5. Column-based Storage Model

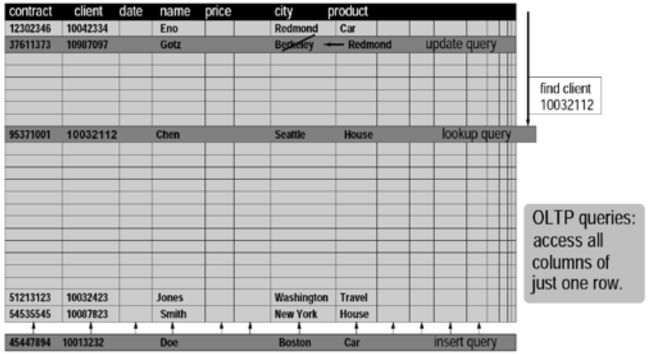
Architecture of Database Systems

Application Characteristics

Dresden Database Systems Group

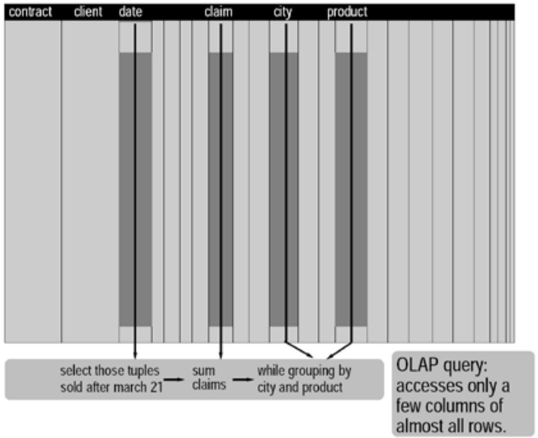
OLTP (On-line Transaction Processing)

- Mix between read-only and update queries
- Minor analysis tasks
- Used for data preservation and lookup
- Read typically only a few records at a time
- High performance by storing contiguous records in disk pages



OLAP (On-line Analytical Processing)

- Query-intensive DBMS applications
- Infrequent batch-oriented updates
- Complex analysis on large data volumes
- Read typically only a few attributes of large amounts of historical data in order to partition them and compute aggregates
- High performance by storing contiguous values of a single attribute



Introduction

Dresden Database Systems Group

Recap – Row-Based Record Management

- Classic N-ary storage model (NSM), also „row-store“
- NSM = Normalized Storage Model

Advantages

- Common database can be read with one page view
- single attributes values can simply be changed

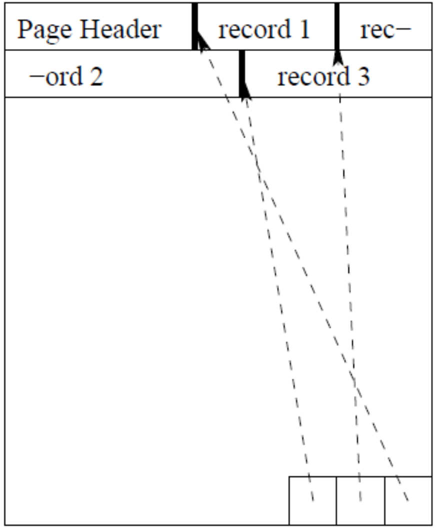
Disadvantages:

- even if only a few attributes value are needed, all attributes values have to be read

→ unnecessary IO-cost

Alternatives: Column-Oriented Storage Models

- Decomposition of a n-ary relation in a set of projections (for example binary relation)
- Identification (and reconstruction) by a key column or position



NSM page organization

Example

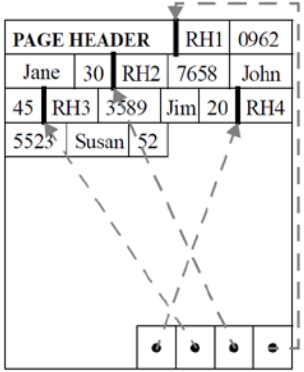
Row-Based Record Management versus Column-Based Record Management

ProdNr	Bezeichnung	Preis
1	Jamaica Blue	8,55
2	Arabica Black	9,95
3	New York Espresso	10,95
4	Guatemala Grande	11,95
5	Breakfast Blend	9,90

SAdr	ProdNr
0x00	1
0x01	2
0x02	3
0x03	4
0x04	5

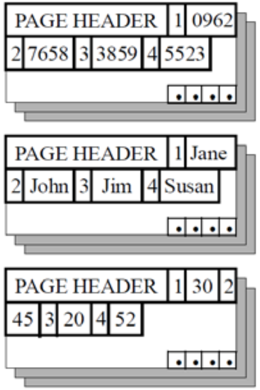
SAdr	Bezeichnung
0x00	Jamaica Blue
0x01	Arabica Black
0x02	New York Espresso
0x03	Guatemala Grande
0x04	Breakfast Blend

SAdr	Preis
0x00	8,55
0x01	9,95
0x02	10,95
0x03	11,95
0x04	9,90



Dataset as a unit

Set of vertical projections



sub-relation R1

sub-relation R2

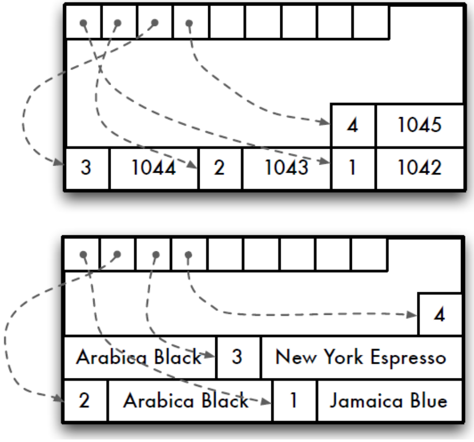
sub-relation R3

Decomposition Storage Model - DSM

G.P. Copeland, S.F. Khoshafian: A Decmposition Storage Model, In: SIGMOD 1985, pages 268-279

Description

- All values of a column (attribute) are saved consecutively
- Addressing by position / logical ID (surrogate)
- Page view (data set consisting of 2 attributes)

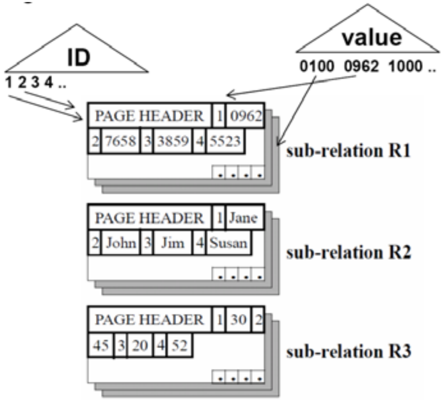


Subrelation ProdNr

Subrelation Bezeichnung

1985: DSM (Decomposition Storage Model)

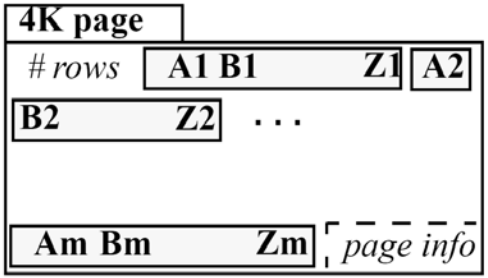
- Proposed as an alternative to NSM (Normalized Storage Model)
- Decomposition storage mode, decomposes relations <u>vertically</u>
- 2 indexes: clustered on ID, non-clustered on value
- Speeds up queries projecting few columns
- Disadvantages: storage overhead for storing tuple IDs, expensive tuple reconstruction costs



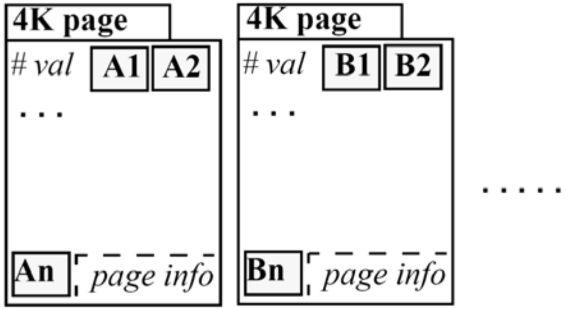
Comparison

Dresden Database Systems Group

ROW STORAGE



COLUMN STORAGE



+ easy to add/modify a record

- might read unnecessary data

+ only need to read in relevant data, more efficient scan operators
+ Compression easily possible (for example run length encoding)

- tuple writes require multiple accesses
- Reading all columns of a single row requires expensive row-reconstruction (1:1 join)

PAX – the Best of Both Worlds?

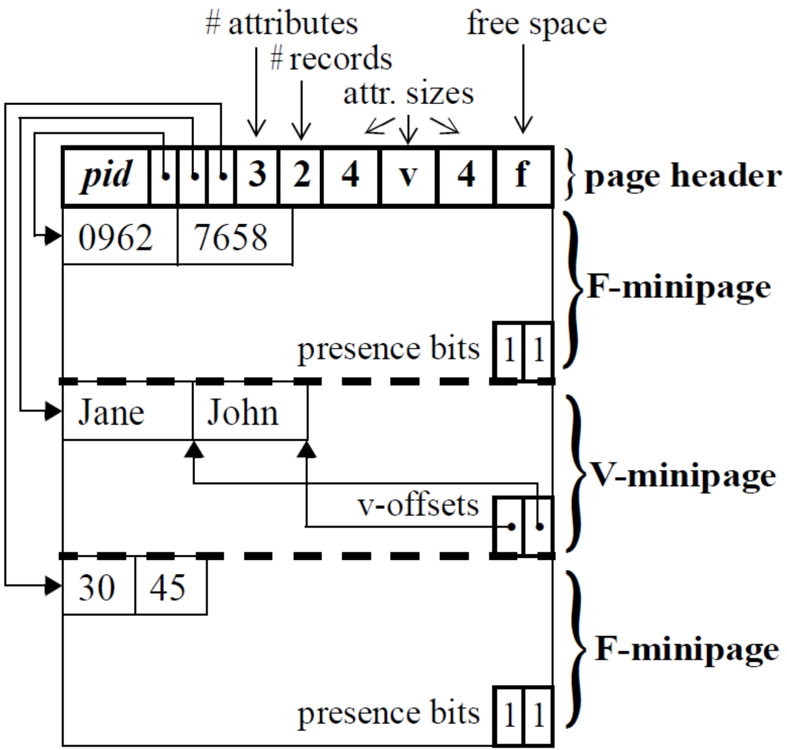
Partition Attributes Across - PAX

GOALS

- Maximizes inter-record spatial locality
- Incurs a minimal record reconstruction cost

