Main-Memory
Database Management Systems

Sebastian Dorok
Bayer Pharma AG and Otto-von-Guericke University Magdeburg
Summer Term 2014
We will talk about

The Past and the Present

Cache Awareness

Processing Models

Storage Models

Design of Main-Memory DBMSs for Database Applications

Summary
Motivation: The Past and the Present
The Past - Computer Architecture

- **Latency:**
  - 5 ns
  - 10 ns
  - 100 ns
  - 5,000,000 ns

- **Capacity:**
  - 200 B
  - 64 KB
  - 32 MB
  - 2 GB

Data taken from [Hennessy and Patterson, 1996]
The Past - Database Systems

- