6. Data Compression

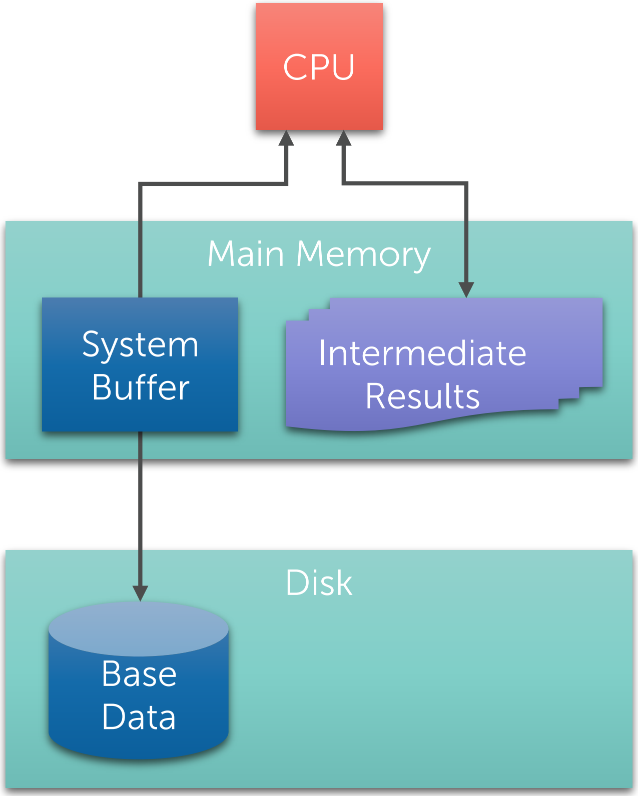
Architecture of Database Systems

Dresden Database Systems Group

Disk Centric DBMS

CHARACTERISTICS

- Base data resides on disk
- Bottleneck between disk and main memory
- Access to disk orders of magnitude slower than to main memory
- Focus on optimization of the access to the disk
- Internal data representation (e.g. intermediate results) less relevant for optimization



Main Memory Centric DBMS

The typical choice for analytical data processing

Characteristics

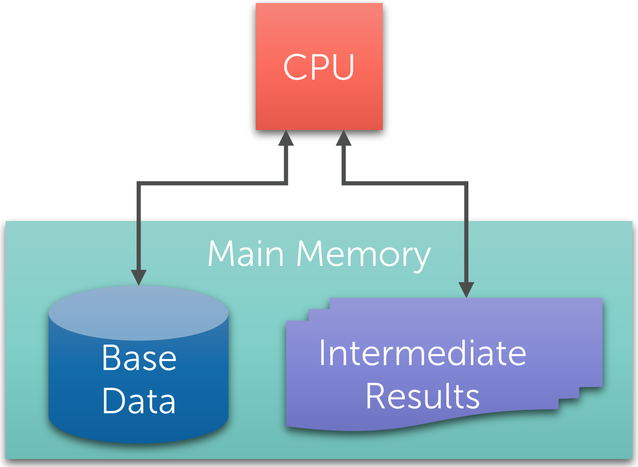
- All data resides in main memory
  (base data *and* intermediate results)
- Try to optimize the access to the memory hierarchy

