Ex}[\text{Geom}(p)] + 1 = 1/p \\
\text{Ex}[W_{\text{max}}] &= \sum_{j=0}^{\infty} j \cdot \text{Pr}[W_{\text{max}} = j] = \sum_{j=1}^{\infty} \text{Pr}[W_{\text{max}} \geq j] \\
&= \sum_{j=1}^{\infty} \left(1 - \text{Pr}[W_{\text{max}} < j]\right) = \sum_{j=1}^{\infty} \left(1 - \prod_{i=1}^{k} \text{Pr}[W_i < j]\right) \\
&= \sum_{j=1}^{\infty} \left(1 - \left(\text{Pr}[\text{Geom}(p) < j]\right)^k\right) = \sum_{j=1}^{\infty} \left(1 - \left(1 - (1 - p)^{j-1}\right)^k\right) \\
&= \sum_{j=0}^{\infty} \left(1 - (1 - (1 - p)^{j})^k\right)
\end{align*}
\]

Taking $k = 4$ (as in 4-Gamma, 4-Wise NS etc.) and $p = (1 - \sqrt[4]{0.1}) \approx 1 - 0.749894 \approx 0.25$ (i.e. 90% of the data takes up at most one byte), we would have an expected $W_{\text{max}}$ of $\sim 7.738$ and expected single-element length of about 3. That would be worse than doubling the amount of space used for the actual compressed elements. This is rather dismal! ... of course if the element lengths are more likely to be identical to the predecessors or successors, such compression schemes can make sense.

Let us, therefore, describe a columnar compression scheme in the same vein as 4-Gamma, 4-Wise NS, SIMD-128 and SIMD-PFOR [DHHL17]:

**PeriodicallyVariableWidth**
Label | Type | Length | Description
--- | --- | --- | ---
period | \( \tau \) | 1 | The period of element length setting
widths | \( \tau \) | \( \left\lfloor \frac{\text{length}}{\text{period}} \right\rfloor \) | The length (in units of \( \tau \)) of the representation of each of \( \text{period} \) subsequent elements of the variable-width column
length | \( \tau \) | 1 | Number of elements in the decoded variable-width column
data | \( \tau \) | \( \sum_{i} \text{widths}[i] \) | Concatenation of the sequences of \( \tau \) values constituting the data of each of the elements

Each sequence of \( \text{period} \) consecutive elements \( \text{period} \cdot i, \ldots, \text{period} \cdot (i+1) - 1 \) of the encoded variable-width column have the same size: They each take up \( \text{widths}[i] \) type-\( \tau \) elements in \text{data}. The same is true for the last few elements (which may be less than \( \text{period} \) in number). \text{total_length} indicates how many variable-size elements the decoded “column” actually has.

**Uniform element lengths via separate subcolumns** The performance pitfalls of the schemes mentioned above (including \( \text{PeriodicallyVariableWidth} \)), along with their requirement of computing a prefix sum over element lengths, motivates the consideration of alternative approaches. One that immediately comes to mind when we recall Subsection 3.5.2 is the “transposition” from variable-length elements to multiple (sub)columns of uniform width. In our case, we can reduce the number necessary columns by padding elements of less-frequent widths — while elements of frequent lengths should not require such padding. The overall number of columns should thus be limited. The down-side of such a scheme is the need to store the position of all these elements. This detriment may also be reduced by some creative use of padding: It is cheaper (possibly much cheaper) to pad an element by another bit (or basic column element type) than to relegate it to a subcolumn of patches.

### 3.7 Multi-column compression

This chapter has mostly been concerned with compression schemes for a single column (with the half-exception of variable-width columns). Using single-column schemes on a table yields one separate and independent compressed representation for each one of the columns; but this separation is often wasteful in terms of space, as some table columns are usually very well correlated. Also, a column store may have use for a copy of several columns in lexicographic order of the tuples they form (see Section 4.2 below). It is thus worthwhile to consider intricating the compression schemes for different columns, to avoid some of the redundancy. Following is a scale of the intensity of such potential intrication:

0. *(Independent representations*: Compression and decompression of each column completely disregards other columns. This is the naive approach, directly following our discussion in this chapter thus far.)

1. *Decoder fusion*: When columns known to be related are decoded (decompressed), the disjoint union of their decoding circuits is considers as a decoding circuit for a column pair. In this circuit, one may opportunistically locate and remove identical, duplicate operators shared by the different decoders (or apply more involved optimizing transformations). Encoding may or may not be performed jointly also (“encoder fusion”), but that is of relatively minor concern.
2. **Derivative representation**: Some columns are compressed under the assumption that certain other related, uncompressed columns are available. At worst, this results in the independent compression scheme; hopefully, only little additional data is necessary over the pre-existing column. When decompressing, the base columns for the derivation, are decompressed first, realizing the assumption.

3. **Co-representation**: One chooses compression schemes for multi-column structures, e.g. pairs or tuples of several types — from the outset. The columns are also decompressed all together.

As one proceeds along this scale, compression ratio and speed for related columns together drops, but for the individual columns it mostly rises.

So as to limit the breadth of this document, we will not explore concrete multi-column schemes and different patterns of combining single-column schemes, except for a single simple case, immediately below. The reader is reminded that the general problem of optimally combining compression schemes for multiple columns is computationally intractable — just like determining the optimal compression scheme for a single column: These are the problems of computing the plain, conditional and/or shared Kolmogorov complexity of columns as words over a finite alphabet. For an in-depth (abstract) treatment of these fundamental theoretical problems, see [LV08].

**Full derivation** Real-life data tables very often have columns which can be perfectly inferred from other columns — even at the single tuple level. For example, a time-series table may have a *date* column, alongside *day_of_week*, or possibly *year*, *month* and *month* as separate columns (e.g. in [Sch]). More generally, such columns tend to show up in databases not in the Third Normal Form (3NF, see [Ken83]). A column $c$ is fully-derivable from a set $B$ of other columns if there is a columnar circuit which, for some assignment of columns from $B$ to its inputs, produces $c$. If, the circuit is just a lifted Elementwise $f$ operator, the column is elementwise-derivable from $B$.

The notion of full derivability can be abused if arbitrary operators are allowed in the circuit, as one can simply define an operator which outputs the desired column; but a constraint is difficult to formulate, as column stores may indeed generate operators dynamically. Elementwise derivability is a useful restriction of the general definition, both because the prevalence of such columns and because the elementwise-applied function is less amenable to such edge-case abuse. Typically, these functions correspond to multivariate arithmetic expressions with some deterministic mathematical functions available in SQL (such as \texttt{SIN()}, \texttt{SQR}() and the like).
Applying the model: Prospective notes

The columnar circuit computational model we have presented in this work is intended for analysis of existing column stores systems — but also as a foundation for developing new ones. This chapter focuses on what a column store might look like when designed with the computational model in mind, and which aims to support and benefit from features more consistently and fully. Chapter 3 explored one aspect of these prospects — the use of compression — in greater detail: This was a demonstration of how the model applies to a subject whose study so far has been somewhat fragmented and piecemeal; and which is possible to present in a a relatively self-contained manner. Other aspects of a column store system: Compilation, plan optimization, choices between representation schemes, mutability, execution flow and so on — would require further, more concrete, work to be explored with confidence. Instead, this chapter will touch on these aspects more speculatively, and in brief; these prospective notes may hopefully indicate paths towards actual implementation.

