SCIENCE
PASSION
TECHNOLOGY

Architecture of DB Systems
05 Compression Techniques

Prof. Dr. Matthias Boehm

Technische Universität Berlin
Faculty IV - Electrical Engineering and Computer Science
Berlin Institute for the Foundations of Learning and Data
Big Data Engineering (DAMS Lab)

Last update: Dec 02, 2023

Announcements/Org

- **#1 Lecture Format**
  - Introduction virtual, remaining lectures blocked **Dec 04 - Dec 07**
  - Optional attendance
  - **Hybrid**, in-person but live-streaming / video-recorded lectures
    - **HS i10** + Zoom: https://tu-berlin.zoom.us/j/9529634787?pwd=R1ZsN1M3SC9BOU1OcFdmem9zT202UT09

Agenda

- **Motivation and Terminology**
- **Compression Techniques**
- **Compressed Query Processing**

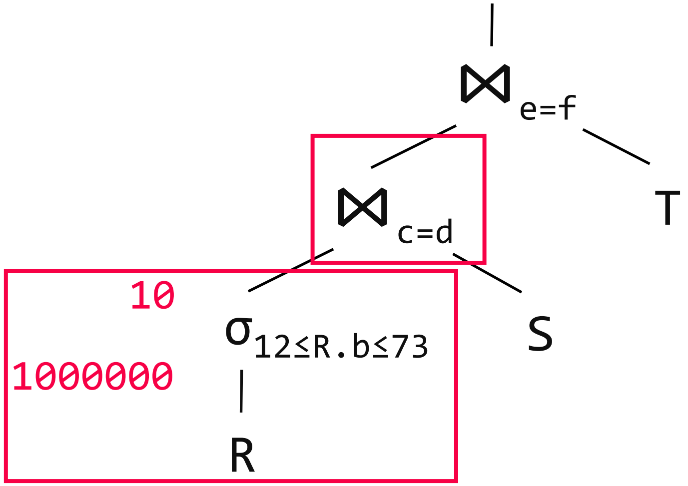
Motivation and Terminology

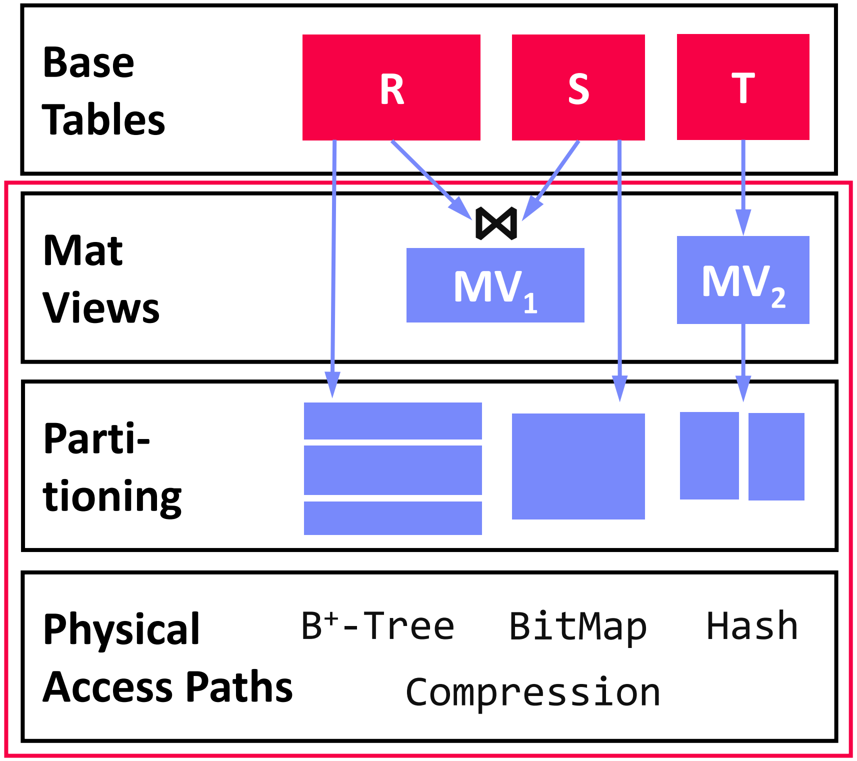
Recap: Access Methods and Physical Design

- **Performance Tuning via Physical Design**
  - Select physical data structures for relational schema and query workload
  - **#1:** User-level, manual physical design by DBA (database administrator)
  - **#2:** User/system-level automatic physical design via advisor tools

- **Example**

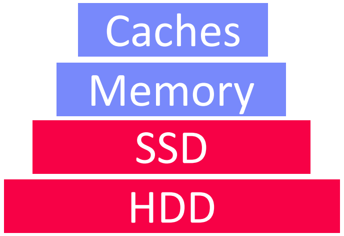
```
SELECT * FROM R, S, T
  WHERE R.c = S.d AND S.e = T.f
    AND R.b BETWEEN 12 AND 73
```





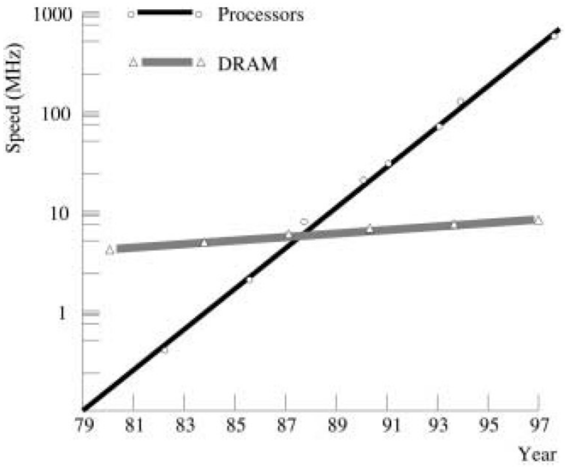
Motivation Storage Hierarchy

- **#1 Capacity**
  - Limited capacity of fast storage
  - Keep larger datasets higher in storage hierarchy
  - Avoid unnecessary I/O