APPROACH

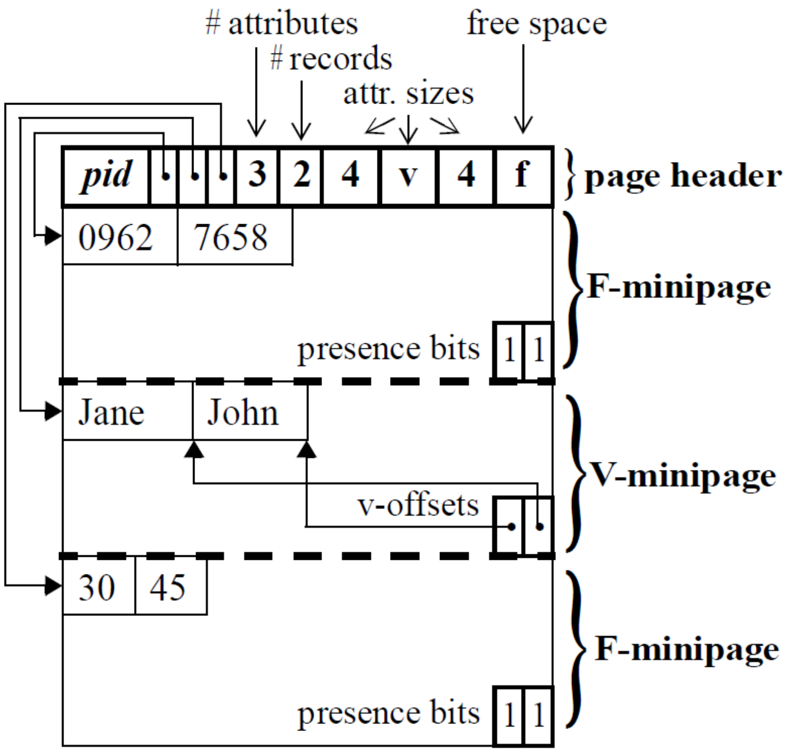
- compromise between NSM and DSM
- keep attributes values of each record on the same page
- using a cache-friendly algorithm for placing attributes values inside the page
  - vertically partitions the records within each page
  - storting together the values of each attribute in minipages



PAX-Design

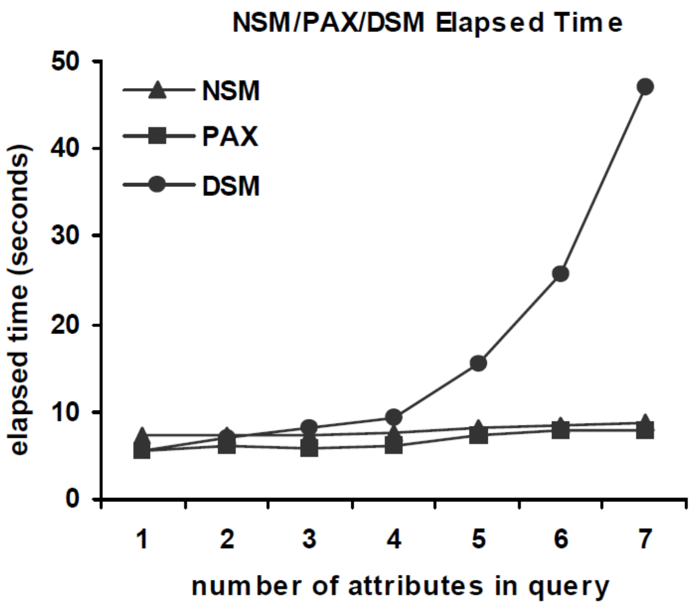
Storage Design

- each page is partitioned in *n* minipages (*n* attributes in a relation)
- Page Header
  - pointers to the beginning of each minipage
  - free space information
  - number of records
  - attributes sizes (fixed length or variable)
- Minipages
  - F-minpage → fixed-length attribute values, precence bits indicate the availability of attributes values for the records (if null, the attribute value is not present)
  - V-minipage → variable-length attributes values, slotted with pointers to each value, null values are denoted by null pointers

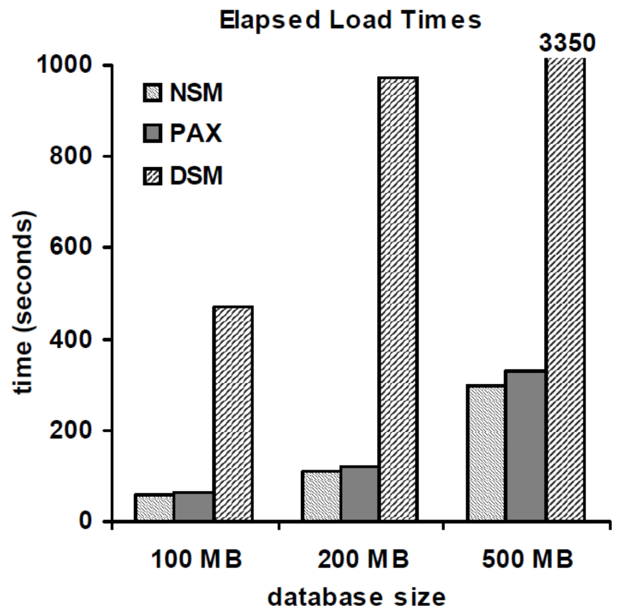


Evaluation

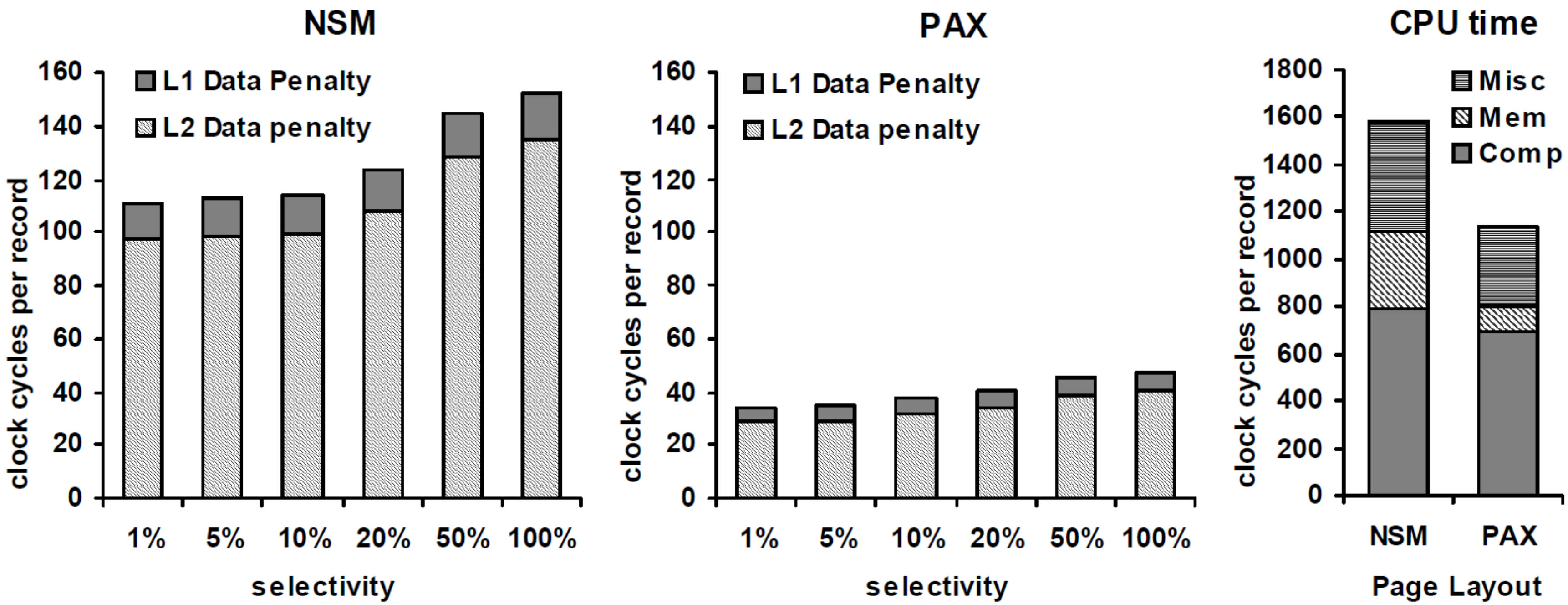
Query Performance (Read)



Bulk-Loading



Cache Behavior



History & Development

From DSM to Column-Stores

Dresden Database Systems Group

1985: Decomposition Storage Model

Late 90s – 2000s: Focus on main-memory performance

- MonetDB
- PAX: Partition Attributes Across
  - Retains NSM I/O pattern
  - Optimizes cache-to-RAM communication

2005: The (re)birth of column-stores

- New hardware and application realities
  - Faster CPUs, larger memories, disk bandwidth
  - Multi-terabyte Data Warehouses
- New approach: combine several techniques
  - Read-optimized, fast multi-column access, disk/CPU efficiency, light-weight compression
- Used in read oriented environments - OLAP

Some column store systems

- MonetDB, C-Store, Sybase IQ, SAP HANA, Infobright, Exasol, X100/VectorWise

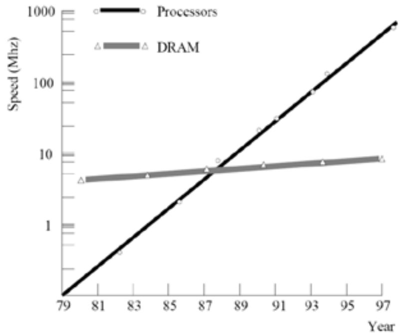
Main Reasons for Columnar Data representation

- Significantly higher cache hit rate
  - Data is more compact represented
- Use of HW-prefetcher
  - Sequential memory scans trigger pre-load of memory pages
- Use of SIMD-instructions
  - Multiple logical operations within one single CPU operation
- Opportunity for aggressive compression
  - Dictionary encoding as prerequisite

Hardware Development - Memory Wall

Hardware improvements not equally distributed

- Advances in CPU speed have outpaced advances in RAM latency



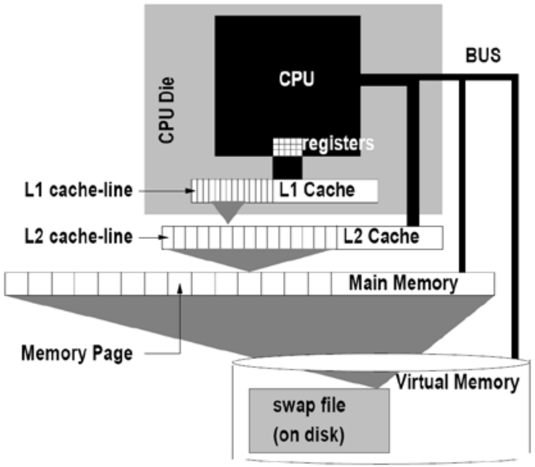
- Main-memory access has become a performance bottleneck for many computer applications
  - Bandwidth
  - Latency
  - Address translation (TLB)

→ Memory Wall

Cache memories can reduce the memory latency

when the requested data is found in the cache.