4.1 Minimizing materializations: Operator fusion via JIT compilation

A reader concerned with optimizing column store performance may become worried as they browse the previous chapters of this work: Ostensibly, this model suggests a column store materialize results after each and every operator in a circuit. Doing so is highly detrimental to query processing performance, and was a main motivation for the evolution of Vectorwise (initially “X100”) out of MonetDB [ZBNH05]. To make matters worse, the columnar circuits tend to be composed of simpler operators than in existing column stores — increasing the number of intermediate materializations even further.

This stands in contrast to the increasing prominence in recent years of DBMSes effecting compilation of query execution plans into native code: [Neu11] and the implementation in HyperDB [KNG11]; MS SQL Server’s Hekaton engine [DFI+13, §2.1.3], the MemSQL DBMS [CWJ+16, §1.1] and others (see also the discussion in [TER18]). While these are not column stores, the motivation is similar: Avoiding a multitude of separate operators applied to ever element or tuple, with unnecessary data movement back and forth. A larger plan made of more operators seems to be the opposite of desirable.
Paradoxically, the copiousness intermediate materialization is intended to achieve the opposite of its immediate effect: The adherence to simple, uniform data structures and the simplicity of the operators maximize execution plans' amenability to optimizing transformations. Many of these let a column store avoid some computation altogether, or use alternative approaches to the computation of a subcircuit; but after the system is done with these kinds of optimizations, numerous materializations still remain. It is important to have some final "bake-in" transformations of the resulting circuit: Inverting the decomposition impetus of our approach in previous chapters, these will take subcircuits made up of multiple simple operations and fuse them into single, complex, customized operators, which typically materializes only its final result.

We have, in fact, already described the formal machinery for effecting the above. In Section 1.3, we formalized the transformation of circuits by replacing an induced subcircuit with another circuit having the same function. We noticed the special case of "lifted operator" single-node circuits, and defined the fusion of a subcircuit into a lifted operator. There remains the question of where the implementations of these complex operators are to be found.

The simple, naive approach is for a large set of operators compiled into (optimized) machine-executable form to be determined statically, apriori: The DBMS’ executable files have their final implemented forms before any queries arrive; and complexities such as SQL UDFs are be handled by an interpreter (or an external compiler run [Raa18]). Indeed, pre-compilation of operators is the approach column stores have taken, by and large, thus far: MonetDB, Vectorwise and C-Store have all of their machine code compiled apriori, with no additional (“Just-in-Time”) compilation for individual queries.

But while for some column stores apriori-compiled operators may be a baseline option — with the columnar circuit model, and the compositional approach previous chapters have taken — this approach is a non-sequitor: The variety of potentially fused subcircuits is enormous. Furthermore, transformation rules and even some representation schemes may be devised with reliance on the elision of materializations. Thus it is absolutely necessary to generate operator implementations — by JIT compilation of a higher or lower-leve language, or by using compilation results of different pieces of code.

Let us consider a concrete example. A simple, archetypical scenario for compiling execution-plan-specific code is treated in [Neu11] (in a non-columnar setting; and it is implemented by the DBMS, HyperDB [KNG11]): A circuit includes two consecutive elementwise operators, Elementwise\(f\) and Elementwise\(g\); and the column store intending to execute this circuit has compiled implementations for both individual operators, not of Elementwise\(f\circ g\); but it is able to fuse the machine code for both operators, with the result being an operator applying \(f\circ g\) to the input. Or perhaps — neither Elementwise\(f\) nor Elementwise\(g\) is available, but the code for \(f\) and code for \(g\) on individual elements is, and the column store can generate a parallelized “for all elements” operator from the code for a scalar. Banking on this ability, the column store may replace the subcircuit containing Elementwise\(f\) and Elementwise\(g\) with the lifted single operator Elementwise\(f\circ g\), maintaining semantic equivalence and circuit output correctness.

There are, however, several important differences between compilation-based operator fusion in the columnar circuit model, and the JIT compilation in aHyPerDB-like system approach:

- **Non-tree circuits:** Columnar circuits are not, in the general case, decomposable into compilable pipeline-segments in the sense of [Neu11, §3.1]; specifically, columnar circuit layouts are DAGs rather than trees, so that the same intermediate column(s) may be used multiple times for different purposes.
• **Fusion after parallelization**: In HyPerDB, operator fusion is a per-record level concept (as in the example of elementwise operators above; although operators like Gather can also be thought of as being “per-record”). With our model, operators can have complex parallelization in their implementation, while still being relevant for fusion. We must therefore allow for more complex memory access patterns in “non-pipeline-breaking” operators (again in the sense of [Neu11, §3.1]) — as otherwise, many of the transformations presented in previous sections would preclude JITing: We would not be able to replace a non-pipeline-breaking operator with an implementation containing “pipeline-temporary-breakers”.

• Perhaps most importantly: In existing systems employing operator fusion (and in HyPerDB particularly), much of the computational work is not represented in execution plans: Application of statistical meta-data, combination of hot and cold structures into a single column, specifics of decompression, etc.; such work is therefore not amenable to fusion or JITing. Our model significantly increases the potential for fusion and JIT compilation of these kinds of work, especially as parts of overall query execution plans.

A related, but different, approach to operator fusion can be found in the more recent [FBN+18]. The differences are fundamental enough to make a comparison of this approach to HyperDB-style JIT’ing exceed the scope of this document (albeit interesting to consider). But this approach is less general than HyPerDB’s, which in turn is less general than fusion in our model.

Finally, recall that both before and after subcircuit contraction, a column store has a columnar circuit to execute. While such circuits expose the potential for parallelism relatively well, the exploitation of this potential on the hardware side is ever more difficult as one considers operators of increasing complexity, on different processing devices, with multiple devices operating at once and cluster-level parallelism. The more sophisticated this exploitation becomes, the more challenging it is to support effective operator fusion. Unlike the trivial case of elementwise work, it may require adherence to certain programming conventions; and perhaps also instrumentation of the compilation chain to keep track of how pieces of code can fit together. Consider the following prototypical example, regarding the complication of even simple considerations: Suppose an operator Op has a GPU implementation involves the execution of two distinct kernels (which cannot be fused together), while a previous Op′ in the execution plan, to be fused with Op, is implemented using just one kernel. Should the fused operator employ three kernels, or two? Or maybe it’s even just one, if the fusion obviates somehow the need for two kernels for Op? And if it’s two kernels, which of them will actually get the first operator’s code fused? Such complications abound.