Remarkable Fact

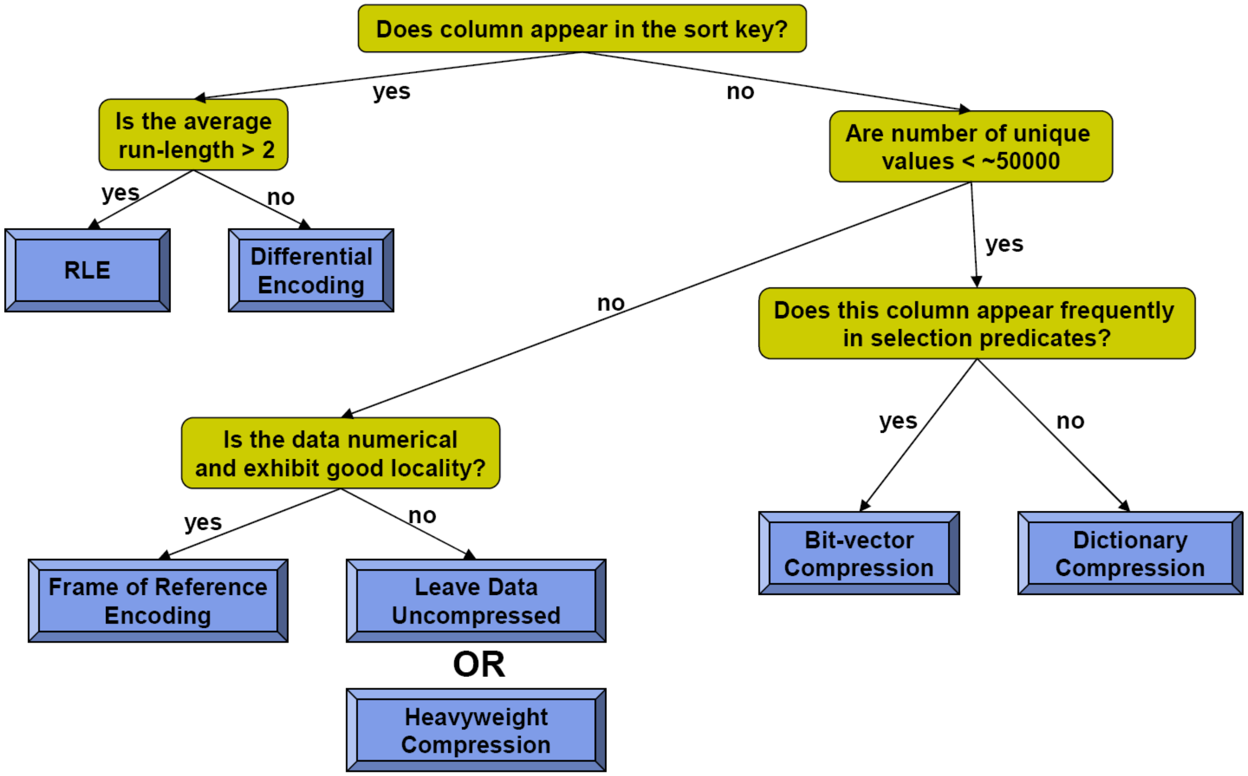
- **Accessing intermediate results is as expensive as accessing the base data**

Consequence

- Optimization potential of intermediate results is high …
  … and grows: the larger the base data the larger the
  intermediate results (in general)

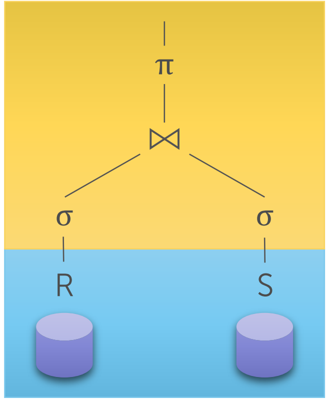


Advisor for Column Store



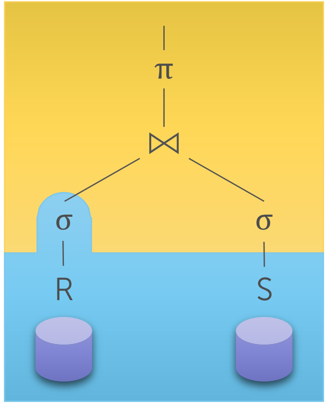
Integration: Decompression Strategies

Eager Decompression



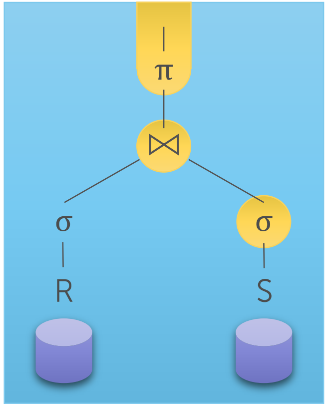
- Compressed pages on disk
- Decompress pages when loaded into memory
- No changes to query processing

Lazy Decompression



- Keep data compressed as long as possible
- Decompress only when required for certain operator
- After that, data remains uncompressed

Transient Decompression



- Decompression before operators if required
- Output in compressed format
- Full exploitation of compression

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

Raw Data Compression

Terminology

Dresden Database Systems Group

Data Compression

- Transforms data to minimize size of its representation
- In contrast: *Data Reliability* is often implemented by adding check and parity bits and increases redundancy and size of the data

Examples

- File compression – Gzip
- CD/DVD players
- digital camera image storage – JPEG
- …