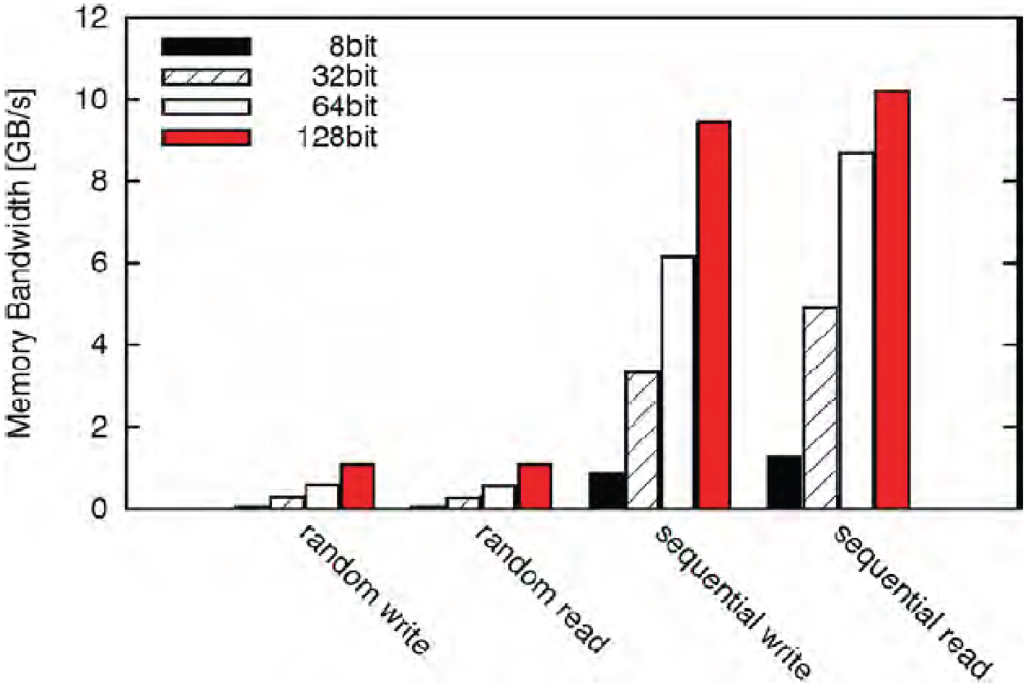
- Vertically fragmented data structures optimize memory cache usage



Memory Performance Comparison

Impact of Caches

- Better cache efficiency
  - Multiple data snippets within the same cache line
- HW prefetcher
  - Sequential access trigger the pre-fetcher to pre-load subsequent memory pages



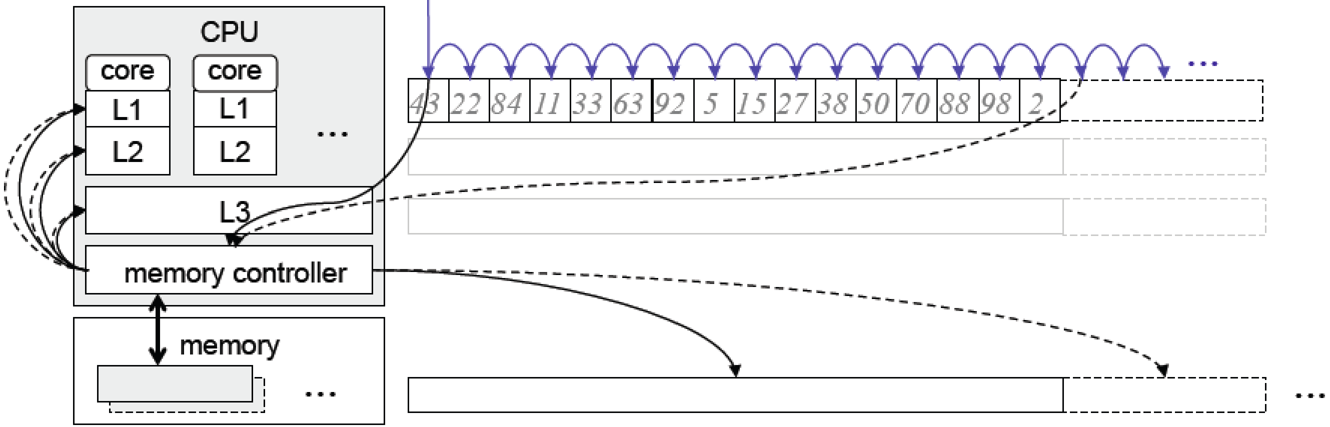
Results for a quad-core i7 2.66GHz, DDR3 1666. 32GB data accessed total.

The Role of Caches

Caches – the sunny side

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequentia

```
for (i=0; i<N; i++) x += A[i];
```



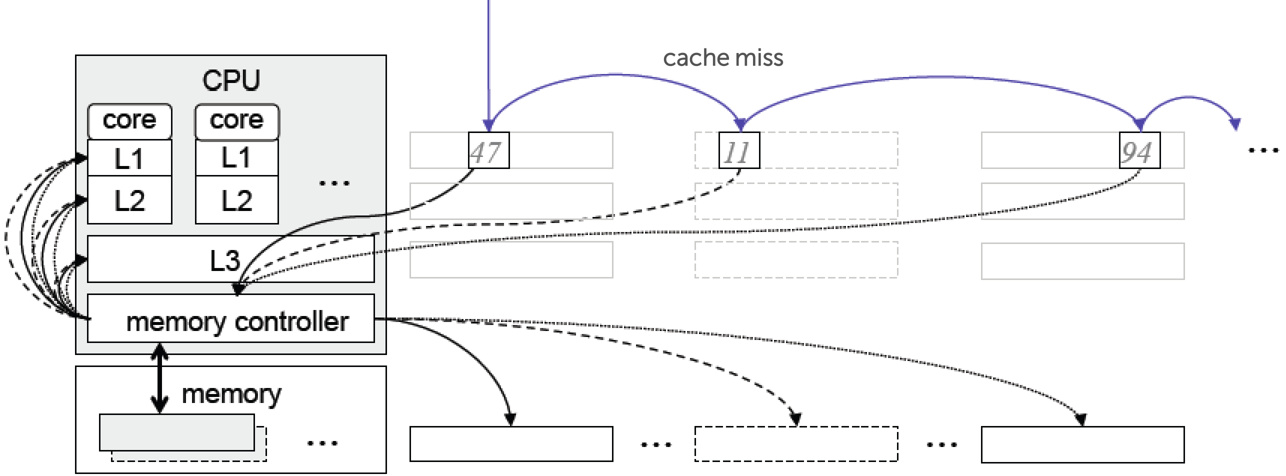
➔ linear memory access maximizes cache & bandwidth utilization

The Role of Caches

CACHES – THE BAD SIDE

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:

```
for (i=0; i<N; i++) x += A[rand()%N];
```



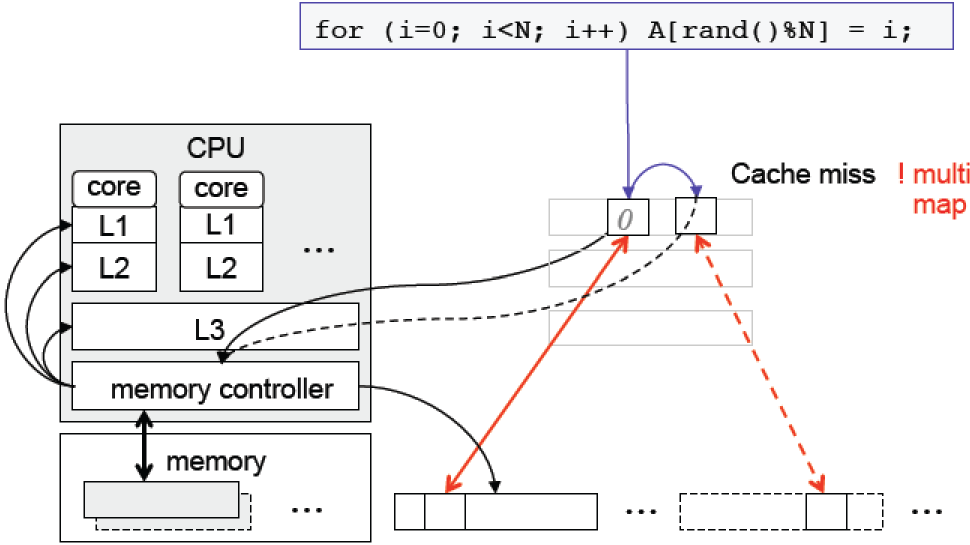
➔ Random memory access wastes up to 98.5%* of bandwidth

The Role of Caches

Dresden Database Systems Group

CACHES – THE UGLY

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes effectively turn into read-modify-write
  - Many memory addresses map into the same cache line(s)
  - "Dirty" cache line needs to be evicted before new one loads



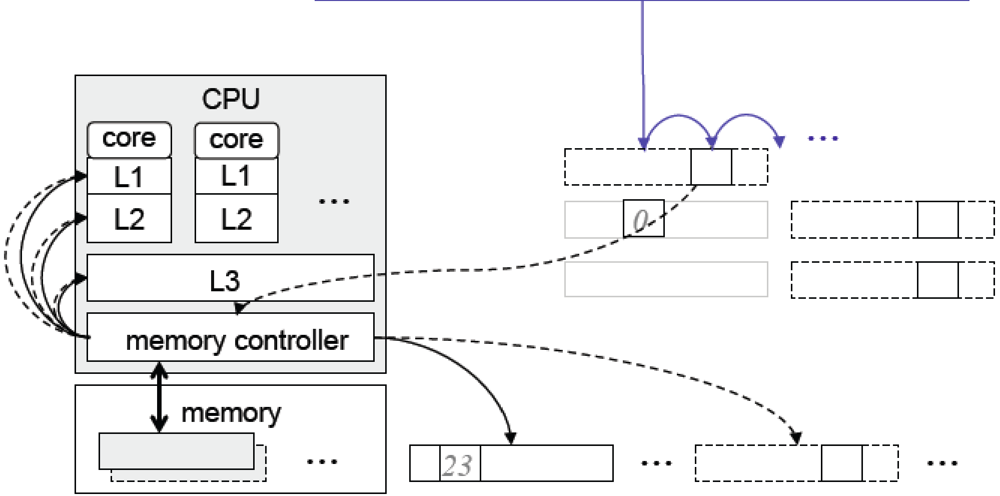
The Role of Caches

Dresden Database Systems Group

Caches – the Ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes effectively turn into read-modify-write
  - Many memory addresses map into the same cache line(s)
  - "Dirty" cache line needs to be evicted before new one loads

```
for (i=0; i<N; i++) A[rand()%N] = i;
```



SIMD = Single Instruction Multiple Data

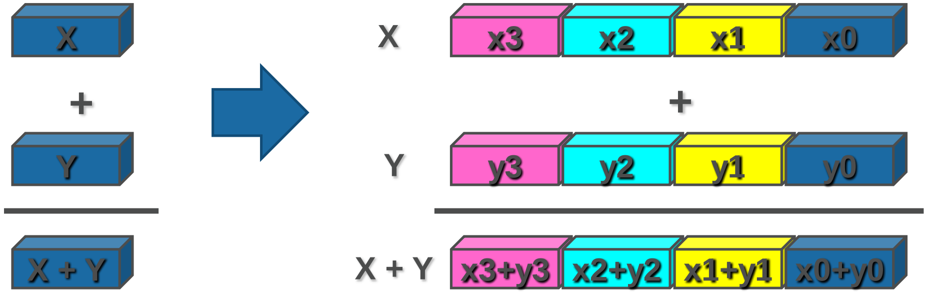
Dresden Database Systems Group

WHAT IS IT?

- Extension of the ISA. Data Types and instructions for the parallel computation on short vectors of integers or floats

Scalar processing

- traditional mode
- one operation produces one result



SIMD processing

- with SSE
- one operation produces multiple results

WHY DO THEY EXIST?

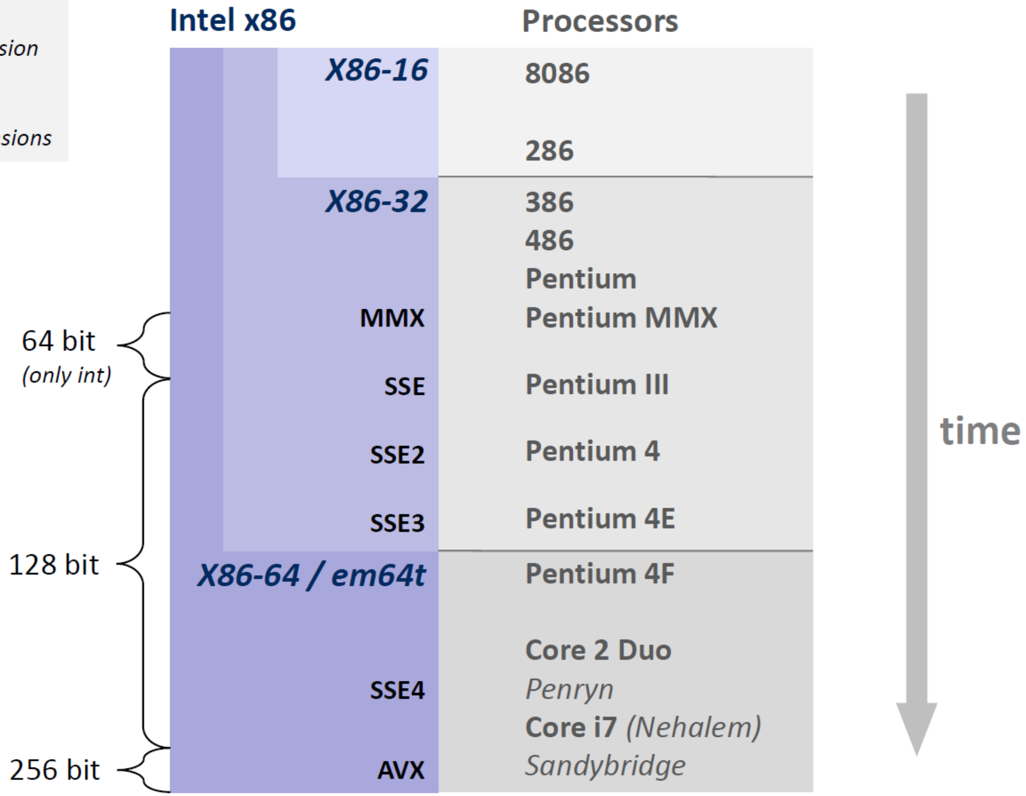
- Useful: Many applications have the necessary fine-grain parallelism
  Then: speedup by a factor close to vector length

- Doable: Chip designers have enough transistors to play with

Development

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions



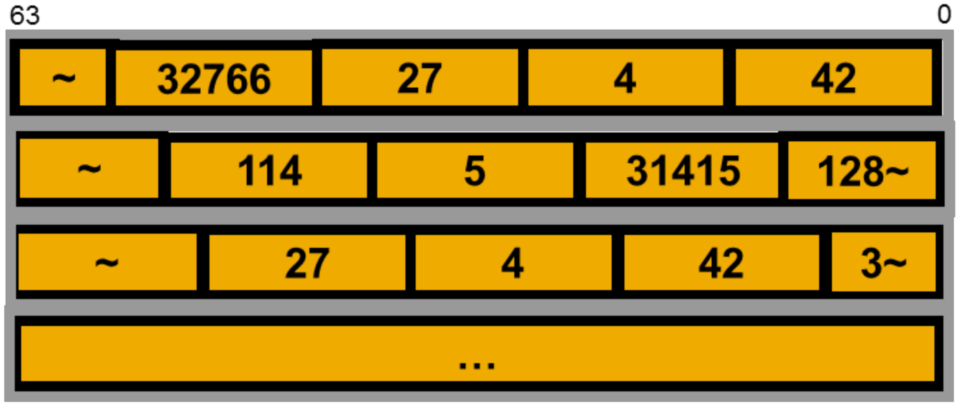
Example: Value Extraction

Example

- Dictionary encoding provide token with length of: 15 bit
- Input: vector of 64 bit values (concatenated 15 bit values)
- Work: extracted values are compared (rangeFrom <= value < rangeTo)
- Output: 32 bit integer holding the index of a match

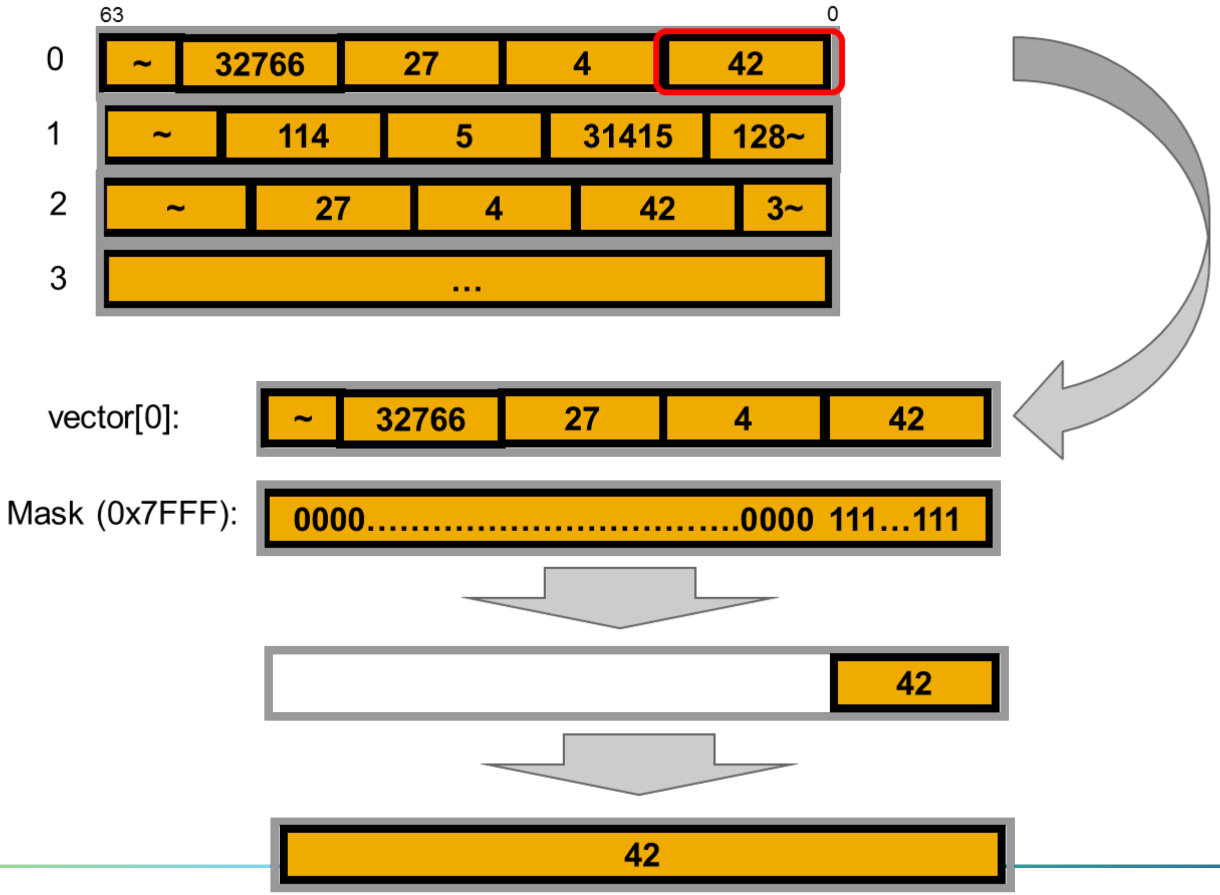
Code

- Unrolled extraction loop to extract 64 values at once
- 15 x 64 bit input integer → 64 x 15 bit values