### 4.2 The choice of compression schemes as a nexus of preprocessing effort

A column store with elaborate machinery for representation and compression of columnar data faces a challenge in choosing (or devising) compression schemes for its columns. Even for some single criterion for evaluating schemes, and even when capping the complexity of the scheme (e.g. in terms of compression and decompression circuit sizes and available operator set) — the task is already rather daunting. This is both due to the size of the search space — thousands upon thousands of potential schemes already with the above restrictions — and the difficulty of the very choice between pairs of candidate schemes.
For example, consider an column $c$ of element type $\tau_{\text{int}}$, which is close, but not identical, to an affine function, both in average distance and the maximum difference between elements — i.e. both by the $L^1$ and by the $L^\infty$ metrics; and suppose we wish to optimize the compression ratio, disregarding all other scheme features. Which compression scheme would yield a better ratio for column $c$: Delta, or a GENERATEDPOLYNOMIAL with degree 1 (modeling $i \mapsto a \cdot i + b$), with elementwise offsets? If $c$ is perfectly affine, both schemes would have essentially the same compression ratio: An all-zero column of elementwise differences vs. an all-zero column of elementwise offsets; both could use an extra composition with the CONSTANT scheme, which would make the representation “infinitely” efficient. However, we do have some differences from the affine function; which should we choose? The best ratio (ignoring the application of further compression schemes) would depend on the extremal offset values (for the noisy affine model scheme), and on the maximum second discrete derivative (for the Delta scheme). And even realizing that these parameters are to be examined — the dilemma is not nearly resolved, since composition of further compression must be considered. Specifically, if one examines the support sets for the first discrete derivative (i.e. the delta column) and for the offsets in the GENERATEDPOLYNOMIAL scheme, respectively — one of them may be small enough to make an application of DICTIONARY worthwhile; or the support set may mostly have weight on a small set of values, allowing for one of the approaches described in Section 3.5 for exploiting the distribution skew? And so on.

This confounding state of affairs, already within limited search spaces for schemes, has led other researchers to approach the selection of a compression scheme as DBMSes typically approach the choice of a query plan: Rule-based pruning and strategic selection of composition steps for improvement of metrics or achievement of features; see [FHL10, §4.1].

But let us venture even farther, and lift the caps on scheme complexity and consideration criteria. In previous chapters of this work we’ve described quite a few (structural) features of a column store which, without the restriction, may induce additional dimensions of complexity to the choice of compression schemes for the columns in a store:

**Compressed-form execution** A key desirable aspect of compression schemes used in DBMSes is their being “lightweight” — easy to decompress despite exhibiting attractive compression ratios. This has guided the entirety of our exploration of columnar compression schemes in Chapter 3 above. We have, however, hinted — and made fully explicit in Section 4.1 above — that we hope to avoid materializing intermediate results as much as possible; and decompressed columns are intermediate results of our columnar circuits when the inputs are compressed. In other words, we would rather avoid actually decompressing our columns.

Several existing column stores have a partial capability to do this, via the “push-down” of operators into decompression routines — so that what eventually gets materialized may be far smaller than the full decompressed column. Selection predicates (and, perhaps, aggregation) are “pushed down” in C-Store/Vertica [AMF06, §5.1], Vectorwise [ZB12, §2], MS SQL Server [LCH+11], and likely also other systems. This is usually (perhaps always) achieved via custom decompression routines supporting such push-down.

The expressivity of the columnar circuit model allows column stores to surpass push-down optimizations. Since decompression is merely another part of the larger circuit; and since the compression schemes devised in Chapter 3 are decomposable, and use simple, non-idiomatic operators as building blocks — it is often be able to conceptually commute the decompression with the following (parts of) the execution: Simple transformations will get us to a point where we apply
a circuit to the compressed form whose output is the compressed form of applying a subsequent subcircuit in the original plan; and a decompression follows afterwards.

A few examples: If a column is compressed in RunEncoding (or RLE, or RPE), any elementwise operation can be applied to the run_values column only, typically much shorter than the entire column; and a selection by predicate can be applied to the values, positions and lengths columns, which, after having some corresponding elements removed, will now describe a subcolumn, compressed in RunEncoding. Two columns compressed using Dictionary can be Join’ed by computing their individual histograms, Join’ing their dictionaries (resulting in two columns of indices into the original dictionaries, a-la-MonetDB), and using the result to generate the appropriate subcolumn of the cartesian-product histogram. These three columns constitute the Join result — compressed in the Dictionary scheme and with the original dictionary for each column.

Unlike in existing systems, the examples above will not result in convoluted custom operators, which are nearly impossible to involve in further optimization. Instead, it will likely be possible to repeat the commutative process, pushing the decompression further and further into the circuit. Eventually, it may pass some vertex-cut in the circuit, beyond which a full materialization of the original column is no longer necessary. When this happens, the execution plan will have avoided decompression altogether.

Expecting to perform this kind of compressed-form execution, a column store considering compression schemes may assign less significance to maximal decompression bandwidth of a candidate scheme — in favor of, say, better compression ratio. On the other hand, not every scheme is so “permeable” to execution order switching; so that the amenability to compressed-form execution itself becomes a desirable metric in the choice among schemes.

Multi-column decoder fusion consideration in scheme choice In Section 3.7 we listed several benefits of using joint compressed representations of correlated columns. Even if we refrain from considering them proper, the choice of individual column compression scheme may need to take inter-column correlation into consideration: The less the schemes chosen for two related columns have in common, the less likely the column store is to benefit from potential “decoder fusion”, i.e. avoiding the repetition of similar or identical work when decoding both correlated columns for the same query. Also, given a choice of scheme, we may prefer sub-optimal settings of its parameters — if these benefit decoder fusion with other columns. Examples could be a choice of segment length to fit the segmented representation of another column; choice of base functions and/or their coefficients for a NoisyGenerated representation; or reuse of another column’s dictionary despite it not being minimal for the reusing column.

Multiple (redundant) schemes for individual columns When faced with several alternative compression schemes for a column, each with different relative advantages and none being overall-optimal — why should a column store only choose one of them? True, holding redundant copies of a column goes against the basic rationale of compression — reducing the amount of memory necessary for the column; but we have already noted this is just one of multiple qualities of a given scheme. Moreover — while more memory would be needed for holding the column, the time required for reading the data from memory by a processor, or loading segments of it into smaller memory spaces, is not increased by the existence of a redundant copy. This possibility expands the representation/compress scheme search space further, now with combinations of multiple schemes translating into different sets of parameters of scheme desirability.
Use of multiple schemes is even more relevant in a distributed setting, where each node only holds some of the columns (or some segments of some columns). As the number of nodes in such a system increases relative to the overall data size, columns will gradually become replicated among the nodes — improving both the availability of nodes with relevant information (needing less initial disk or network I/O), and the reliability / fault-tolerance of the system overall. When this is the case, it becomes even more attractive to choose different schemes for different copies of the column: Such asymmetric replication allows the system to use different copies differently depending on the incoming queries and the availability of computational, memory and network bandwidth resources — without paying the single-node premium of holding multiple copies of the same column: The risk is having the distribution of column replica schemes not match the distribution of “desirable” schemes w.r.t. the query workload.

Such a choice is described in [RDS02, §3], albeit with regards to a secondary disk array: Instead of employing RAID mirroring as protecting from hard disk faults, one stores different representation schemes for the same data (in the case of [RDS02] it was DSM vs NSM representation of tables; see the discussion in [ABH+13, §2.2]).

It should be noted that assymetric replication is potentially less robust as a fault tolerance measure: In the symmetric case, we simply need to get the same piece of data from a different replica; if replication is assymmetric, however, restoring the part of a column representation stored at a certain node requires not just extra computational work, but possibly data from several distinct nodes.