Modeling and Coding

- Components of the compression process

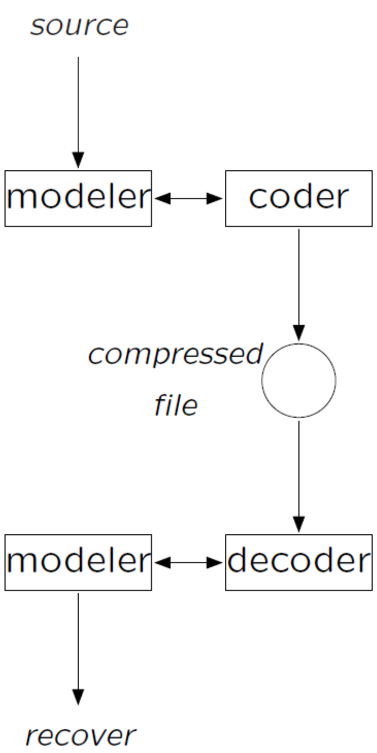
Modeling

- describe form of redundancy
- build abstract prototype of the source
- select source elements for focus

Coding

- encode model and description of how data differs from model
- construct new representation using model
- map source elements to produce output

Compression-Decompression Process



Terminology

- Encoding: compression, reduce representation
- Decoding: recover the original data

- Lossless: recover precisely the original data
- Lossy: original data nor recovered exactly

- 2-pass: pre-scan of source required
- 1-pass: no pre-scan required

Redundancy / Methods

Types of Source Element Redundancy

- Distribution → some elements occur more often than others, e.g., ; in C programs
- Repetition → elements often repeated, e.g. 1 or 0 in b/w image bit maps
- Patterns → correlation of symbols occurrence, e.g.. „th, qu“ in English
- Positional → some elements occur mostly in the same relative positions

Methods for Compression

- Pre-filtering → reduce complexity of data may remove relevant details
- Eliminate redundancy → remove any repeated information
- Use human perception models → remove irrelevant details in ways that maximize humans‘ ability to detect the information loss
- Post-filtering → attempt to further reduce/mask artifacts that were introduced by information loss

Types of Codes

Dresden Database Systems Group

A Code is a Mapping

- from source = stream over alphabet S
- to compressor output = stream of code words over alphabet C

Fixed Code

- Code word set is time invariant
- Selection is predetermined

Static Code

- Code word set is time invariant
- Selection dictated by model

Adaptive Code

- Dynamic code word set (varies over time)
- Selection/modification dictated by model

Classified by Input-Output Rates

- for time-invariant codes

Fixed-to-Fixed rate Code: $S \rightarrow C$

- ASCII code

Fixed-To-Variable Rate Code: $S \rightarrow C^+$

- Morse, Huffmann Codes

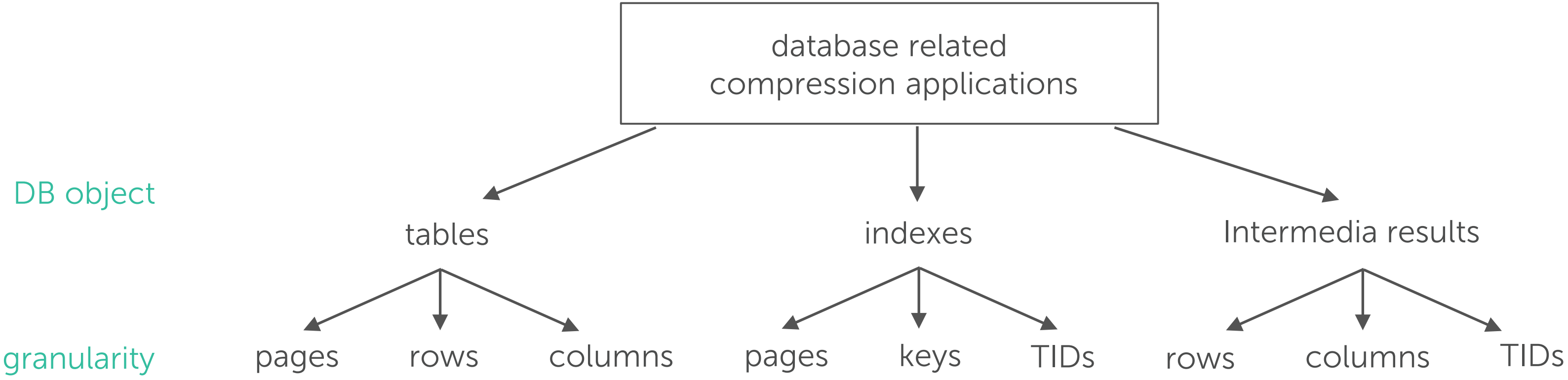
Variable-to-Fixed Rate Codes: $S^+ \rightarrow C$