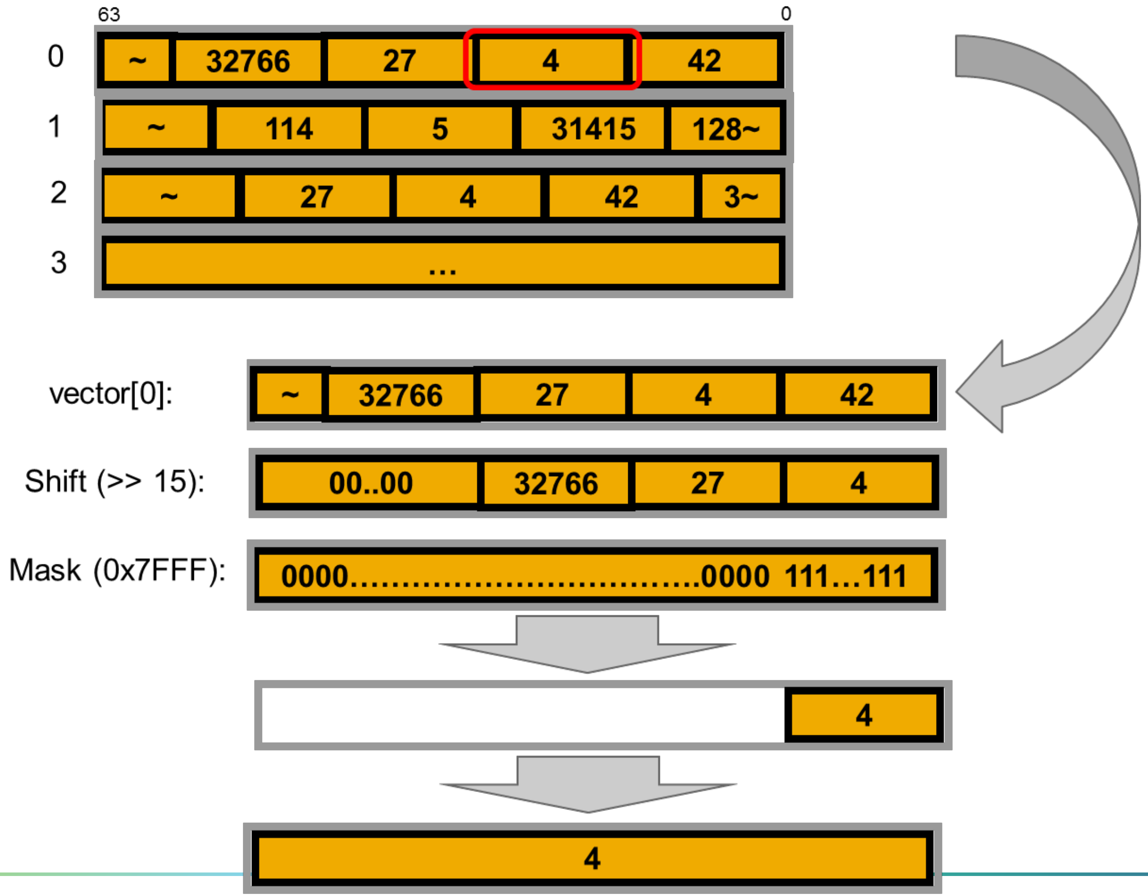
Example: Value Extraction

STEP 1:



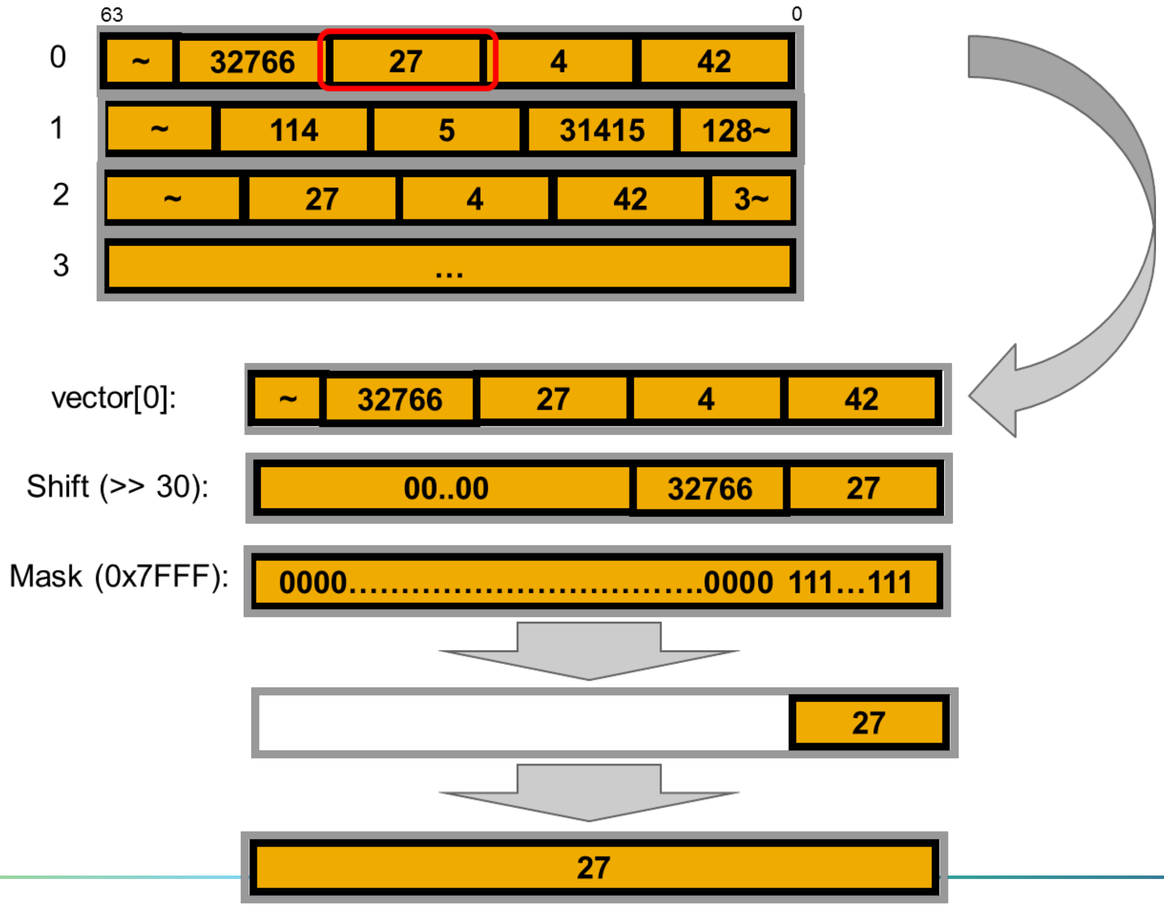
Example: Value Extraction

STEP 2:



Example: Value Extraction

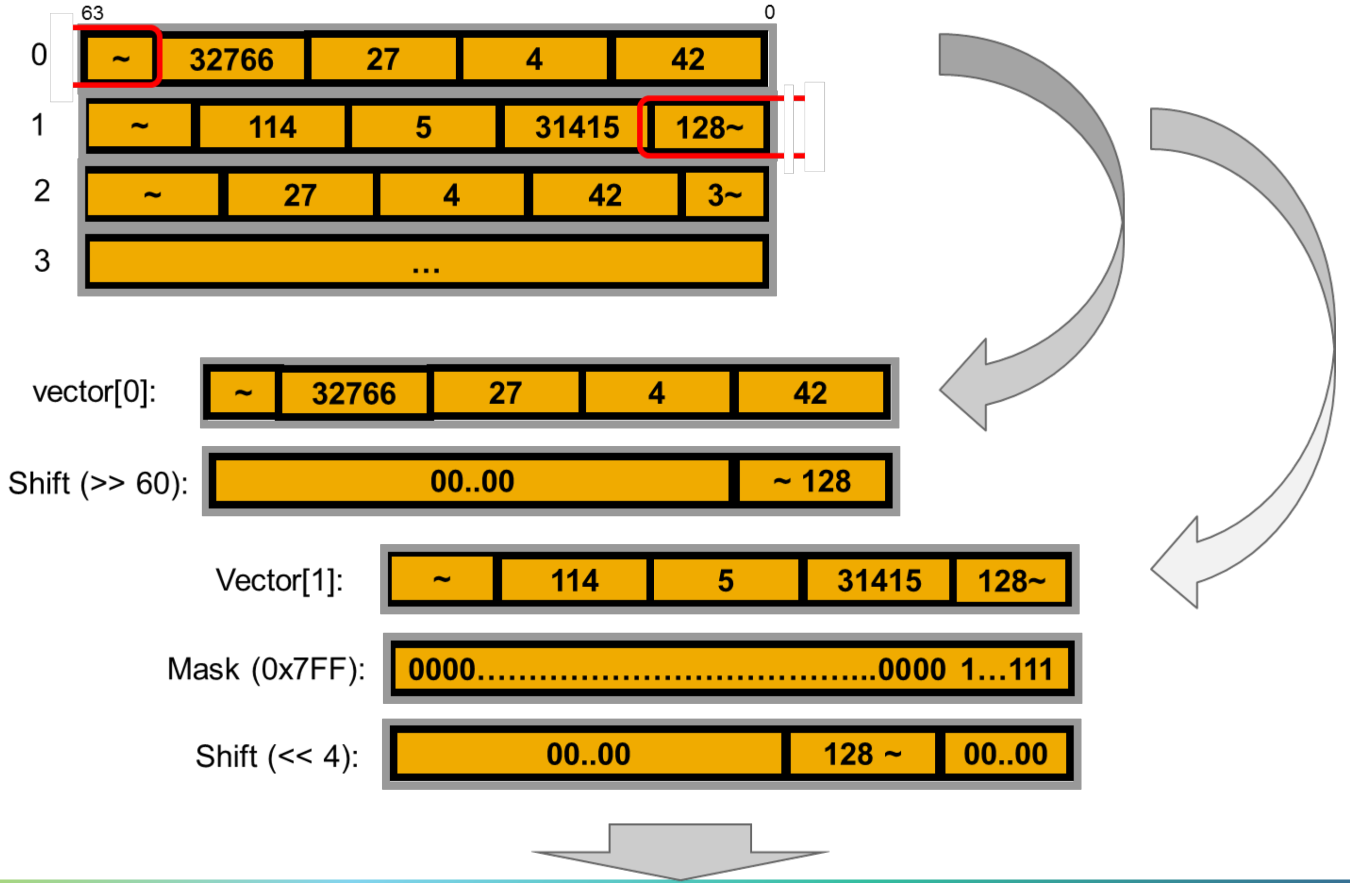
STEP 3:



Example: Value Extraction

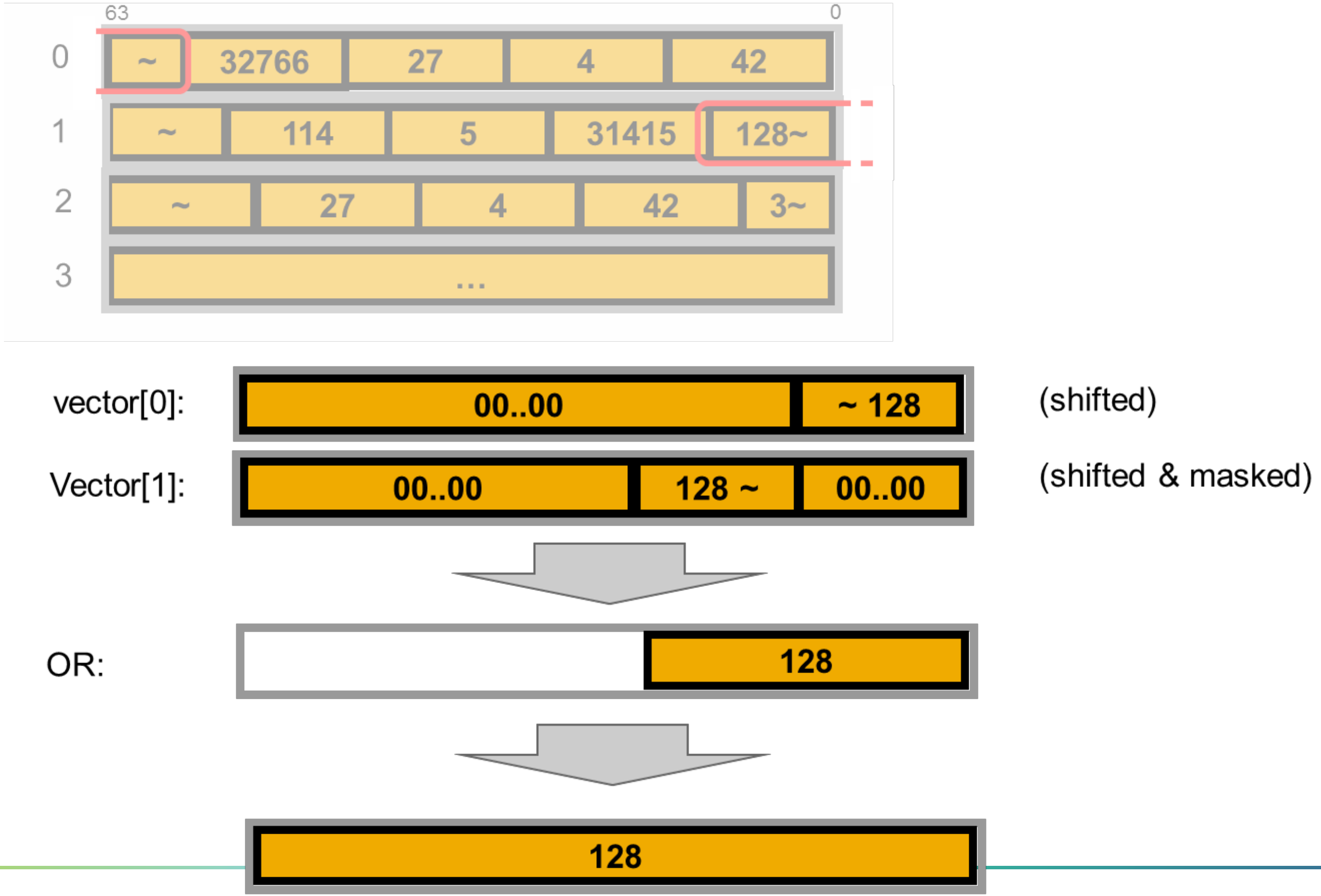
Dresden Database Systems Group

STEP 4:



Example: Value Extraction

STEP 4 CONT.:



BitWeaving

BitWeaving

Idea

- Fast scan method for column stores
- Two types of BitWeaving
  - BitWeaving/H (Horizontal bit organization)
  - BitWeaving/V (Vertical bit organization)

Storage layout

- Packs codes “horizontally” into processor words
- Uses an extra bit (delimiter bit) in each code
- Column-scalar scan: parallel predicate evaluation on packed codes

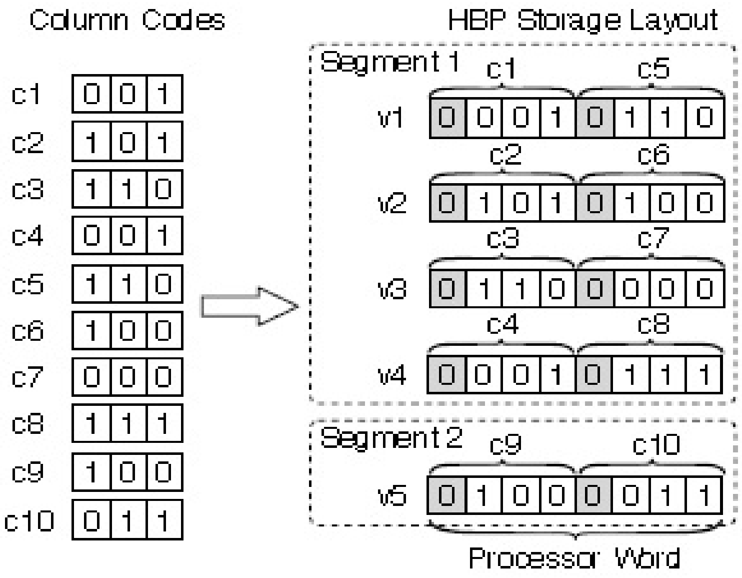
BitWeaving/H – Storage Layout

Horizontal Layout

- k denote the number of bits needed to encode a code
- each code is stored in a (k +1)-bit section whose leftmost bit is used as a delimiter between adjacent codes.
- w denotes the width of a processor word → inside the processor word, $\lfloor w/(k+1) \rfloor$ sections are concatenated together
- The column is divided into fixed-length segments, each of which contains (k+1)*$\lfloor w/(k+1) \rfloor$ codes

Query: c<5

	$v_1(c_1,c_5)$	$v_2(c_2,c_6)$	$v_3(c_3,c_7)$	$v_4(c_4,c_8)$	$v_5(c_9,c_{10})$
$X =$	$(0001\ 0110)_2$	$(0101\ 0100)_2$	$(0110\ 0000)_2$	$(0001\ 0111)_2$	$(0100\ 0011)_2$
$Y =$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$
$mask =$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$
$X \oplus mask =$	$(0110\ 0001)_2$	$(0010\ 0011)_2$	$(0001\ 0111)_2$	$(0110\ 0000)_2$	$(0011\ 0100)_2$
$Y + (X \oplus mask) =$	$(1011\ 0110)_2$	$(0111\ 1000)_2$	$(0110\ 1100)_2$	$(1011\ 0101)_2$	$(1001\ 1001)_2$
$Z = (Y + (X \oplus mask)) \wedge \neg mask =$	$(1000\ 0000)_2$	$(0000\ 1000)_2$	$(0000\ 1000)_2$	$(1000\ 0000)_2$	$(1000\ 1000)_2$



BitWeaving/H storage layout (k = 3; w = 8)
Delimiter bits are marked in gray.

BitWeaving/V

Storage Layout

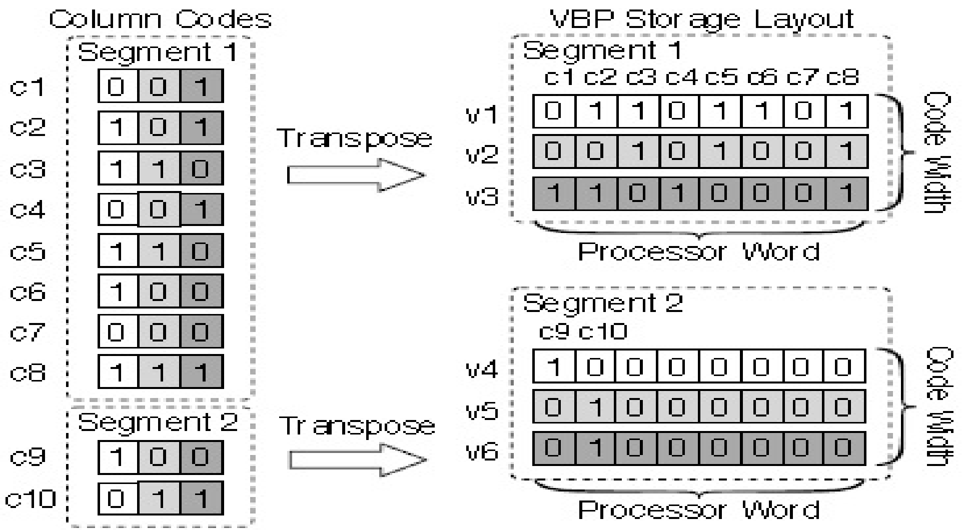
- Bit-level columnar data organization, i.e. its like a bit-level columnar store.