**Multi-column schemes: Derivation, co-encoding**  As long as a column store has only one single representation per column, it is likely this will be a single-column, independent scheme; derivative or joint representation would be risky choices, considering the overhead of processing extra columns’ data when they are not even used in a query. But once we have opened the door to holding multiple redundant representations, or if we’re willing to adapt our schemes to the projected workload (see below), It becomes much more realistic for multi-column schemes to be used. This again expands the search space for compression schemes: In our description so far there has been a separate search spaces for each column (perhaps with some allowance for decoder fusion considerations); now it is a single search space, a choice of multiple schemes over all subsets of columns, with the constraint of their union covering every column.

**Indices: Ordered multi-column representations**  In general, table indices [AS10, §11] are data structures which make it easy to access table records according to a certain order of those records; and they are a key feature in non-columnar DBMSes, both for transactional and analytical queries.

In a column store, an index is essentially a copy of a sequence of columns, permuted into a lexicographic order of the induced records. But — that definition is only true for the uncompressed representation of the index; it would be quite wasteful to actually store a full copy: First, the permuted copy of a column shared its support and frequency distribution with the original column — which may well be enough to consider co-encoding them. Second, within every single column in the index, the lexicographic order induces long sorted stretches of elements, even if not all-column sortedness. Typically, this makes index columns compress very well with schemes such as RLE/RPE/RunEncoding, or something more complex — thus the overhead of keeping such sorted copies is limited. A specific (and useful) case of such index structures is the self-indexing of
individual columns, i.e. keeping an extra, sorted copy of a column irrespective of any other column. Compression schemes used on a column’s self-index provide information about a column which may be put to good use when planning queries, even if the sorted copy is not used instead of the column itself. When a self-index is compressed using RunEncoding or RLE, it provides us with a (sorted) representation of the column’s support set. This fact was used for query plan optimization in [AMP+16].

Columnar DBMSes differ on the extent of the use of indices: MonetDB only offers support for self-indexing of a single column, and ignores other SQL statements involving table indexes [Monb]). C-Store (and possibly, Vertica) takes the diametrically opposite approach: Its integration of indices is so deep, that in the papers introducing C-Store, the storage scheme is not even presented in terms of columns, but rather in terms of multiple (relational algebra) projections of tables onto subsets of columns [AMF06, §3], each with some specific sort order. When processing a query, C-Store determines which are the relevant projections to decompress (often only a single one), according to the scan orders useful for the query’s execution plan. These projections typically decompose into a “key” and a “value”, in which case the “key” columns are a (multi-column) self-index, and the “value” column is indexd by the multi-column ones. all but the last of the “key” columns in the projection typically exhibit long runs of identical values, due to the lexicographic sorting, and are thus efficiently compressed. However, C-Store also always keeps the compressed permutation of the records relative to some basic order, offsetting the saving in space somewhat. Also, other than the indices, the columns are compressed independently, so that correlation among the “non-key” columns of the projection is under-exploited [AMF06].

Note. Some non-columnar DBMSes now utilize the column-store form of an index as their own index, to speed up analytic work, employing some upcasting and/or a Dictionary-like compression scheme followed by an RLE-like scheme [LCH+11]. The acceleration of non-columnar DBMSes by columnar indices is also explored in [EHRB+11, JSD+13] and in [ABH+13, §4.8].

Workload dependence Adptation of the physical layout of data to the workload a system expects is both common and useful in row-oriented DBMSes, e.g. in the form of materializing table views and choosing indices/sort orders to index by [AS10, §24.1.6, §24.1.7]. Existing column stores can generally benefit from auxiliary, workload-tuned indexes, and some, like C-Store/Vertica, also have “column groups” [AMF06] which can be seen as a limited materialized views, and can be chosen according to an expected workload. This can well be done with a larger space of compression and representation schemes of (permuted sets of) columns: One alters the search or selection process to give more weight features which will benefit the processing of the expected queries with the available data (or expected future data), rather than trying to satisfy abstract general benefits.

A column store could also gradually adapt its choice of schemes to its workload, noticing when an alternative scheme it is now missing would better serve it, and switching at some threshold. This is easier if a column store holds multiple representations of columns, in which case it could “evaluate” schemes on actual data rather than only use analytic assumptions; and it does not have to discard all of its previously-held schemes, so that the threshold for adopting a scheme should not be as high. The practice of column cracking [IKM07] can be cast in terms of gradually refining a representation scheme of a column as multiple disjoint subcolumns (satisfying value range constraints), based on incoming range queries (and the prediction that future queries would be similar).
4.3 Port digraph grammars for optimization

A typical DBMS, columnar or otherwise, generates an initial execution plan from an SQL query, then proceeds to a distinct phase of plan optimization (in some systems these are two separated into two phases, optimization of the “logical” / relational algebra plan, and then the selection/optimization of a “physical” plan). [AS10, §12.1, §13]. These optimizations regard a set of operators which is fixed at compile-time, rather than variable and discovered at run-time; and so is the set of attributes, tags or meta-data which the system may compute. Under these circumstances, optimizers (especially for “physical plans”) tend to be implemented as general functions, implemented as part of the DBMS’ codebase, in the same programming language as the rest of its code, and with wide or unrestricted access to DBMS internals. For concrete examples in column stores, consider MonetDB’s default optimizer pipeline [Mon18]; or the compression-related optimizers of C-Store [AMF06, Fig. 3].

An alternative approach is exemplified by Spark SQL’s Catalyst optimization engine [AXL+15]. Spark SQL is not a column store, and its data model is quite dissimilar to the column-based model in this work; but the developers of Spark SQL’s optimization engine (named Catalyst) prioritized for extensibility and external-developer involvement. They therefore designed the engine to expose the query execution plans to user-supplied optimization rules. Catalyst takes “rules” which interact with the plan in a rather domain-specific fashion [AXL+15, §4]; and the results of rule application is eventually JITed along with the Scala code in which the engine is written. However, these “rules” are still really full-blown Scala functions [AXL+15, Pha] which the column store executes.

In a column store following the columnar circuit execution model, execution plans are much more expressive, both in capabilities and in practical use: previous chapters (and sections in this chapter) demonstrate how various structural features of today’s column store can “implemented” through columnar circuit execution plans. This implies more (or most) optimization work will involve straightforward circuit transformations; it is partly for this reason that we lengthily and carefully exposited circuit transformations in Chapter 1, even beyond what later chapters required. If we also allow for decorating vertices (operators), ports and edges with some property labels that optimization rules can match, the expressivity, and the coverage of optimization rules expands even further.

It is at this point that the need for utilizing a more general-purpose port-graph rewriting system, or (port-)graph grammar, becomes apparent: Without apriori knowledge of the port-digraphs we need to work on, nor of the set of usable vertices (operators), nor of the set of usable transformation rules — we require a mechanism for repeated rule selection and application, coupled with an abstract, general representation of what a transformation rule constitutes. With our model lending itself towards larger graphs and further decomposition than other column stores, match patterns will require some flexibility and may be larger sometimes than two or three vertices. The automation of rule consideration, prioritization, matching and application should prove to be quite challenging to implement efficiently (despite being computationally tractable). The choice of an appropriate formalism for is not immediately obvious, especially considering the variety of established models (see [Roz99, Vol. 1]) on one hand, and with our interest in rewriting rather than derivating from scratch or identifying a language. Going into any detail on this matter, however, would be well beyond the scope of this work.
4.4 Mutable columns and column store write support

The columnar circuit model regards columns as immutable. Rather than columns changing, “new” columns are the result of an application of operators to existing ones, within a circuit. This restriction makes for a stark contrast with the extensive work on mutable, transactional DBMSes, both as practical implemented systems and as objects of theoretical study.