- Lempel-Ziv Methods

Variable-to-Variable Rate Code: $S^+ \rightarrow C^+$

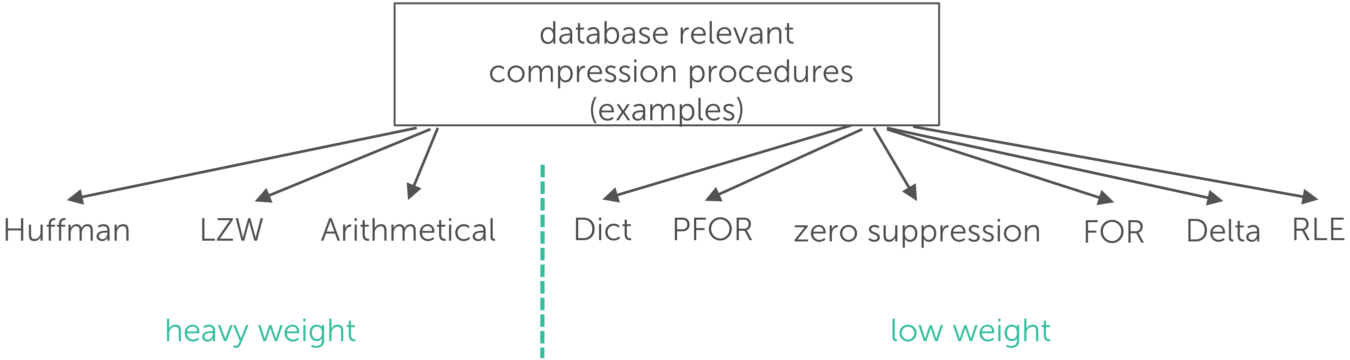
- Run length Encoding, Arithmetic Coding

Classes of Application Areas



Classes of Algorithms

Dresden Database Systems Group



Data Compression as an optimization approach

- Many different procedures and algorithms
- Use of light-weighted procedures for main-memory based approaches

Run Length Encoding - RLE

Description

- Also called run length encoding (Lauflängenkodierung)
- Long sequences of same values are replaced by the one-time storage of the value combined with the frequency of the repeats
- Particularly for column oriented data organization; further supported by sorting to achieve longer sequences

Ort
Berlin
Berlin
Berlin
Ilmenau
Ilmenau
Ilmenau
Magdeburg
Rostock
Rostock

Ort	
3	Berlin
3	Ilmenau
1	Magdeburg
2	Rostock

Delta Coding - Delta

Description

- Storage of value differences to previous value instead of the value itself
- Particularly for consecutive values with a small difference
- Sorting helpful for optimize compression ratio

KNr	PLZ
...	39106
...	39114
...	39126
...	98684
...	98693

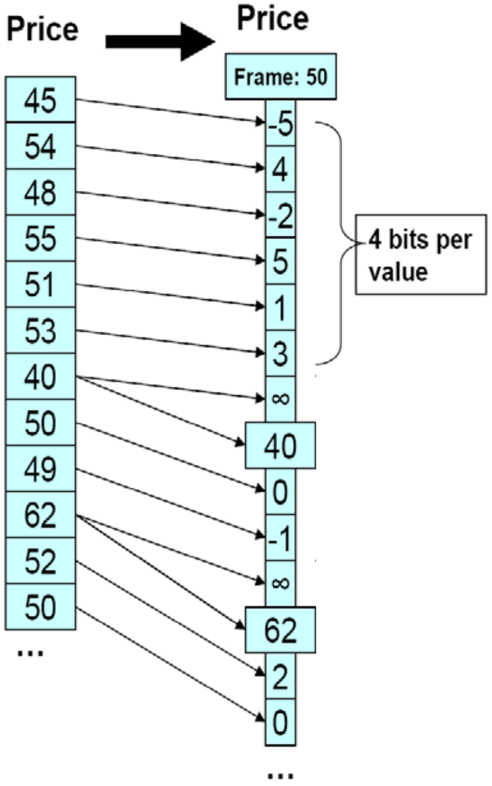
PLZ
39106
8
12
59558
7

Frame Of Reference Encoding - FOR

Dresden Database Systems Group

Description

- Encodes values as b bit offset from chosen frame of reference
- Special escape code (e.g. all bits set to 1) indicates a difference larger than can be stored in b bits
- After escape code, original (uncompressed) value is written