Column-scalar scan

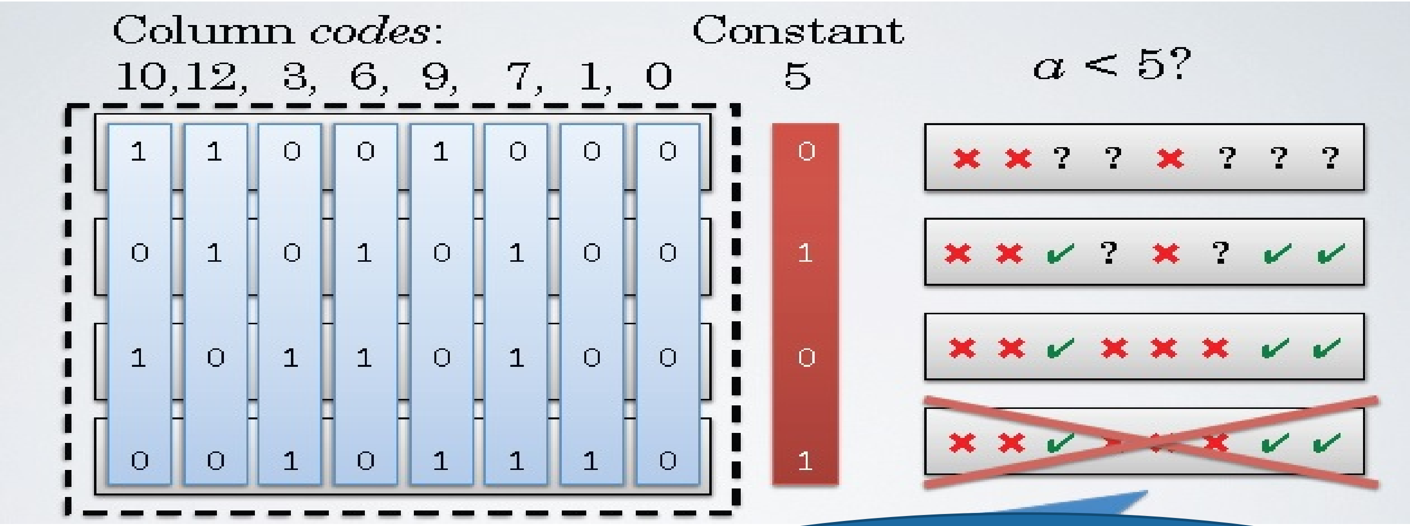
- Predicate evaluation is converted to logical computation on these "words of bits"

Example

- middle bits of codes are marked in light gray, whereas the least significant bits are marked in dark gray



Example BitWeaving/V



Query

- A<5

Early Pruning

- Predicate eval may stop as soon as all results are identified

Early Pruning: terminate the predicate evaluation on a segment, when all results have been determined.

Dresden Database Systems Group

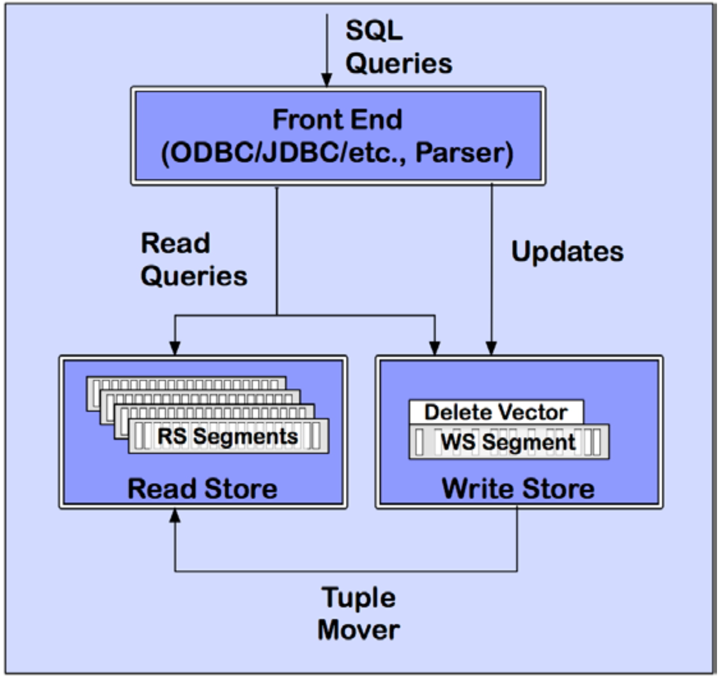
Architecture of commercial products

Vertica

Dresden Database Systems Group

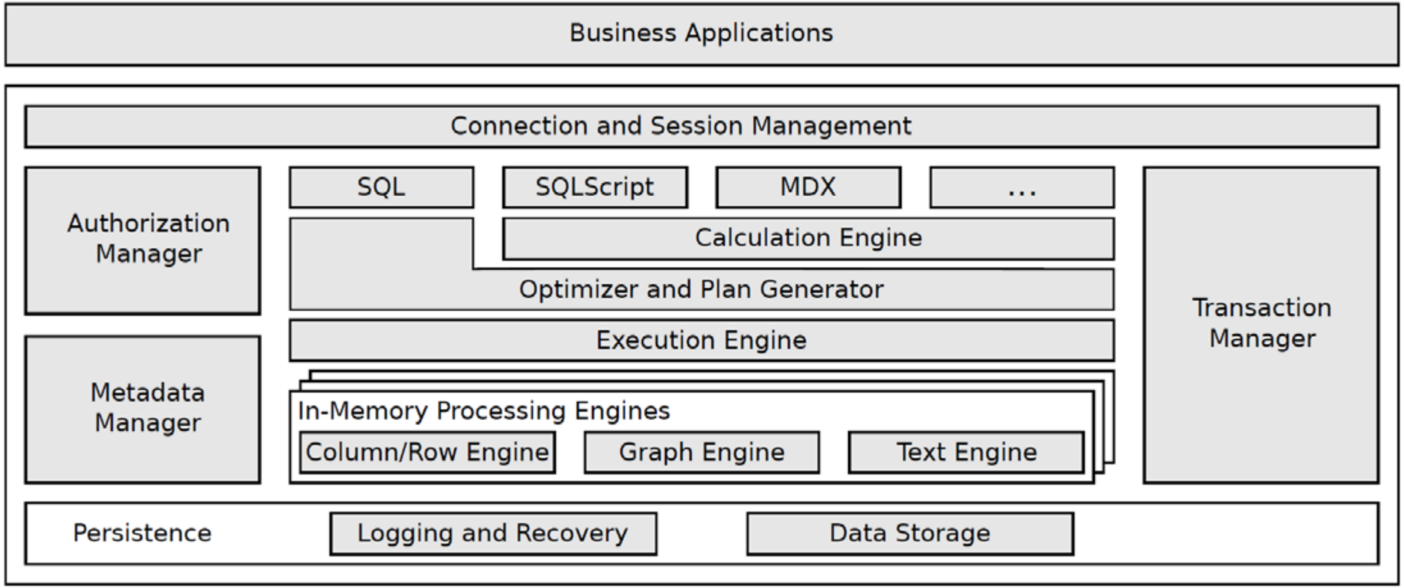
Vertica Analytic Database

- DBMS Optimized for Next-Generation Data Warehousing (OLAP)
- Hybrid Store consisting of two distinct storage structures
  - WOS (Write-Optimized Store): fits into main memory and is designed to efficiently support insert and update operations; WOS is unsorted and uncompressed
  - ROS (Read-Optimized Store): bulk of the data; sorted and compressed; making it efficient to read and query
- Tuple Mover
  - Moves data out of the WOS and into ROS
- Structure
  - WOS and ROS are organized as DMS



SAP HANA

Architecture



Hybrid Storage Architecture for Column Stores

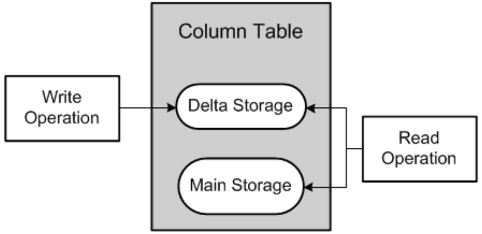
Dresden Database Systems Group

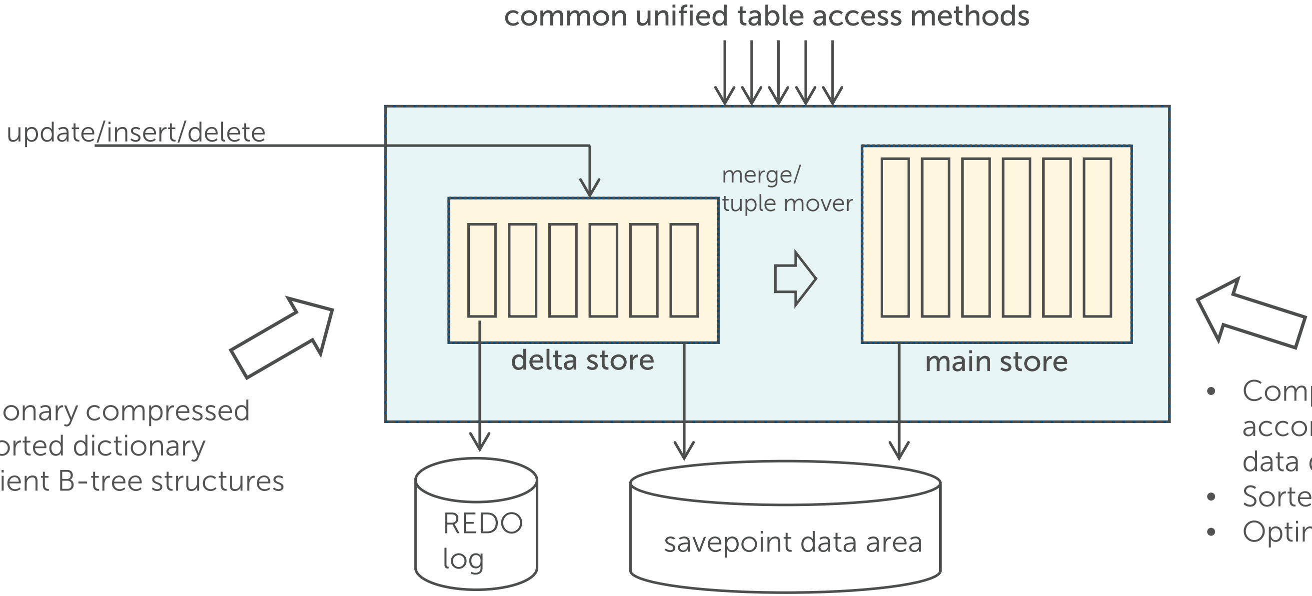
Use of compression implies two stores

- Write optimized store (WOS)
- Read optimized (compressed) store (ROS)