Could we complicate the model to also allow for column mutation? This is, perhaps, conceivable: Changes to input columns would propagate into the circuit (e.g. through an overlaying standard-representation subcolumn, see Subsection 2.2.2) — with operators having to either re-execute, or have a “delta version” execute on the original inputs and outputs, along with the changes. However, such a complication is probably too unwieldy to be worthwhile.

A more palatable alternative is for a column store utilizing the columnar circuit model to allow mutation, but have circuits isolated (in the ACID sense [HR83]) from mutations’ effects. Such isolation would be achieved if representations of mutated columns each contained a complete representation of the original column, which can remain intact (and in-place in physical memory), even as the column mutates. A system applying this principle to each and every change a column undergoes must necessarily utilize versioned data structures (a.k.a. persistent data structure for its columns — as defined in [DSST86], and used extensively in some non-columnar DBMSes, such as LogicBlox [AtCG15]).

For the purpose of an analytics-focused column store, however, we can make do with much weaker “isolative” nature of a column’s representation — and we need look no further for it than the schemes already presented in previous chapters. Recalling the discussion in Subsection 2.2.3, we can adapt SUBCOLUMNOVERLAY, so that an overlaying subcolumn holds appended (or modified) elements; and deleted elements can be represented using an additional index subset representation (see Subsection 2.4.1). Alternatively, the deletions could also be represented by a subcolumn — to be “subtracted” from the SUBCOLUMNOVERLAY-encoded column; this approach would be similar to that of MonetDB [Bon02] (see also the concise description in [HZN10, §2]): Separate the main read-only data structure for a column (or a projection of columns) from an auxiliary but still columnar structure for insertions and deletions. Occasionally — either every period of time or when changes exceed a certain size — column data will be re-integrated by reconstructing a complete and up-to-date column with no mutations, overlaid by empty insertions and deletions subcolumns.

Other existing column stores use non-columnar representations (delta-structures) for column mutations. SAP HANA, for example, uses two-level structures situated in CPU cache: A row-major delta-table in L1 cache, and a column-major delta table in L2 cache; when the former fills up, it is merged into the latter, and when the latter fills up, it’s merged into the main columns in main memory [SFL12]. Another prominent delta-structure design are the B-tree-like positional delta trees (PDTs) used in Vectorwise [HZN10], which take more after index structures in row-oriented DBMSes rather than actual tables. While these may be faster for transaction processing workloads, they would be more difficult (if not impossible) to make usable with columnar circuits for analytic queries. This difficult is also an explanation of C-Store’s choice: Its columns are a Read Store, and a separate Write-Store — for inserts, updates and deletions — is handled by a separate non-columnar DBMS — external to C-Store proper — which uses B-trees under-the-hood [SAB05, §6].
4.5 Execution progression and residual plans

The columnar circuit model abstracts away — at least prima facie — most aspects of execution on real machines. The following, at least, ones must somehow be accounted for when actually computing the output of a circuit (“executing” the circuit):

**Operator execution status** Columnar circuits do not define an order of execution for their constituent operators. Operator execution is not instantaneous, and multiple operators may execute concurrently. How should a column store, adhering to the columnar circuit model, keep track of which operators have concluded execution; which have not started execution; and which are currently executing?

**Multiple finite memory spaces** Memory is a limited resource, and the faster it is, the less of it real-life machines have. Even if we choose to treat faster memory as cache, and consider secondary storage (e.g. magnetic or flash drives) to be “large enough” to hold all relevant data — we would still not be able to fit the memory spaces on co-processors into a uniform, single-space view of memory. Also, performance considerations demand that column stores take great care to effectively utilize their small, faster memory. How would a column store reconcile this necessity with the huge uniform columns in the circuits it is to execute/simulate?

**Sub-operator-resolution scheduling** Simpler column store execution engines use the *operator-at-a-time* model for plan execution [MKB09]: A plan operator is executed as a whole; there are no multiple recognized states before its execution completes. A prominent example is MonetDB [Monc], which executes MAL instructions in sequence (see Figure 1.3b; but note that the multi-threaded capability makes MonetDB more of a per-thread operator-at-a-time system [IKG12, §3]). This can indicated trivially by decorating the nodes of a columnar circuit.

Other column stores use finer-resolution scheduling: They operate on relatively small chunks of a column (called “vectors” in Vectorwise [ZB12]; and “morsels” in a further development of this notion in [LBKN14]). For multiple reasons, this choice of chunk-at-a-time execution is quite beneficial, performance-wise, at least on CPUs. However — it is unreasonable to make column-chunks into “first-class citizens” of execution plans, as that would mean plan sizes would be huge — linear in the size of the data. In Vectorwise, and likely in other chunk-at-a-time systems, chunks are not represented in the plan; instead, the column store’s execution engine has the “hard-wired” capability of conceptually breaking columns up into chunks, keeping track of the different chunks, and recombining them after processing into a full column when necessary.

More generally, chunk-at-a-time execution seems to be paired with some idiosyncratic internal representation of query plan execution state. This implies that complex patterns of composition and de-composition of execution — the very ones which are core to the columnar circuit model — do not apply to this finer-resolution representation. The idiosyncracy also implies that very little optimization is possible on the currently-executing parts of the plan — as the column store’s optimizer mechanism is not aware (or not more than superficially aware) of the out-of-plan execution state information.
4.5.1 Splitting operators and circuits

It so happens, that all three challenges described above can be addressed using a single “silver bullet”: A well-developed capacity for splitting. The possibility of splitting an operator and its inputs is not fundamental to the columnar circuit model, and we have barely used it so far in this work (see Subsection 3.4.3 for an exception); but no column store implementation can avoid it.

Splitting operator input columns is less trivial than it seems. First note that a split of a column can result in two or many (subcolumn) parts; the part sizes can be uniform or small “chunk” vs large “rest”; and the resulting subcolumns can be contiguous or form a complex non-contiguous pattern. For a column unto itself, this is still perfectly trivial; and so is the case for splitting, say, an Elementwise operator. But as the operator’s semantics are more complex, so does the splitting of its input become. A few examples:

- **PrefixSum** (see Subsection 3.2.2): A typical implementation of this operation — and any reduction — is itself based on breaking up the column into smaller chunks: Compute a sum of the elements in the chunk; (recursively) compute a prefix sum of the chunk-sums, then, for each chunk add the sum of preceding chunks to the prefix sum within the single chunk. Partitioning the input column “outside” the operator merely adds another level of this procedure.

- **Sort** (see Subsection 3.1.2): Similarly to PrefixSum, the computation on each part is a smaller Sort; however, the combination of the parts is more involved — a merging of sorted sequences. Alternatively, one could go through making the two parts into bitonic sequences, then finally merging them into a proper sorted sequence (see [SHG09] regarding the relevance of bitonic sorting in parallel settings).