Bit-Vector Encoding

Description

- Small number of different values: one bit string per column value
  - 1, if tuple at the position has that value
  - 0, otherwise

- Length of bit string accords to number of tuples
- Use i.a. for bitmap-indexes

KNr	Kundenstatus
...	Premium
...	Silber
...	Standard
...	Standard
...	Standard
...	Premium
...	Silber
...	Standard

Premium:	1000.0100
Silber:	0100.0010
Standard:	0011.1001

Dictionary Encoding - DICT

Description

- Use of a dictionary for all (string-)values and input of a code for the actual column value
- Particulary for frequent and long values

KNr	Bundesland
...	Thüringen
...	Thüringen
...	Sachsen
...	Sachsen-Anh.
...	Hessen
...	Bayern
...	Hessen
...	Sachsen-Anh.

Bundesland
0100
0100
0010
0011
0001
0000
0001
0011

Dictionary

Bayern	0000
Hessen	0001
Sachsen	0010
Sachsen-Anh.	0011
Thüringen	0100

Null Suppression

Dresden Database Systems Group

Main idea

- Partition the integer into a sequence of *leading zero bytes* and a sequence of *effective bytes*
- Store only number of leading zero bytes and effective bytes
- Leading zero bytes stored using 2 bits

Example

- Uncompressed binary representation of the integer 100

$$\underbrace{00000000\ \ 00000000\ \ 00000000}_{\text{3 leading zero bytes}}\ \ \underbrace{01100100}_{\text{1 effective byte}}$$

- Compressed binary representation of the integer 100

$$\underbrace{11}_{\text{number of leading zero bytes}}\ \ \underbrace{01100100}_{\text{1 effective byte}}$$

Null Suppression (SIMD-Implementation)

Dresden Database Systems Group

Main idea

- Partition the integer into a sequence of *leading zero bytes* and a sequence of *effective bytes*
- Store only number of leading zero bytes and effective bytes
- Leading zero bytes stored using 2 bits

Example

- Uncompressed binary representation of the integer 100

$$\underbrace{00000000\ \ 00000000\ \ 00000000}_{\text{3 leading zero bytes}}\ \ \underbrace{01100100}_{\text{1 effective byte}}$$

- Compressed binary representation of the integer 100

$$\underbrace{11}_{\text{number of leading zero bytes}}\ \ \underbrace{01100100}_{\text{1 effective byte}}$$

Elias Gamma Encoding

Dresden Database Systems Group

Main idea

- Partition the integer into a sequence of *leading zero bits* and a sequence of *effective bits*
- Unary encode the number of effective bits before the effective bits (use l zero bits to denote a length of l+1 effective bits)

Example

- Uncompressed binary representation of the integer 100

$$\underbrace{00000000\ \ 00000000\ \ 00000000\ \ 0}_{\text{25 leading zero bits}}\underbrace{1100100}_{\text{7 effective bits}}$$

- Compressed binary representation of the integer 100

$$\underbrace{00000\ 0}_{\text{6 leading zero bits}}\underbrace{1100100}_{\text{7 effective bits}}$$

k-wise Null Suppression - Encoding

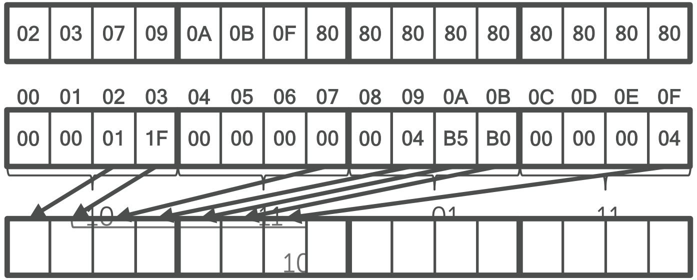
Dresden Database Systems Group

Main idea

- Use horizontal memory layout
- Utilize the SIMD-byte-permute instruction

Steps of compression by example

1) Count leading zero bytes
2) Determine compression permutation mask id
3) Load the compression permutation vector
4) Permute the bytes
5) Write back compressed byte sequence and permutation mask id