Merge Process

- Moves data from delta to main store





- Dictionary compressed
- Unsorted dictionary
- Efficient B-tree structures

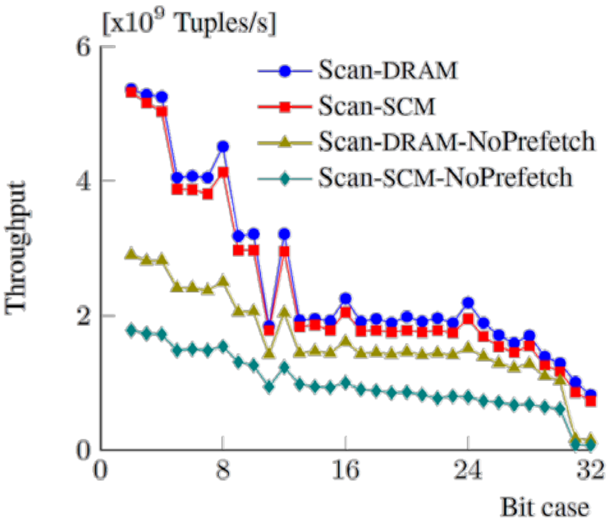
- Compression schemes according to existing data distribution
- Sorted dictionary
- Optimized for HW-scans

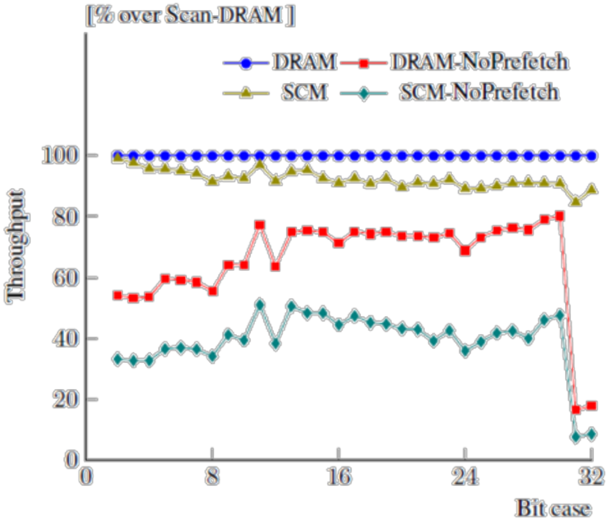
NVRAM for ROS-Structure

Micro Experiments: SIMD Scan performance on DRAM and SCM

- With prefetching: average penalty for using SCM instead of DRAM is only 8%.
- Without prefetching: average penalty for using SCM instead of DRAM is 41%.

NOTE For operators with sequential memory access patterns, SCM performs almost as good as DRAM





NVRAM for WOS-Structure

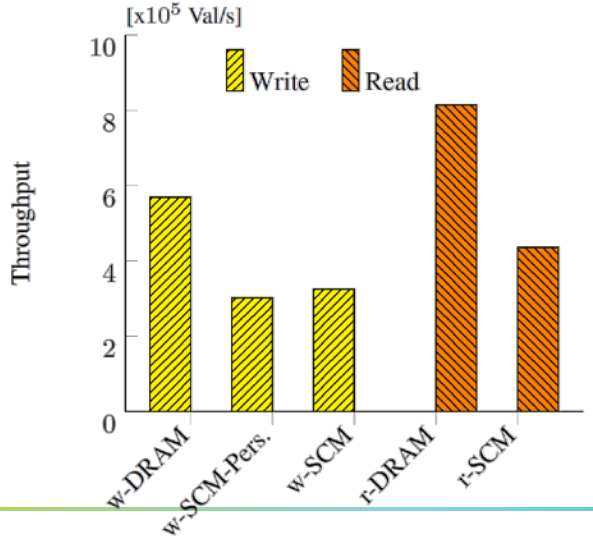
Dresden Database Systems Group

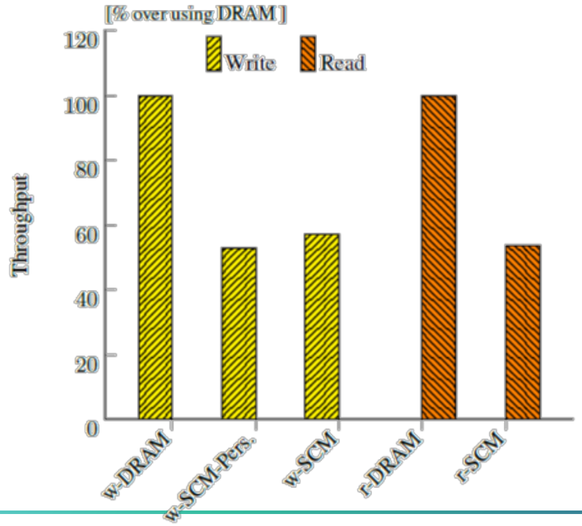
Micro Experiment

- Skip List read/write performance on DRAM and SCM
- 47% penalty for reads, and 43-47% penalty for writes for using SCM instead of DRAM.

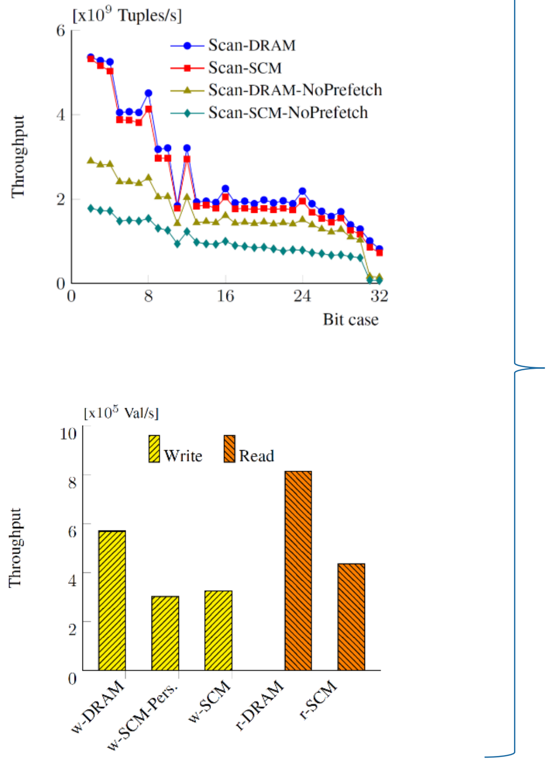
Operations with random memory access patterns are expensive in SCM

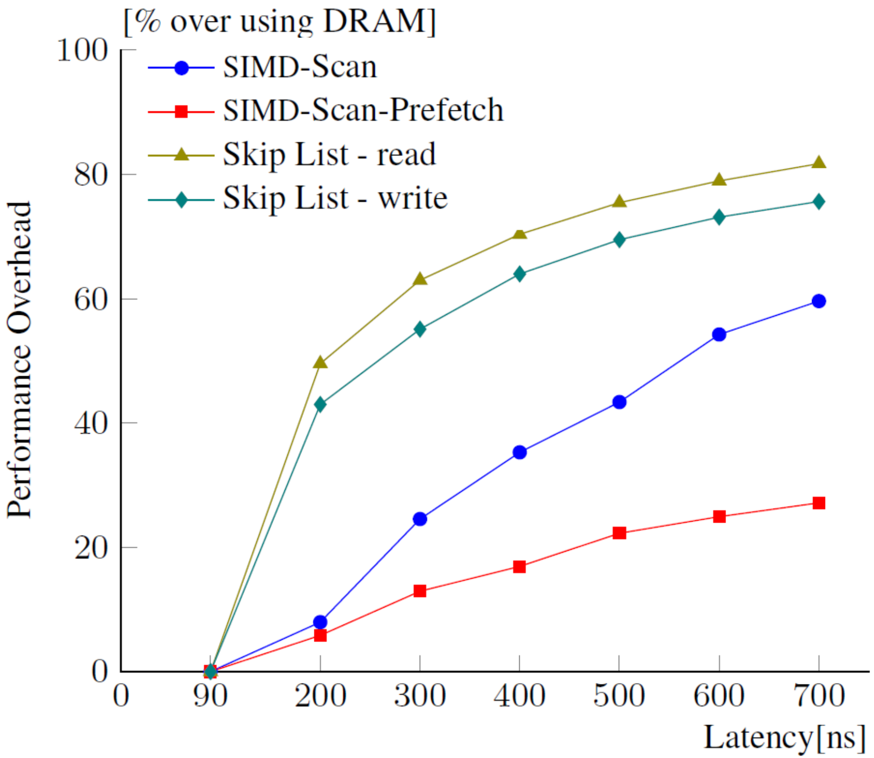
- Writing persistent and concurrent data structures is NOT trivial





Impact of NVRAM latency





SIMD: 16bit case / SL: 64Byte

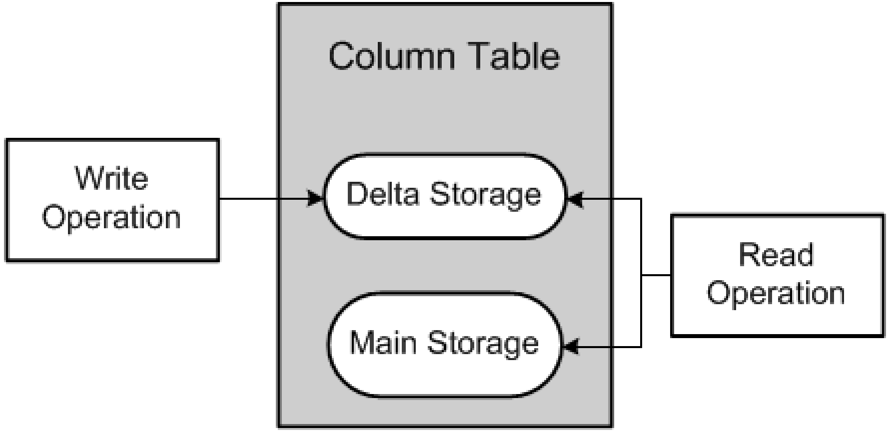
SAP HANA – Column Store

Main and Delta Store

- Main Store: main part of the data; compressed data
- Delta Store: all data changes are written; basic compression and optimized for write access

Merge Process

- Moves data from delta to main store



SAP HANA –Column Store/2

Dresden Database Systems Group

The Delta Merge Operation