Caches
Memory
SSD
HDD

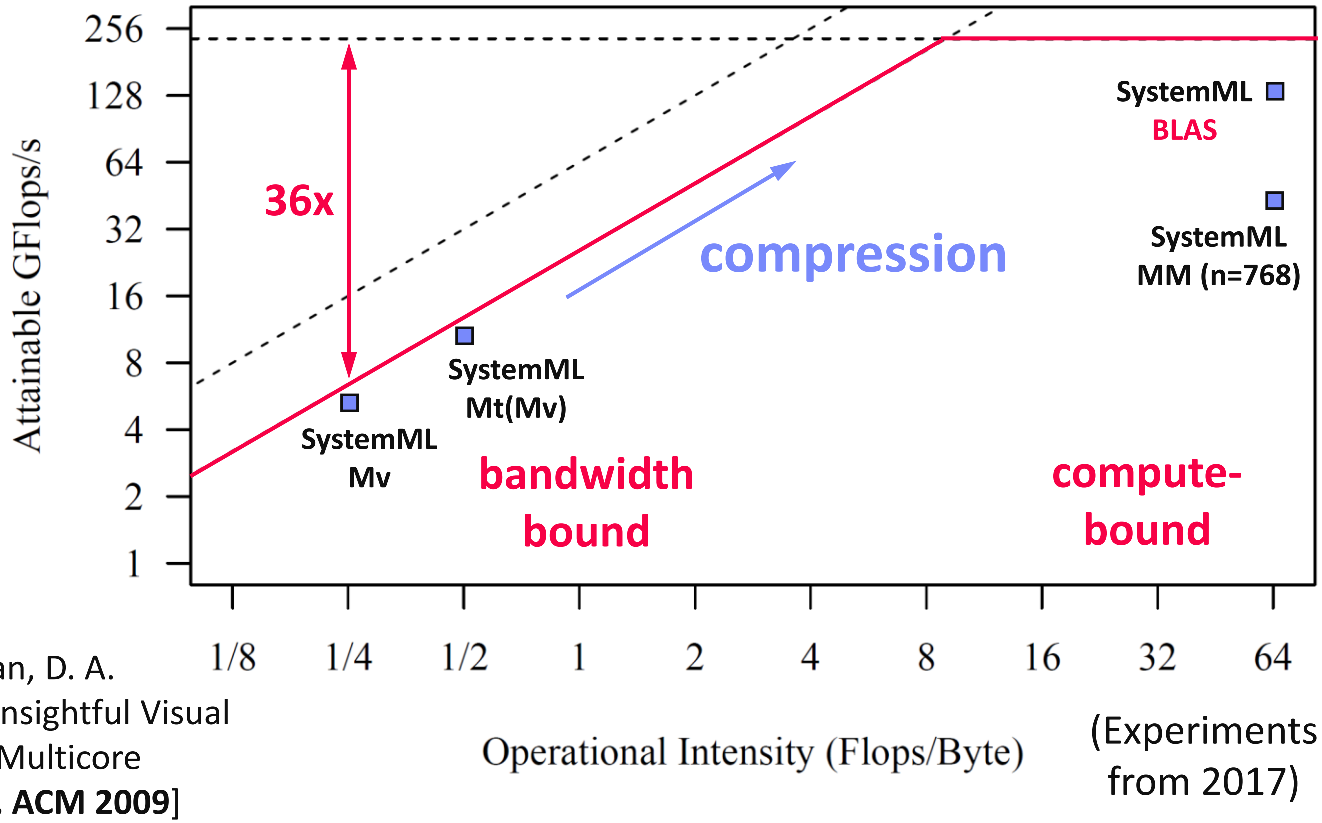
- **#2 Bandwidth**
  - **Memory Wall:** increasing gap CPU vs Memory latency/bandwidth
  - Reduce bandwidth requirements



[Stefan Manegold, Peter A. Boncz, Martin L. Kersten: Optimizing database architecture for the new bottleneck: memory access. **VLDB J. 9(3) 2000**]

Excursus: Roofline Analysis

- **Setup:** 2x6 E5-2440 @2.4GHz–2.9GHz, DDR3 RAM @1.3GHz (ECC)
  - Max mem bandwidth (local): 2 sock x 3 chan x 8B x 1.3G trans/s → **2 x 32GB/s**
  - Max mem bandwidth (QPI, full duplex) → **2 x 12.8GB/s**
  - Max floating point ops: 12 cores x 2*4dFP-units x 2.4GHz → **2 x 115.2GFlops/s**
- **Roofline Analysis**
  - Off-chip memory traffic
  - Peak compute



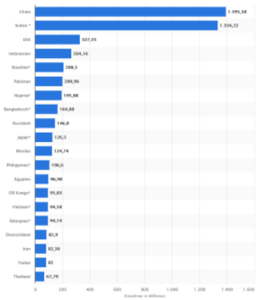
(Experiments from 2017)

[S. Williams, A. Waterman, D. A. Patterson: Roofline: An Insightful Visual Performance Model for Multicore Architectures. **Commun. ACM 2009**]

Motivation Data Characteristics

- **Skew**
  - Highly skewed **value distributions** (frequencies of distinct values)
  - Small number of distinct items

China **1.4**
India **1.3**
USA **0.33**
Germany **0.08**
Austria **0.009**



- **Correlation**
  - Correlation between tuple attributes
  - Co-occurrences of attribute values

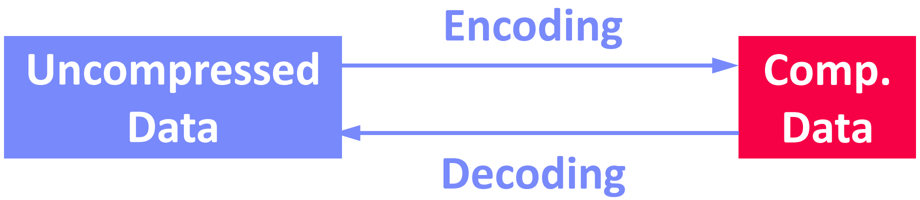
`OrderDate < ReceiptDate`
(usually 2-3 days)

- **Lack of Tuple Order**
  - Relations are multi-sets of tuples (no ordering requirements)
  - Flexibility for internal reorganization

[Vijayshankar Raman, Garret Swart: How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. **VLDB 2006**]

Compression Overview

- **Compression Codec**
  - Encoder
  - Decoder



Encoding

Uncompressed Data

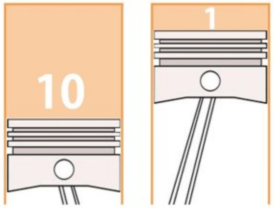
Comp. Data

Decoding

- **Lossless vs Lossy**
  - **Lossless:** guaranteed recovery of uncompressed data
  - **Lossy:** moderate degradation / approximation

    → Images, video, audio; ML training/scoring

- **Compression Ratio**
  - CR = Size-Uncompressed / Size-compressed
  - Ineffective compression: CR < 1

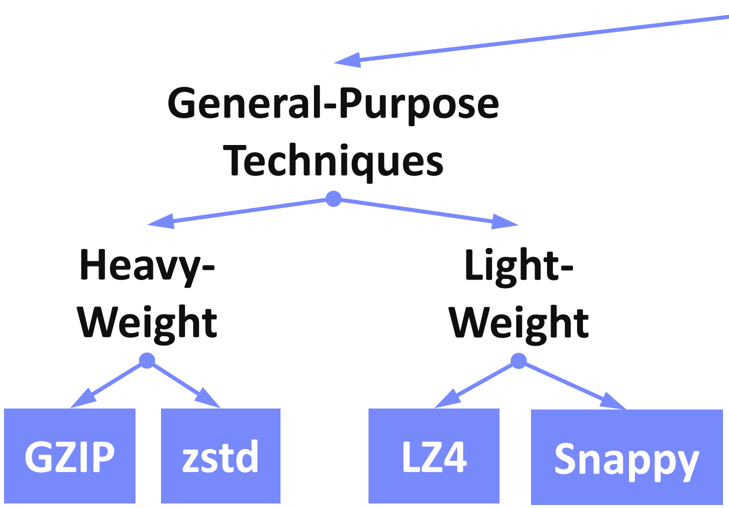


10 : 1

- **Metrics**
  - Compression ratio vs encode/decode time vs encode/decode space
  - Block-wise vs random access, operation performance, etc