Permutation Masks

Permutation Mask Table

- Based on 32-bit integers and a fixed size of SIMD-registers

- 32-bit integers → 4 possible configuration of leading zero bytes (00, 01,10,11)
- For each of this configuration, a permutation mask can be computed

- 256-bit SIMD-register → 4 integer values → that means 4*4 configuration of leading zero bytes
- Permutation masks can be computed (restricted areas)

Leading zero configuration	Permutation mask
…	
10 11 01 11	02 03 07 09 0A 0B 0F 80 80 80 …
…	

k-wise Null Suppression - Decoding

DECOMPRESSION

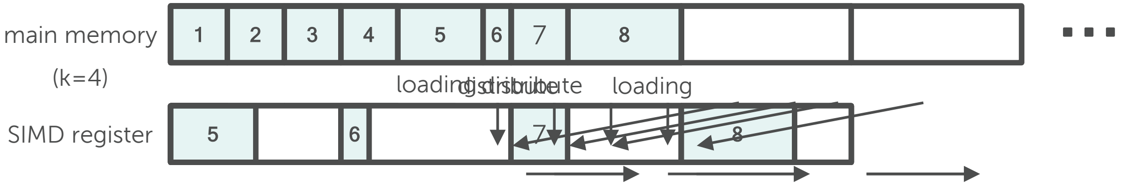
	Value 1	*Value 2*	*Value 3*	*Value 4*
Inverse permutation mask B7	80 80 00 01	80 80 80 02	80 03 04 05	80 80 80 06

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Compressed values	01	1F	00	04	B5	B0	04	00	00	00	00	00	00	00	00	00
Uncompressed values (32-bit)	00	00	01	1F	00	00	00	00	00	04	B5	B0	00	00	00	04

Memory Layouts

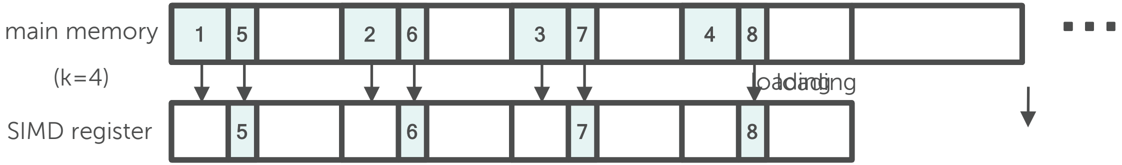
Code words in the horizontal layout

- Stored like for sequential processing
- Loaded using a single SIMD load instruction
- Need to be distributed into *k* elements of an SIMD register



Code words in the vertical layout

- Stored in different memory words
- Already distributed after loading
- Each *k* consecutive must be equal-sized



RleSimd: Format

Idea of Run Length Encoding

- View subsequent occurrences of the same value as a run
- Each run representable by its value and length
  → just two integers

RleSimd

- Our vectorized implementation of RLE
- Uses SIMD instructions to parallelize comparisons
- Currently based on 128-bit vector operations (SSE2)

uncompressed

00	98	76	54
00	98	76	54
00	98	76	54
00	98	76	54
12	34	56	78
00	00	AB	CD
00	00	AB	CD
00	00	AB	CD

Rle

00	98	76	54	run value
00	00	00	04	run length
12	34	56	78	run value
00	00	00	01	run length
00	00	AB	CD	run value
00	00	00	03	run length

RleSimd: Compression

Dresden Database Systems Group

..	AB	CD	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54

uncompressed memory

RleSimd: Compression

..	AB	CD	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54	uncompressed memory

1) _mm_set1_epi32()

00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54	128-bit vector register

RleSimd: Compression

Dresden Database Systems Group

..	AB	CD	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54	uncompressed memory

1) _mm_set1_epi32()	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54	128-bit vector register
2) _mm_loadu_si128()	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54	

RleSimd: Compression

Dresden Database Systems Group

..	AB	CD	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54	uncompressed memory