- **Gather** (see Subsection 3.1.2): While this is not an elementwise operator, it is elementwise operator with respect to its pos input, but not with respect to the data input. If we were to split up data — say, into the first and second halves — we could not then simply employ two Gather’s with the two halves (data$_1$, data$_2$), since some of the indices in pos may be out-of-range; and for the second half, all indices would need to be adjusted. One straightforward approach to handling these difficulties would be splitting pos into two as well, using a filter computed by Elementwise$_{\geq|\text{data}|/2}$ (the split may not be into halves of course); we then have to subtract $|\text{data}_1|$, elementwise, from the elements of the second part of pos. Finally, the two Gather-resulting columns need to be re-integrated using the filter vector. (Note this approach is not the only one possible.)

How far do can these split recipes generalize? While they cannot exist for an arbitrary choice of input of any conceivable operator — they do exist essentially for all inputs of the operators we have been using in this work (see the Index of Operators). Moreover, these splits are nothing other than columnar circuit transformation rule with certain specific features; and a simple, localized rule at that. Of course the question of when these transformations are to be applied is a different matter: Some are relevant before execution has occured, but many become relevant during execution, repeatedly. Also, the decision to apply one of these requires additional information besides the circuit itself, e.g. information regarding actual column length, execution status and so on. More on this in subsection below.

Splitting single operators also generalizes to subcircuits, i.e. instead of duplicating a single operator, the split inputs of a subcircuit are fed to two, potentially-but-not-necessarily-identical,
circuits. This kind of subcircuit-split was used in the system described in [AMP+15] for scheduling parts of the data on different hardware devices and/or different threads. But perhaps a more prominent example is MonetDB’s mechanism for multi-threaded execution, named “Mitosis”: Some (easy to split) columns are split up, one part per each working thread, while others are untouched; and the MAL execution plan (see the example in Figure 1.3b) has some of its instructions replicated for each of the split column parts. This is performed as one of MonetDB’s plan optimizers [Mon18], [IKG12, §3].

A precursor to the suggestions here is the effect by the author and others of splits and merges via execution plan transformations in [AMP+15, AMP+16]. It is also worth noting that MonetDB’s multi-threading is heavily based on involves splitting columns, with the parts visible in the execution plan — a procedure named “mitosis” and constituting an execution plan optimizer [IKG12, §3], [Mon18].

4.5.2 Putting splits to use

With subcircuit splits achievable through a circuit transformation (and no “unsplittable” operators in use) — any circuit is just a few transformations away from having an operator with small enough inputs to schedule a chunk for execution: At most one split transformation per input of the chosen operator. Similarly, the circuit is a few transformations away from having chunks of input columns which can be transmitted to a different memory space where another processor can access them and start working on a chosen operator. In both cases, the splitting can be expanded so that a larger subcircuit may execute using the split-off inputs; also in both cases, the actions above can be repeated several times to allow for concurrent execution on multiple chunks of the data, either using the same initial memory space or in different spaces. An example of both of the above would be concurrent execution of an operator on different chunks by different CPU threads and by a discrete GPU accessing chunks placed in its global memory. (A more simplistic form of such a mechanism was used in the AXE heterogeneous execution framework [AMP+15]).

In other words: Using splits, we reduce the challenge of executing circuits a-chunk-at-a-time to the more straightforward operator-at-a-time execution: Operators are only ever executed on input columns which are small enough not to merit further breakup, or tracking of the execution beyond the “awaiting-/during-/after- execution” trichotomy of operator-at-a-time column stores.

It should be stressed that the splitting described above, even when applied repeatedly, does not make the circuit balloon in size, to comprise the huge number of small column-chunks and small operators which end up being scheduled distinctly for execution: Only the single small part which is to be executed next is split-off; most of each column remains in a single large (sub)column; and the chunks already processed are also not kept separate — they too are consolidated occasionally into larger partial-result columns.

Finally, to address the remaining challenges listed at the beginning of this section, we can decorate the nodes and ports involved in splits, to indicate which column lies in which memory space, as well as the “awaiting-/during-/after- execution” status of each operator. Without the splits, each node and port would require complex information regarding what has happened to which part of it; the splits allow for the subsumption of execution engine idiosyncratic complexity by the complexity of of representation schemes for columns and subcolumns, which a column store must already contend with.
4.5.3 Residual state and a streamlined execution flow

A side-effect of representing ongoing execution state within the execution plans (with the modification of allowing vertex and port tags) is that doing so is essentially maintaining an residual plan: A plan whose execution, given the various columns currently available to a column store, produces the same results as the original query. The initial residual plan when execution commences is the execution plan originally drawn-up; while the final residual plan simply has the final results as an input with no further execution necessary.

Residual plans, being (decorated) columnar circuits, are subject to the same kind optimizing transformations as initial execution plans (and see also Section 4.3). Now, this is also the case, essentially, for operator-at-a-time column stores like MonetDB; but with our model, plan optimizers or optimizing transformation grammars must already be tolerant of columns being being in a broken-up state, represented as multiple subcolumns with their own potentially different representation scheme, and requiring a circuit of computations just to re-materialize again. Thus there is more potential for optimization to be applicable during execution.

Considering our discussion above of the progression of execution using splits, and the potential for dynamic residual-plan optimization, we can conceive of the following rough outline for a column store’s plan execution step:

- **Feeding the processing hardware**: If some processor (CPU, GPU, FPGA etc.) is running low on scheduled work,
  - Find an unscheduled operator with small enough input & output sizes, which are all accessible by said processor, and schedule it.
  - If none was found, find such an operator but without the restriction on input & output sizes, which could be split up and scheduled
  - If one was found, apply a split transformation.
  - If none was found, search for a set of columns to move into to a memory space directly accessible to the processor, and which would be schedulable
  - If no such set of columns was found, search for a set of columns which could be split up (or their generating operators split up) so that it would constitute a set of inputs for an operator schedulable on the processor

- **Feeding communication channels**: If some channel or bus is running low on scheduled transfers:
  - Find a column which one can estimate would could be copied or moved directly over the channel into a more beneficial location.
  - If one was found, schedule it for transfer

- **Acknowledging conclusion of computational work**: If some operator has completed its execution,
  - Remove the operator its outputs become assigned inputs of the circuit. Unused inputs of the deleted operator are marked unused (a later optimization may remove them).
  - Create new circuit inputs and assign the operator’s outputs to them.

- **Optimization**: Search for an optimizing transformation which could apply to the (residual) circuit. (The search could be longer and/or produce multiple optimizations; or may be timed and shorter, so as not to neglect minding the hardware, and continue the search at a later iteration.)

This procedure is already attractively succinct, considering how it covers decompression, use of index structures, parallel processing on heterogenous hardware, dynamic reoptimization, some
memory management (through the deletion of unused nodes) and so on. We can go even further, however, in streamlining the execution step: Recall that most of the conditions-and-actions above are nothing but applications of specific (sub)circuit transformation rules. They may be more “structural” than “algorithmic”; there may be some importance to their relative order, or they may otherwise inter-relate — but then, this can be true for other transformations. We only need to extend our node and port tag sets, to cover not only execution status but location: Which processor is executing an operator, in which memory space a column is located and where it is being transmitted. With this modification, the execution step can be described much more generically:

1. Determine out-of-circuit system changes not yet reflected in the plan (e.g. an operator having completed its execution or buffer transfer complete).
2. Update the residual plan (= residual circuit) to reflect the changes.
3. Update the (possibly-implicit) space of applicable circuit transformations, including transformations scheduling operator execution or transfers. The update includes the prior probability distributions of benefit for these transformations.
4. Search, or continue a search, within the space of applicable circuit transformations (including unused input removal, execution or transfer scheduling, a split of an operator or a subcircuit, etc.), and possibly choose to apply one to the plan.
5. Schedule executions or transfers according to the changes in the residual plan.