Classification of Compression Techniques

- **Lossless Compression Schemes**



General-Purpose Techniques

DB-Centric Techniques

Heavy-Weight

Light-Weight

GZIP

zstd

LZ4

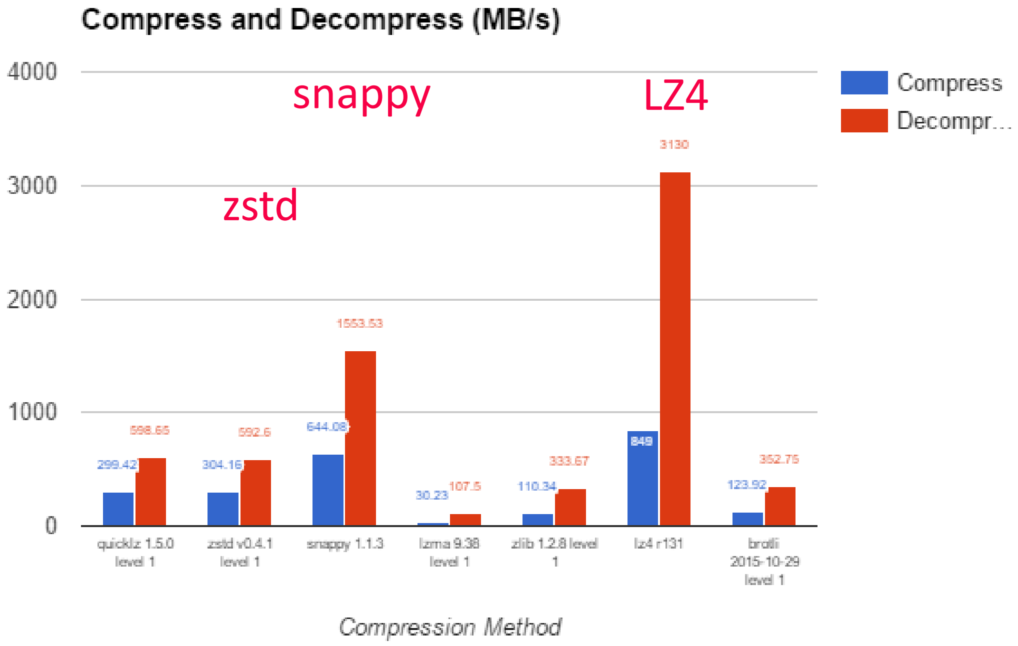
Snappy

Huffman + Lempel-Ziv

Excursus: General-purpose Compression

- **Compression/ Decompression**
  - CR zstd: 5.24
  - CR snappy: 3.65
  - CR LZ4: 3.89

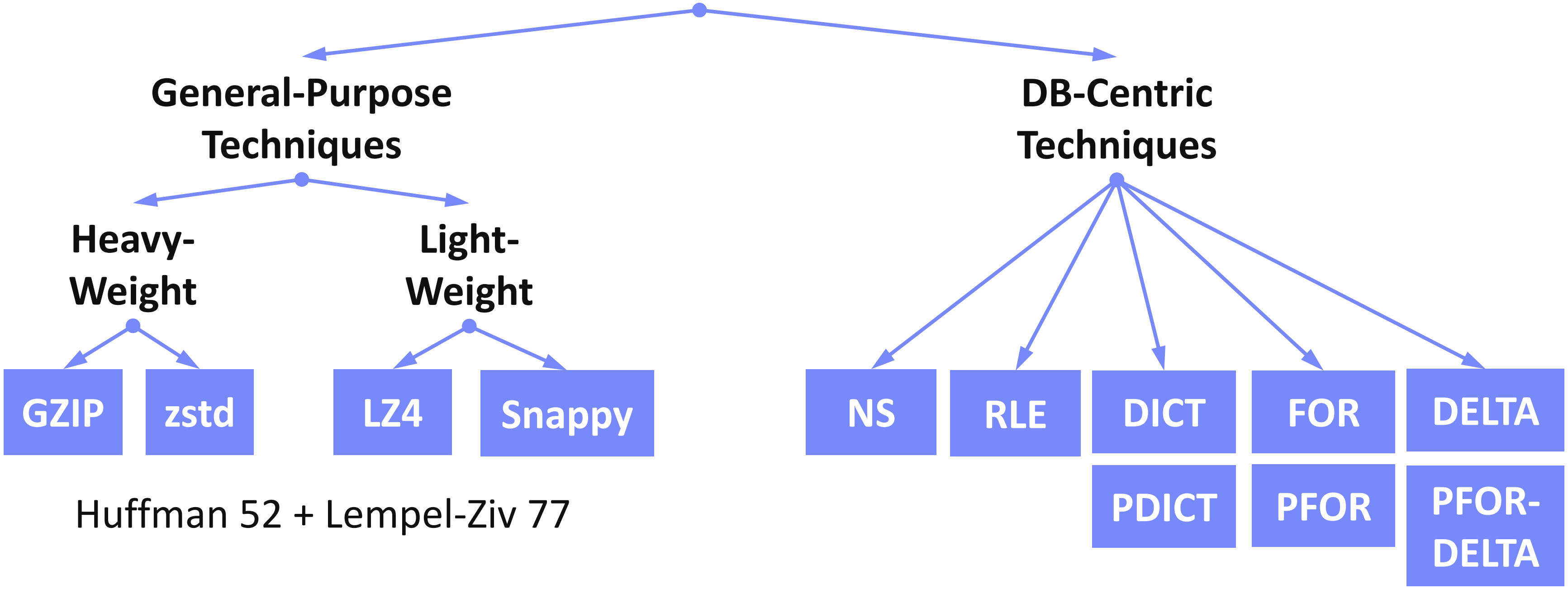
[https://web.archive.org/web/20200229161007/https://www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/]



- **Example** Apache Spark **RDD Compression**
  - `org.apache.spark.io.LZ4CompressionCodec` (**default in 2.x, 3.x**)
  - `org.apache.spark.io.SnappyCompressionCodec` (**default in 1.x**)
  - `org.apache.spark.io.LZFCompressionCodec` (**default in 0.x**)
  - `org.apache.spark.io.ZStdCompressionCodec`

Classification of Compression Techniques, cont.

- **Lossless Compression Schemes**



General-Purpose Techniques

Heavy-Weight

Light-Weight

DB-Centric Techniques

Huffman 52 + Lempel-Ziv 77

(all heavy-weight from a DB perspective)

Compression Techniques

Null Suppression (NS)

[Benjamin Schlegel, Rainer Gemulla, Wolfgang Lehner: Fast integer compression using SIMD instructions. **DaMoN 2010**]

- **Overview**
  - Compress **integers** by omitting **leading zeros** via variable-length codes
  - Universal compression scheme w/o need for upper bound

42

`00000000 | 00000000 | 00000000 | 00101010`

- **Byte-Aligned** (Example)
  - Store mask of two bits to indicate leading zero bytes
  - 2 bits + [1,4] bytes → max CR (INT32) = 3.2

42 `11 | 00101010`

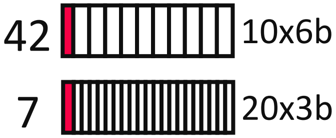
7 `11 | 00000111`

- **Bit-Aligned** (Example: Elias Gamma Encoding)
  - Store $N = \lfloor \log_2 x \rfloor$ zero bits followed by effective bits
  - 2 * [1,32] -1 bits → max CR (INT32) = 32

42 `00000 | 101010`

7 `00 | 111`

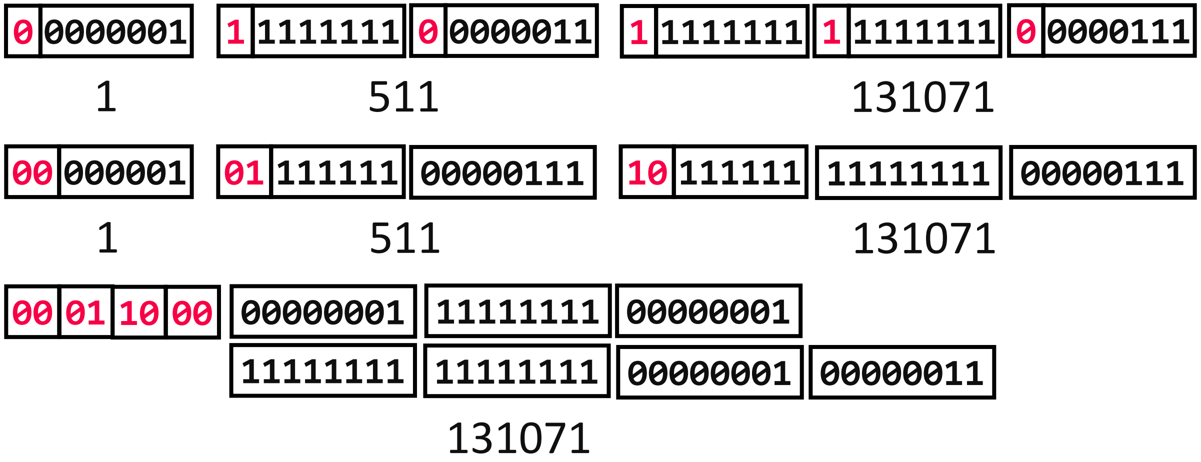
- **Word-Aligned** (Example: Simple-8b)
  - Pack a variable number of integers (max 2^{60}-1) into 64bit
  - 60 data bits, 4 selector bits (16 classes: 60x1b, 30x2b, …, 1x60b)



Null Suppression (NS), cont.

[Jeff Dean: Challenges in Building Large-Scale Information Retrieval Systems, Keynote **WSDM 2009**]

- **Varint (Variable-Length Integers)**
  [also Byte-Aligned]



  - **Base 128 Varint** (continuation bits)
    `0|0000001` → 1
    `1|1111111` `0|0000011` → 511
    `1|1111111` `1|1111111` `0|0000111` → 131071
  - **Prefix Varint** (2 bit #bytes)
    `00|000001` → 1
    `01|111111` `00000111` → 511
    `10|111111` `11111111` `00000111` → 131071
  - **Group Varint**
    `00|01|10|00` `00000001` `11111111` `00000001`
    `11111111` `11111111` `00000001` `00000011`
    131071
  - **Examples:**
    - Google Protobuf messages, SQLite custom varint

- **Zig-Zag Encoding**
  - Map signed integers to unsigned integers to have small varint byte length

Run-Length Encoding (RLE)

- **Overview**
  - Compress sequences of equal values via **runs** of `(value[,start],run-length)`
  - Redundant 'start' allows parallelization / unordered storage
  - Applicable to **arbitrary data types** (defined `equals()`)

- **Example**
  - Uncompressed

C	C	C	C	C	A	A	A	A	F	F	F	F	F	B	B	B

  - **Compressed**

C	**5**		A	**4**		F	**5**		B	**3**

    - Different physical encodings for values and lengths:
    - E.g., split runs w/ length ≥ 2^{16} to fit into fixed 2 byte

Dictionary Encoding (DICT)

- **Overview**
  - Build dictionary of distinct items and encode values as dictionary positions
  - Applicable to **arbitrary data types** → integer codes

- **Example**
  - Uncompressed

A	C	B	B	A	C	D	A	A	D	C	B	A	B	B	C	D

  - **Compressed**

Dict	
0	A
1	C
2	B
3	D

0	1	2	2	0	1	3	0	0	3	1	2	0	2	2	1	3

- Explicit or implicit (position) codes
- Fixed bit width: $\log_2 |Dict|$
- Different ordering of dictionary (alphanumeric, frequency)

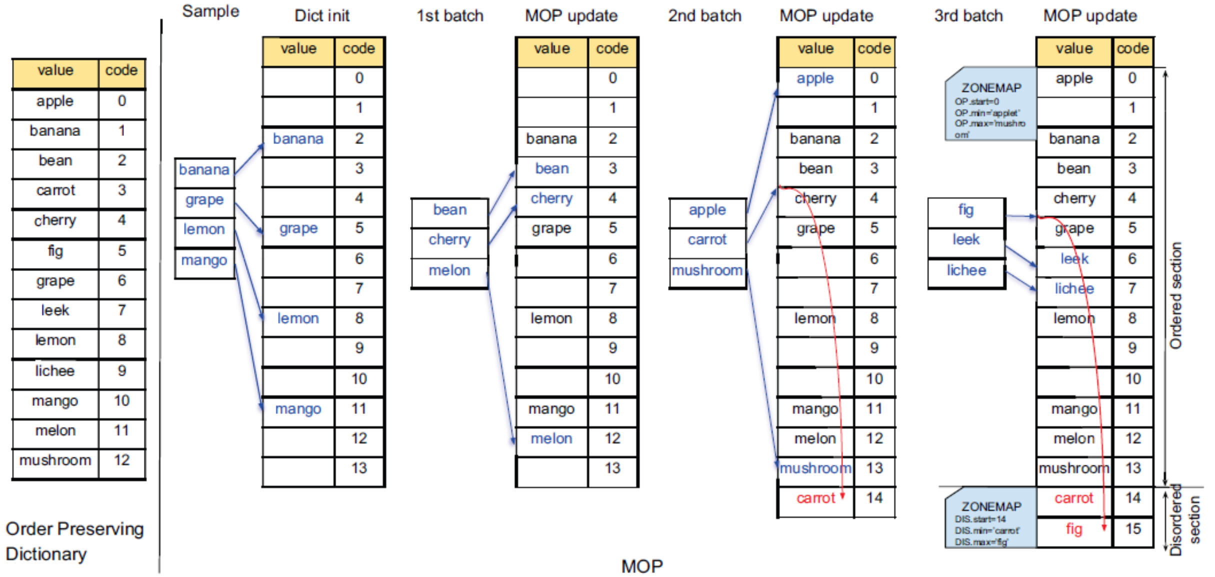
Dictionary Encoding (DICT), cont.