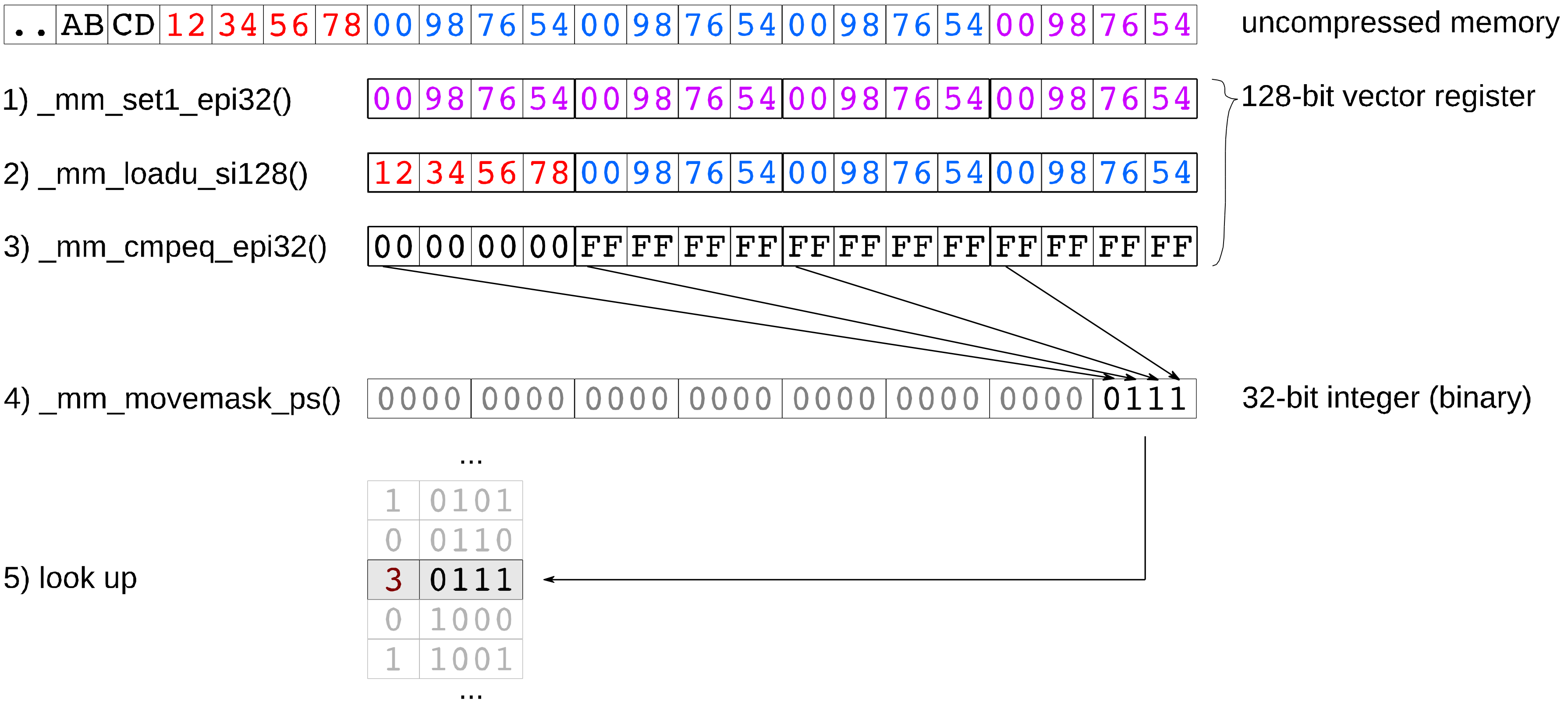
1) _mm_set1_epi32()	00	98	76	54	00	98	76	54	00	98	76	54	00	98	76	54
2) _mm_loadu_si128()	12	34	56	78	00	98	76	54	00	98	76	54	00	98	76	54
3) _mm_cmpeq_epi32()	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

$\left.\right\}$ 128-bit vector register

RleSimd: Compression

	.. AB CD 12 34 56 78 00 98 76 54 00 98 76 54 00 98 76 54 00 98 76 54	uncompressed memory
1) _mm_set1_epi32()	00 98 76 54 00 98 76 54 00 98 76 54 00 98 76 54	128-bit vector register
2) _mm_loadu_si128()	12 34 56 78 00 98 76 54 00 98 76 54 00 98 76 54	
3) _mm_cmpeq_epi32()	00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF	
4) _mm_movemask_ps()	0000 0000 0000 0000 0000 0000 0000 0111	32-bit integer (binary)

RleSimd: Compression



RleSimd vs. RleSeq: Compression Times

Dresden Database Systems Group

Setup

- Implementation in C++
- g++ using –O3 optimization flag
- Intel Core i3-2350M
- 4 GB main memory

Results

- 100M uncompressed integers
- Run length > 7
  → RleSimd faster than RleSeq
- Speed up converges on 2.0 for long runs
- Peak speed up of 3.0
- Fast compression is what we need for balanced query processing