In this formulation, multiple generic steps will be taken to cover the different actions in one single step of the longer, more specific one formulated above.
Bibliography

[AB09] Sanjeev Arora and Boaz Barak. Computational complexity: A modern approach, 2009.

[ABH+13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idr eos, Samuel Madden, et al. The design and implementation of modern column-oriented database systems. Foundations and Trends in Databases, 5(3):197–280, 2013.

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In Proceedings of the International Conference on Data Engineering (ICDE), pages 466–475. IEEE, 2007.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 671–682, 2006.

[AMP+15] Adnan Agbaria, David Minor, Natan Peterfreund, Ofer Rosenberg, Eyal Rozenberg, and Roman Talyansky. Towards a real GPU-speedup in SQL query processing with a new heterogeneous execution framework. In GPU Technology Conference (GTC). NVIDIA, 2015. https://www.gputechconf.com/.

[AMP+16] Adnan Agbaria, David Minor, Natan Peterfreund, Ofer Rosenberg, and Eyal Rozenberg. Overtaking cpu dbmses with a gpu in whole-query analytic processing. In Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems using Modern Processor and Storage Architectures (ADMS), 2016.

[AS10] S. Sudarshan Abraham Silberschatz, Henry F. Korth. Database System Concepts. McGraw-Hill, 6th edition, 2010.

[AtCG+15] Molham Aref, Balder ten Cate, Benny Green, Todd J.and Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 1371–1382. ACM, 2015.

[AXL+15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 1383–1394. ACM, 2015.

[Bat79] Don S. Batory. On searching transposed files. ACM Transactions on Database Systems (TODS), 4(4):531–544, 1979.

[Bon02] Peter Boncz. Monet: A next-Generation DBMS Kernel for Query-Intensive Applications. PhD thesis, Universiteit van Amsterdam, 2002.

[CJW+16] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. Proceedings of the VLDB Endowment, 9(13):1401–1412, Sep 2016.
George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Record*, 14(4):268–279, May 1985.

C Cherry, M. H. Kubba, D. E. Pearson, and M. P. Barton. An experimental study of the possible bandwidth compression of visual image signals. *Proceedings of the IEEE*, 51(11):1507–1517, 1963.

Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with Roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 3rd edition, 2009.

Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

Carl De Boor. *A practical guide to splines*. Springer, New York, revised edition, 2001.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1243–1254, New York, NY, USA, 2013. ACM, ACM.

Patrick Damme, Dirk Habich, Jiana Hildebrandr, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2017.

Wei Ding and Bede Liu. Rate control of MPEG video coding and recording by rate-quantization modeling. *IEEE transactions on circuits and systems for video technology*, 6(1):12–20, 1996.

J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *Proceedings of STOC*, pages 109–121, New York, NY, USA, 1986. ACM.

George F. Estabrook and Robert C. Brill. The theory of the TAXIR accessioner. *Mathematical Biosciences*, 5(3):327–340, 1969.

Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment*, 9(12):960–971, 2016.

Amr El-Helw, Kenneth A Ross, Bishwaranjan Bhattacharjee, Christian A Lang, and George A Mihaila. Column-oriented query processing for row stores. In *Proceedings of the international workshop on Data Warehousing and OLAP (DOLAP)*, pages 67–74. ACM, 2011.

Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, volume 2, 2018.

John T Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

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

T. Ferguson and J. Rabinowitz. Self-synchronizing Huffman codes. *IEEE Transactions on Information Theory*, 30(4):687–693, 1984.

Clark D French. Teaching an OLTP database kernel advanced datawarehousing techniques. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 194–198. IEEE, 1997.
[GAT+15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidiya Srinivasan. Amazon redshift and the case for simpler data warehouses. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 1917–1923. ACM, 2015.

[GS91] Goetz Graefe and Leonard D. Shapiro. Data compression and database performance. In Proceedings of the Symposium on applied computing (SAC), pages 22–27. IEEE, 1991.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. The Bell System Technical Journal, 29(2):147–160, April 1950.

[HBB+12] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gânceanu, and Marc Nunkesser. Processing a trillion cells per mouse click. Proceedings of the VLDB Endowment, 5(11):1436–1446, 2012.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR), 15(4):287–317, 1983.

[HT71] Te C. Hu and Alan C. Tucker. Optimal computer search trees and variable-length alphabetical codes. SIAM Journal on Applied Mathematics, 21(4):514–532, 1971.

[Hu82] T. C. Hu. Combinatorial Algorithms. Addison-Wesley, 1982.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the Institute of Radio Engineers (IRE), 40(9):1098–1101, 1952.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 543–554. ACM, 2010.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two decades of research in column-oriented database architectures. IEEE Data Engineering Bulletin, 35(1):40–45, 2012.

[IKG12] Milena Ivanova, Martin Kersten, and Fabian Groffen. Just-in-time data distribution for analytical query processing. In Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS), pages 209–222. Springer-Verlag, 2012.

[IKM07] Stratos Idreos, Martin Kersten, and Stefan Manegold. Database cracking. In Proceedings of the Conference on Innovative Data Systems Research (CIDR), volume 7, page ???, 2007.

[Ive62] Kenneth E. Iverson. A Programming Language. In Proceedings of the Spring Joint Computer Conference (JCC), pages 345–351, New York, NY, USA, 1962. ACM.

[JSD+13] Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich, Karen Khachatryan, and Alexander Bunte. How Achaeans would construct columns in Troy. In Proceedings of the Conference on Innovative Data Systems Research (CIDR), volume 13, pages 6–9, 2013.

[Ken83] William Kent. A simple guide to five normal forms in relational database theory. Communications of the ACM, 26(2):120–125, 1983.

[KJ10] Zvi Kohavi and Niraj K. Jha. Switching and Finite Automata Theory. Cambridge University Press, 2010.

[KNG11] Alfons Kemper, Thomas Neumann, and D. Garching. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Proceedings of the International Conference on Data Engineering (ICDE), 2011.

[KSS18] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. Algorithmic Aspects of Parallel Data Processing, volume 8. 2018.
[TER18] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In Proceedings of the 2018 International Conference on Management of Data, pages 307–322. ACM, 2018.

[Thea] The MonetDB DBMS source code. https://github.com/MonetDB/MonetDB.

[Theb] The MonetDB type system. https://www.monetdb.org/wiki/MonetDB_type_system.

[Tit97] Mark R Titchener. The synchronization of variable-length codes. IEEE Transactions on Information Theory, 43(2):683–691, 1997.

[Tra17] The Transaction Processing Council. TPC Benchmark H (rev 2.17.3), 2017. http://www.tpc.org/tpch.

[WS11] Haizhou Wang and Mingzhou Song. Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. The R journal, 3(2):29, 2011.