- **Order-preserving Dictionaries**
  - Create **sorted dictionary** where `order(codes) = order(values)`
  - Support for updates via **sparse code assignment** (e.g., 10, 20, 30)
  - CS-Array-Trie / CS-Prefix-Tree as encode/decode index w/ shared leafs

[Carsten Binnig, Stefan Hildenbrand, Franz Färber: Dictionary-based order-preserving string compression for main memory column stores. **SIGMOD 2009**]

- **Mostly Order-preserving Dictionaries**
  - Ordered and disordered dictionary sections

[Chunwei Liu et al: Mostly Order Preserving Dictionaries. **ICDE 2019**]



Order Preserving Dictionary

value	code
apple	0
banana	1
bean	2
carrot	3
cherry	4
fig	5
grape	6
leek	7
lemon	8
lichee	9
mango	10
melon	11
mushroom	12

Sample: banana, grape, lemon, mango

Dict init

value	code
	0
	1
banana	2
	3
	4
grape	5
	6
	7
lemon	8
	9
	10
mango	11
	12
	13

1st batch: bean, cherry, melon

MOP update

value	code
	0
	1
banana	2
bean	3
cherry	4
grape	5
	6
	7
lemon	8
	9
	10
mango	11
melon	12
	13

2nd batch: apple, carrot, mushroom

MOP update

value	code
apple	0
	1
banana	2
bean	3
cherry	4
grape	5
	6
	7
lemon	8
	9
	10
mango	11
melon	12
mushroom	13
carrot	14

3rd batch: fig, leek, lichee

MOP update

value	code
apple	0
	1
banana	2
bean	3
cherry	4
grape	5
leek	6
lichee	7
lemon	8
	9
	10
mango	11
melon	12
mushroom	13
carrot	14
fig	15

Ordered Section

Disordered Section

Frame of Reference Encoding (FOR)

- **Overview**
  - Compress values by storing **delta (difference) to reference value**
  - Mostly integer types → smaller integer domain

- **Example**
  - Uncompressed

701	698	702	700	699	698	700	701	701	700	703	702

  - **Compressed**

700

1	-2	2	0	-1	-2	0	1	1	0	3	2

Cannot handle trends very well

Delta Encoding (DELTA)

- **Overview**
  - Compress values by storing **delta (difference) to previous value**
  - Mostly integer types (good when sorted) → smaller integer domain
  - Dedicated techniques for differences of file contents (diff/git)

- **Example**
  - Uncompressed

5	5	6	6	7	7	7	9	9	12	13	14	15	16	17	17	18

  - **Compressed**

5	0	1	0	1	0	0	2	0	3	1	1	1	1	1	0	1

    - Delta
    - Double Delta (differences of differences)

Can create RLE
opportunities
for linear trend

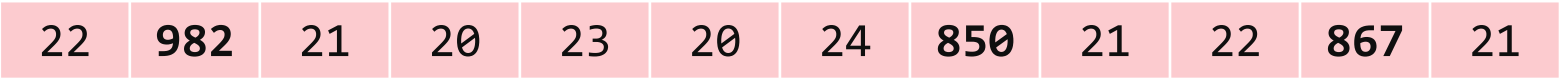
Patched Compression Methods (PFOR)

[Marcin Zukowski, Sándor Héman, Niels Nes, Peter A. Boncz: Super-Scalar RAM-CPU Cache Compression. **ICDE 2006**]

- **Patched Frame of Reference (PFOR)**
  - Store positive offsets to reference value
  - Exceptions in uncompressed form (accessible via entry points and offsets to next exception)
  - Branchless two-pass decoding

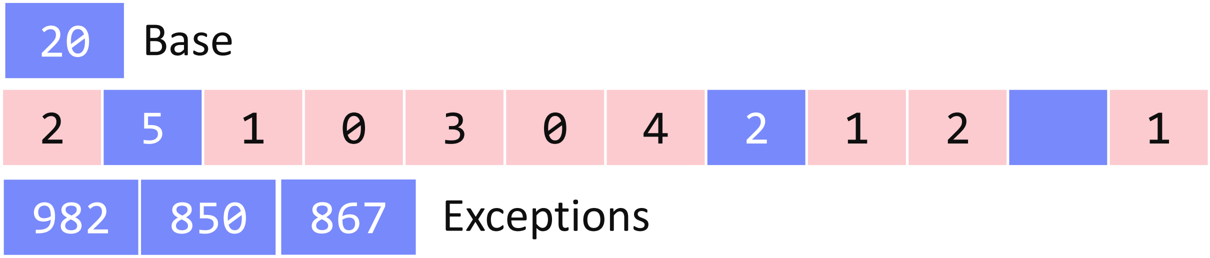
- **Example**
  - Uncompressed

Outliers would destroy fixed-width codes



22	**982**	21	20	23	20	24	**850**	21	22	**867**	21

  - **Compressed**



20	Base

2	5	1	0	3	0	4	2	1	2		1

982	850	867	Exceptions

Patched Compression Methods (Others)

- **PFOR-DELTA**
  - Apply cascade of DELTA – PFOR (PFOR on differences)
  - Handling of exceptions to handle large differences of subsequent values

[Marcin Zukowski, Sándor Héman, Niels Nes, Peter A. Boncz: Super-Scalar RAM-CPU Cache Compression. **ICDE 2006**]

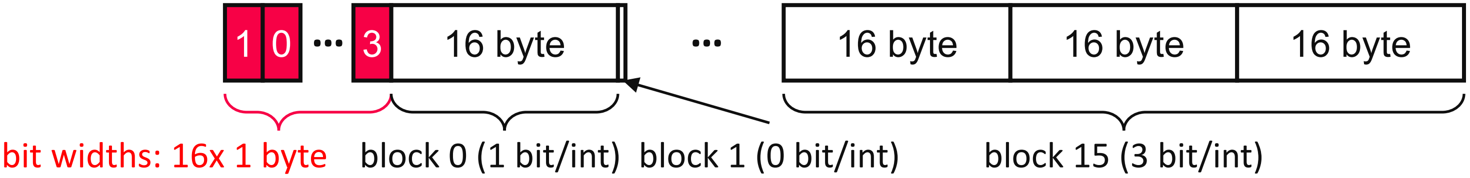
- **Patched Dictionary Compression (PDICT)**
  - Dictionary encoding, where only frequent values are encoded
  - Exceptions for infrequent values, previous/new dictionary per block
  - Reduces dictionary size

Removes long tail of infrequent distinct items from dictionary

Excursus: SIMD Implementation

- **How to reduce overhead of (de)compression?**
  - Vectorization of (de)compression by SIMD instructions
  - (De)compress several data elements at once
  - Main driver of research on database-centric lightweight compression for years
- **SIMD-BP128**
  - Targets compression of 32-bit ints using Intel's SSE with 128-bit vectors (16 byte)
  - Sub-divide integer sequence into blocks of 128 integers
  - Determine maximum bit width in a block by bitwise OR
  - Pack all integers in a block using that number of bits
  - Dedicated vectorized (un)packing routine for each bit width
  - Store bit width in [0, 32] as one byte, combine 16 of those in memory

[Daniel Lemire, Leonid Boytsov: Decoding billions of integers per second through vectorization. **Softw. Pract. Exp. 45(1) (2015)**]



1 0 … 3 | 16 byte | … | 16 byte | 16 byte | 16 byte

bit widths: 16x 1 byte

block 0 (1 bit/int)

block 1 (0 bit/int)

block 15 (3 bit/int)

Comparative Evaluation

- **Experimental Survey**
  - Different data characteristics
  - Compression methods:

    DELTA, RLE, FOR, RLE, DICT,
    SIMD-BP128, SIMD-FastPFOR,
    4-Wise NS, 4-Gamma, Masked VByte,
    Simple-8b, SIMD-GroupSimple
  - Cascades of compression methods