[ZB12] Marcin Zukowski and Peter Boncz. Vectorwise: Beyond column stores. IEEE Data Engineering Bulletin, 35(1):21–27, 2012.

[ZBNH05] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 — A DBMS in the CPU cache. IEEE Data Engineering Bulletin, 28(2):17–22, 2005.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM–CPU cache compression. In Proceedings of the International Conference on Data Engineering (ICDE), pages 59–59. IEEE, 2006.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. IEEE transactions on Information Theory, 24(5):530–536, 1978.
# List of Figures

1.1 Illustration of an operator as a single vertex within a columnar circuit .......... 9  
1.2 A port digraph ................................................................. 12  
1.3 A possible query execution plan for query Q6 of the TPC-H benchmark .......... 13  
1.4 Resolution of the function computed by a columnar circuit ......................... 15  
1.5 An induced columnar subcircuit ............................................. 19  

2.1 The \textsc{Indexed} column representation scheme .......................... 24  
2.2 The standard subcolumn representation scheme ................................. 25  
2.3 An overlay of partially-disagreeing subcolumns .................................. 26  
2.4 A union of compatible subcolumns ............................................. 26  
2.5 Decoders for representation schemes involving subcolumn combinations ......... 28  
2.6 The \textsc{UniformlySegmented} representation scheme, \(\ell\)-segmented views and nearly-matrices ................................................................. 30  
2.7 Rearranging data in a segmented view: Segment length change vs. transposition .. 31  
2.8 Replication in the two dimensions of a segmented view .......................... 32  
2.9 A subcolumn encoded with the \textsc{SegmentedSubcolumn} scheme ............... 33  
2.10 Decoder for the \textsc{SegmentedSubcolumn} representation scheme .......... 33  
2.11 Operator segmentation example .............................................. 34  
2.12 The \textsc{DenseIndexSet} representation scheme ............................ 35  
2.13 Materialization of an index set partition into the \textsc{Partitioned}\(_k\) scheme ...... 37  
2.14 Difference between the \textsc{Assemble} and \textsc{ComposeSegments} operators .... 39  

3.1 An example of patching with simple inputs .................................... 51  
3.2 A decoder for patched columns using \textsc{PatchPositionsScheme} for patches .... 51  
3.3 Decoders for columns compressed with two alternative schemes ................. 53  
3.4 The \textsc{VariableLengthDictionary} compression scheme ........................ 55  
3.5 A column and its compression with \textsc{RunEncoding} and related schemes ......... 58  
3.6 Compressor and decompressor for the \textsc{RunPositionEncoding} scheme .......... 58  
3.7 The \textsc{NaiveDelta} compression scheme .................................... 62  
3.8 Decoder and subcircuit implementations for the \textsc{CascadedDictionaries} compression scheme ................................................................. 68
Index of computational operators

Assemble, 38, 39
ComposeSegments, 38, 39
Concatenate, 28, 58, 62
Derivative, 49
Elementwise, 10, 13, 15, 19, 36–38, 44, 58, 62, 74, 76, 85
Gather, 46, 58, 77, 85
Integral, see PrefixSum
Iota, 23, 44
Join, 46, 54, 56, 79
Length, 15, 19, 23, 35–37, 58, 62
Partition, 46
Permute, 23, 34
PrefixSum, 49, 58, 62, 85
ReplicateSegments, 31, 32
Replicate, 10, 15, 19, 31, 32, 35, 37, 62
Scalar, 15, 19, 35, 37, 62
Scatter, 27, 28, 35, 46
SelectIndices, 35, 37
Select, 13
Sort, 46, 85
Split, 58, 62
SubcolumnOverlay, 50
Transpose, 30, 31, 38
# Index of representation and compression schemes

Listed here are representation schemes for column, subcolumns and index subsets.

| Representation Scheme                          | Page Numbers             |
|----------------------------------------------|--------------------------|
| BITMAP, see ValueIndicators                  |                          |
| CappedRLE, see CappedRunLengthEncoding       |                          |
| CappedRunLengthEncoding                      | 57, 59                   |
| CappedWidth                                  | 41                       |
| CascadedDictionaries                          | 67–69                    |
| ColumnByOverlay                              | 27                       |
| CommonPrefix                                 | 70                       |
| CommonUpperHalf                              | 70, 71                   |
| ComplementingSubcolumns                      | 27, 28, 37, 42           |
| Components                                   | 38, 39                   |
| ConcatenatedComponents                        | 39                       |
| Constant                                     | 42, 44, 45, 56, 59, 60, 78|
| ContiguousIndexSet                           | 36                       |
| Delta                                        | 62, 63, 66, 68, 78       |
| DenseIndexSet                                | 35, 37                   |
| Dictionary                                    | 45–47, 54, 63, 64, 67, 69, 71, 78, 79, 81 |
| DiscardHighBits                              | 45, 46, 48, 57, 71       |
| DisjointSubcolumnUnion                       | 27, 28, 71               |
| EquiknottedSpline                            | 60, 61                   |
| FOR, see FrameOfReference                    |                          |
| FrameOfReference                             | 61, 66, 71               |
| GeneralizedSpline                            | 59, 60                   |
| GeneratedPolynomial                          | 44, 78                   |
| Generated                                    | 44, 48, 59               |
| ImplicitDenseIndexSet                        | 36                       |
| Indexed                                      | 23–25                    |
| KnottedSpline                                | 60                       |
| MDICT, see MonotoneDictionary                 |                          |
| MonotoneDictionary                           | 47                       |
| NS, see DiscardHighBits                      |                          |
| NaiveDelta                                   | 61, 62                   |
| NoisyGenerated                               | 48, 62, 79               |
| NullSuppression, see DiscardHighBits         |                          |
| OverlaidColumn                               | 27                       |
| PartitionedIndexSet                          | 37                       |
| Partitioned                                  | 36                       |
| Partition                                     | 37, 52, 53               |
| PatchedDelta                                 | 63                       |
| Patched                                      | 27, 42, 50, 51           |
| PeriodicallyVariableWidth                    | 72, 73                   |
| RLE, see RunLengthEncoding                   | 59                       |
| RPE, see RunPositionEncoding                 | 59                       |
| RunEncoding                                  | 56–59, 65, 79–81         |
| RunLengthEncoding                            | 52, 57–59, 65, 70, 79–81 |
| RunPositionEncoding                          | 57–59, 79, 80            |
| Segmentation                                 | 29, 40, 41, 50, 53       |
| SegmentedSubcolumn                           | 32, 33, 53               |
| Segmented                                    | 29, 34                   |
| Shattered                                    | 39, 40                   |
| SparseIndexSet                               | 35, 69                   |
| Spline                                       | 59, 60                   |
| SubcolumnDictionary                          | 68, 69                   |
| SubcolumnOverlay                             | 27, 28, 67, 69, 83       |
| SubcolumnUnion                               | 27                       |
| Subcolumn                                    | 24                       |
| UDICT, see UniqueDictionary                  |                          |
| UniformSegmentDictionaries                   | 63, 64                   |
| UniformSegmentation                          | 29, 30                   |
| UniformlySegmented                           | 29–34                    |

97
UniqueDictionary, 47
VWDICT, see VariableWidthDictionary

ValueIndicators, 39
VariableWidthDictionary, 54, 55