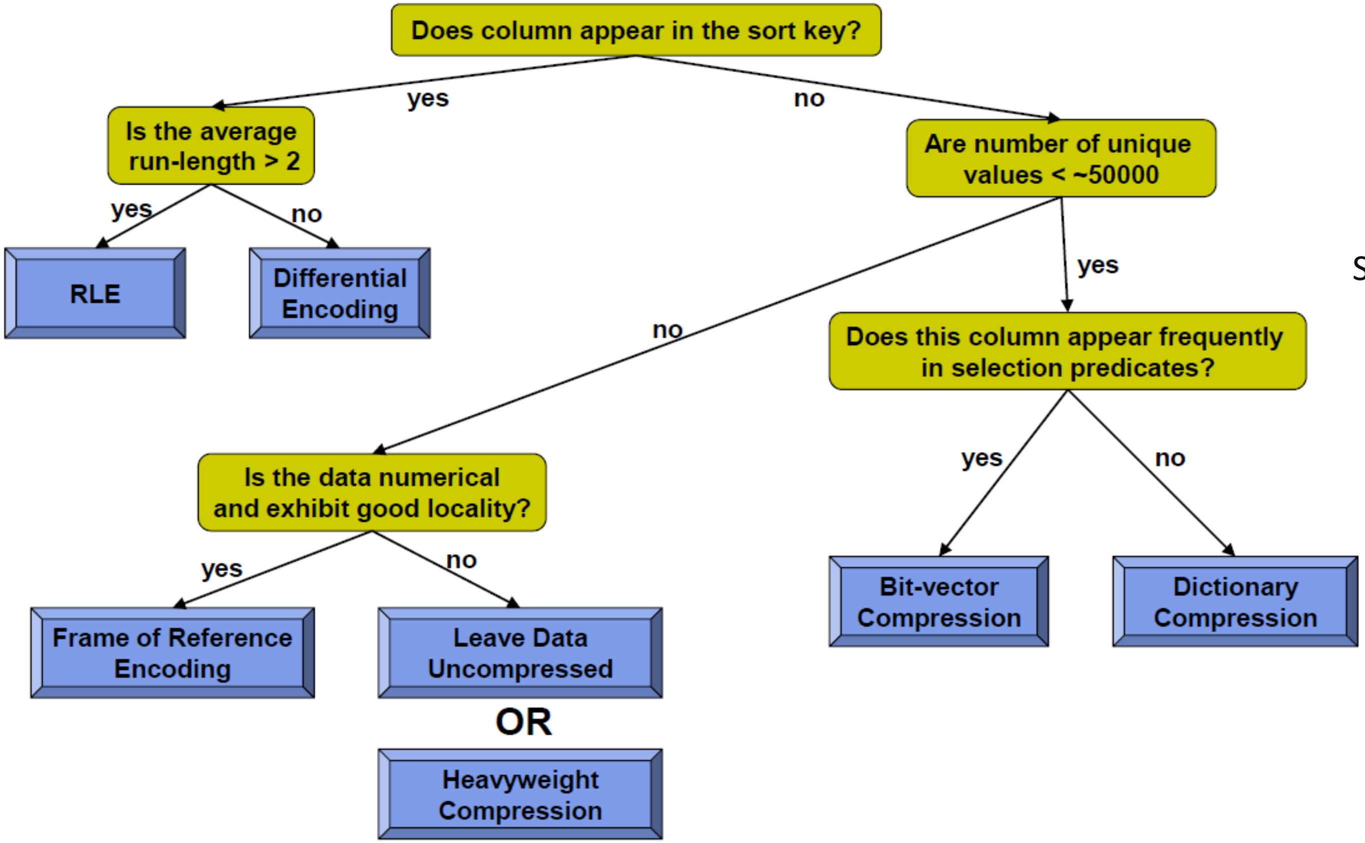
[Patrick Damme, Dirk Habich, Juliana Hildebrandt, Wolfgang Lehner: Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). **EDBT 2017**]

"[…] there is **no single-best lightweight integer compression algorithm**. The compression rates and performances of all algorithms differ significantly, depending on the data characteristics and the employed SIMD extension."

- **Towards a Cost-based Selection**
  - Logical and physical level
  - Cost estimation functions

[Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, Wolfgang Lehner: From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. **ACM Trans. Database Syst. 44(3) 2019**]

Selecting Compression Methods



Does column appear in the sort key?

yes

no

Is the average run-length > 2

yes

no

RLE

Differential Encoding

Are number of unique values < ~50000

no

yes

Does this column appear frequently in selection predicates?

yes

no

Bit-vector Compression

Dictionary Compression

Is the data numerical and exhibit good locality?

yes

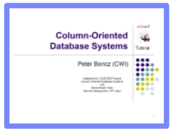
no

Frame of Reference Encoding

Leave Data Uncompressed

OR

Heavyweight Compression



Column-Oriented Database Systems

Peter Boncz (CWI)

[Peter Boncz: Column-Oriented Database Systems, adapted from **VLDB'09 tutorial**]

- **Inspired by**
  C-Store Compression Paper

[Daniel J. Abadi, Samuel Madden, Miguel Ferreira: Integrating compression and execution in column-oriented database systems. **SIGMOD 2006**]

Compressed Query Processing

Selection Predicates

- **Equivalence Predicates $\sigma_{attr='D'}(R)$**
  - DICT: code lookup

Dict	
0	A
1	C
2	B
3	D

D → 3

0	1	2	2	0	1	**3**	0	0	**3**	1	2	0	2	2	1	**3**

position vector

6	**9**	**16**

  - RLE: return RLE runs

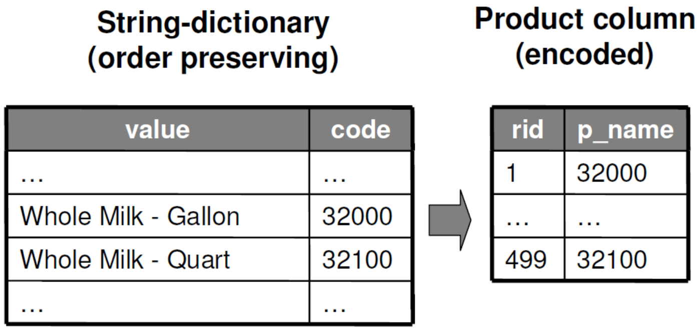
C	**5**	A	**4**	D	**5**	F	**3**

- **Range Predicates $\sigma_{3<a<7}$ (R)**
  - **#1** sort the dictionary by value (insert tradeoff)
  - **#2** expand small integer domains + dictionary lookup (e.g., $\sigma_{a=4 \vee a=5 \vee a=6}$ (R))
  - **#3** decompress otherwise

Selection Predicates, cont.

- **Order Preserving Dictionaries**
  - Direct support for range predicates on encoded data
  - Support for LIKE predicates (suffix)

[Carsten Binnig, Stefan Hildenbrand, Franz Färber: Dictionary-based order-preserving string compression for main memory column stores. **SIGMOD 2009**]



String-dictionary (order preserving)

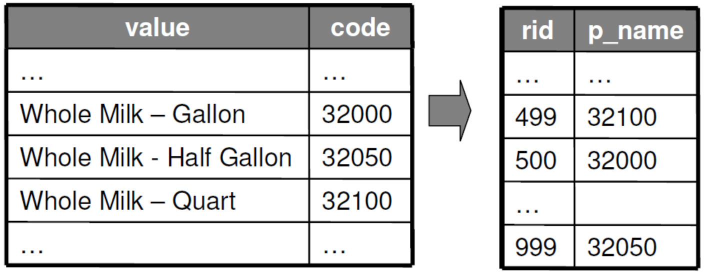
value	code
…	…
Whole Milk - Gallon	32000
Whole Milk - Quart	32100
…	…

Product column (encoded)

rid	p_name
1	32000
…	…
499	32100

Query (original):

```
Select SUM(o_total), p_name
  From Sales, Products
 Where p_name='Whole Milk*'
   Group by p_name
```



value	code
…	…
Whole Milk – Gallon	32000
Whole Milk - Half Gallon	32050
Whole Milk – Quart	32100
…	…

rid	p_name
…	…
499	32100
500	32000
…	
999	32050

Query (rewritten):

```
Select SUM(o_total), p_name
  From Sales, Products
  Where p_name ≥ 32000
   And p_name ≤ 32100
   Group by p_name
```

Grouping and Aggregations

- **Basic Hash Aggregates**
  - Grouping directly with compressed codes
  - DICT, FOR, RLE, etc

Dict	
0	A
1	C
2	B
3	D

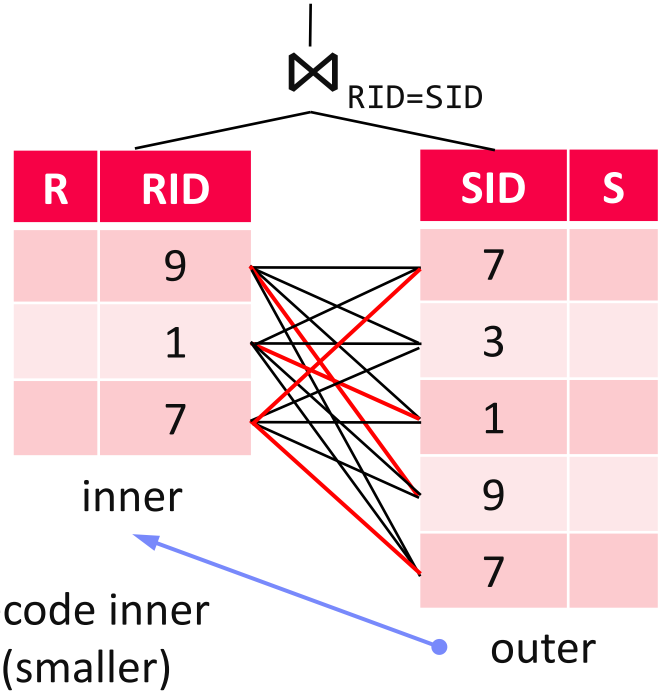
0	1	2	2	0	1	3	0	0	3	1	2	0	2	2	1	3

Hash Table	
0	Agg A
3	Agg D
1	Agg C
2	Agg B

- **Encoding-Specific Aggregation**
  - RLE sum → agg += run-length*run-value
  - RLE min → agg = min(agg, run-value)
  - FOR sum → for all codes: agg += code; agg += |codes| * base-value

Joins

- **Overview Compressed Joins**
  - (Equi-)Joins directly over compressed data
  - **Beware:** binary operation → encodings need to match (**global code**)
  - Recoding of one of the inputs if necessary (e.g., DB2 BLU recode inner)



- **Encoding-Specific Aggregation**
  - One input RLE: decompress other and output RLE encoded data
  - One input bitvector: decompress other and output RLE encoded data (obtained from bitvector)

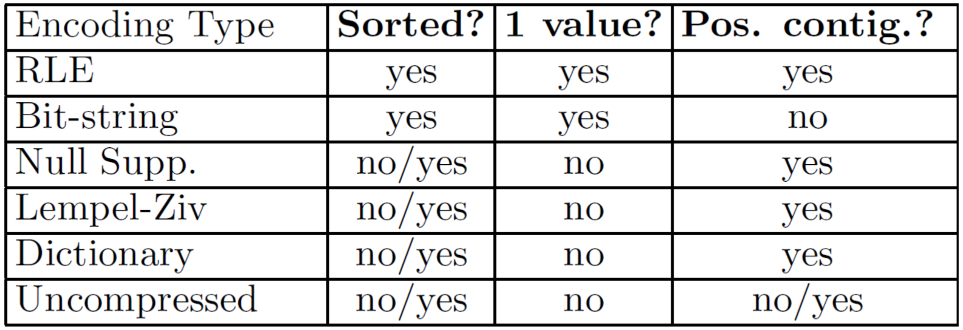
Abstractions for Simpler Code

- **Motivation**
  - Code complexity for combinations of encoding schemes
  - Affects all operators → maintenance operators/compression schemes

- **Compressed Block Properties**
  - `isOneValue()`: block contains just one value and many positions for that value
  - `isValueSorted()`: all values of the block are sorted
  - `isPosContig()`: block contains consecutive subset of column

[Daniel J. Abadi, Samuel Madden, Miguel Ferreira: Integrating compression and execution in column-oriented database systems. **SIGMOD 2006**]

- **Iterator Access:** `getNext()`, `asArray()`
- **Block Information:** `getSize()`, `getStartValue()`, `getEndPosition()`

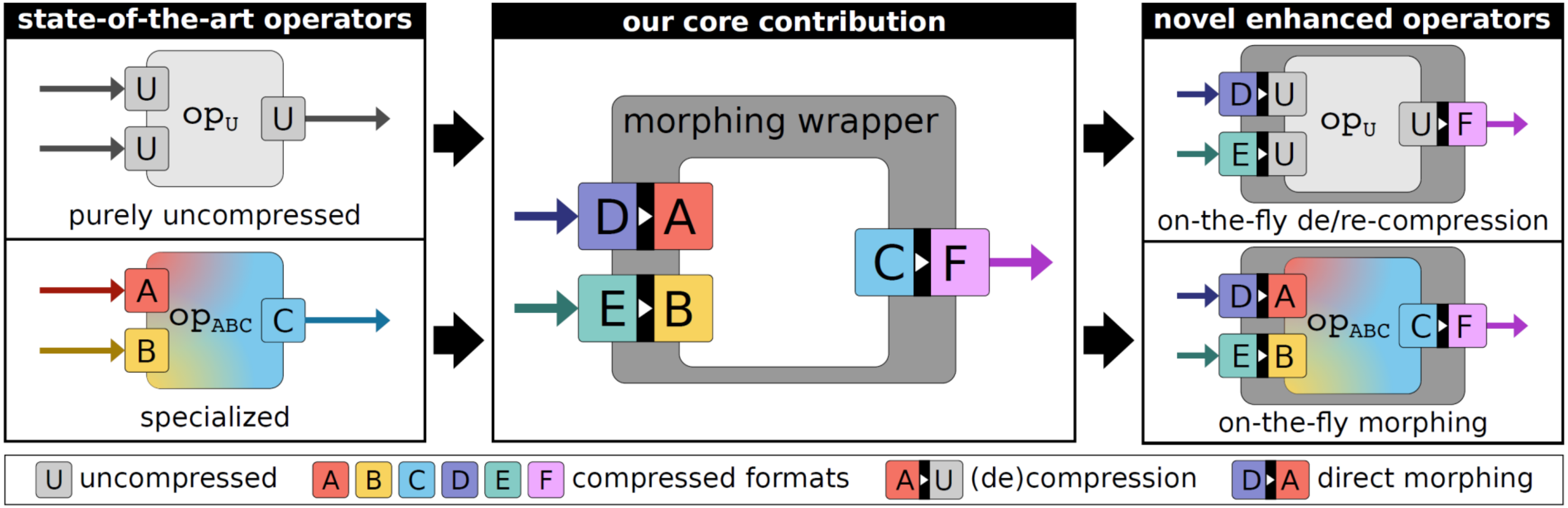


Encoding Type	**Sorted?**	**1 value?**	**Pos. contig.?**
RLE	yes	yes	yes
Bit-string	yes	yes	no
Null Supp.	no/yes	no	yes
Lempel-Ziv	no/yes	no	yes
Dictionary	no/yes	no	yes
Uncompressed	no/yes	no	no/yes

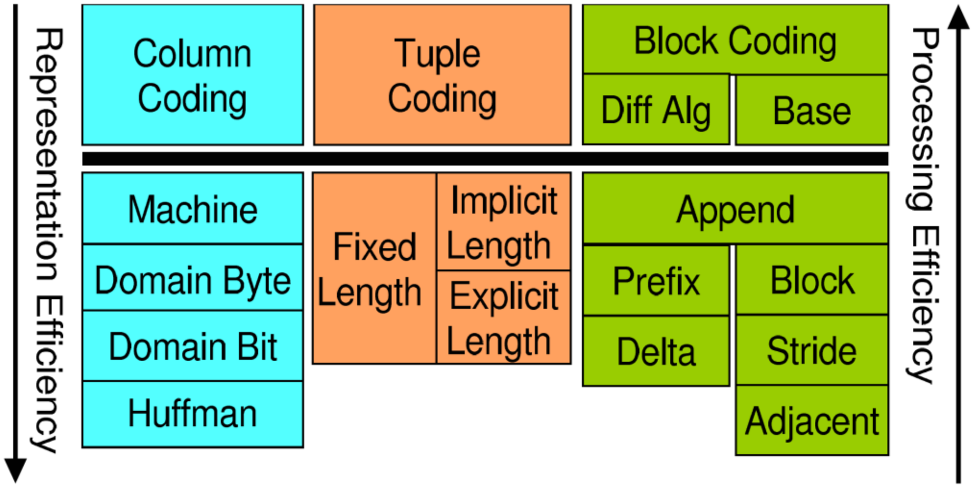
Abstractions for Simpler Code, cont.

- **Motivation**
  - Improve query performance by (re)compressing intermediates
  - Change from one compressed format to another

[Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner: MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. **PVLDB 13(11) 2020**]



Data Layout – Compression Granularity



Representation Efficiency

Column Coding

Tuple Coding

Block Coding

Diff Alg

Base

Machine

Domain Byte

Domain Bit

Huffman

Fixed Length

Implicit Length

Explicit Length

Append

Prefix

Block

Delta

Stride

Adjacent

Processing Efficiency

[Allison L. Holloway, Vijayshankar Raman, Garret Swart, David J. DeWitt: How to barter bits for chronons: compression and bandwidth trade offs for database scans. **SIGMOD 2007**]

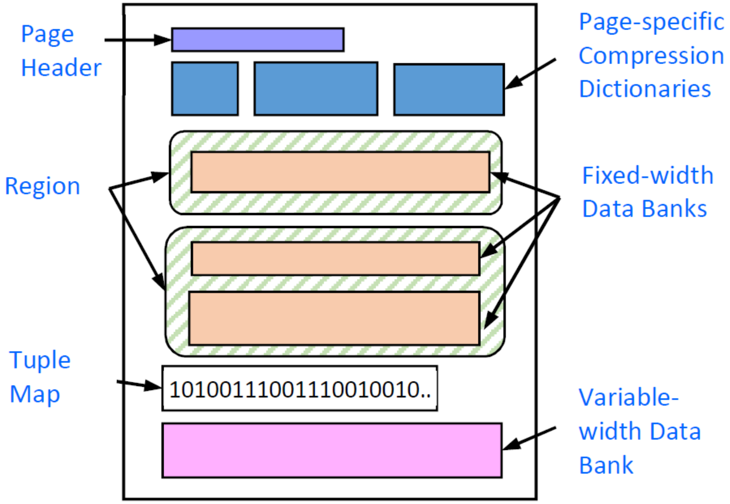
"All the results have shown that the Huffman coded and delta coded formats compress better but normally take more CPU time. […] When I/O and memory subsystem times are also included in the decision, the format to choose becomes less clear-cut. If a physical format optimizer or system administrator had this information and a fast scan generator, they could make a more informed choice as to the best way to store the data."

- **Column Coding**
  - Select encoding for individual attributes (column values) – tradeoffs
- **Tuple Coding**
  - Combine column codes into tuple codes (fixed, variable)
- **Block Coding**
  - Compress a sequence of tuples into a compressed block (concat, diff)

03 Buffer Pool Management

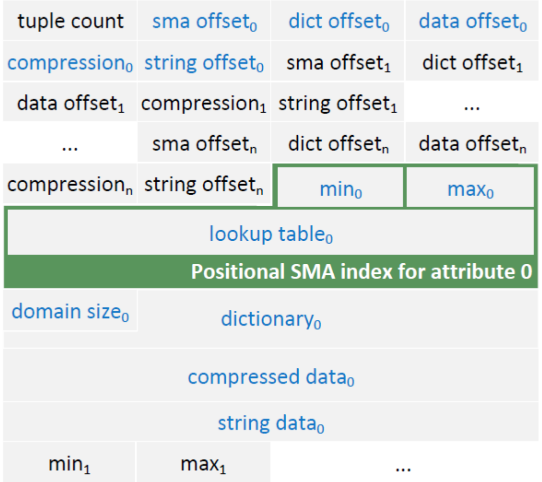
Data Layout – Example Block Layouts

- **DB2 BLU**



[Vijayshankar Raman et al: DB2 with BLU Acceleration: So Much More than Just a Column Store. **PVLDB 6(11) 2013**]

- **Data Blocks**



tuple count	sma offset$_0$	dict offset$_0$	data offset$_0$
compression$_0$	string offset$_0$	sma offset$_1$	dict offset$_1$
data offset$_1$	compression$_1$	string offset$_1$	…
…	sma offset$_n$	dict offset$_n$	data offset$_n$
compression$_n$	string offset$_n$	min$_0$	max$_0$
lookup table$_0$			
Positional SMA index for attribute 0			
domain size$_0$	dictionary$_0$		
compressed data$_0$			
string data$_0$			
min$_1$	max$_1$	…	

[Harald Lang: Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. **SIGMOD 2016**]

03 Buffer Pool Management
04 Index Structures and Partitioning
07 Query Compilation and Parallelization

DB-Compression Beyond Relational Databases

- **Information Retrieval**
  - (web) search engines
  - Inverted index, postings lists (sorted lists of document ids)
- **Time Series**
  - Internet of Things, sensor networks, server/application metrics, etc.
  - Sequences of data points (measurement + time)
- **Machine Learning**
  - Various application fields
  - Matrices/tensors of various characteristics
- **Graph Databases**
  - Social networks, road networks, proteins, etc.
  - Graphs represented as adjacency lists, matrices

Also benefit from the compression techniques discussed today

Summary and **Q&A**

- **Motivation and Terminology**
- **Compression Techniques**
- **Compressed Query Processing**

- **Next Lectures** (Part B)
  - **06 Query Processing** (operators, execution models)
  - **07 Query Compilation and Parallelization**
  - **08 Query Optimization** (rewrites, costs, join ordering)
  - **10 Adaptive Query Processing**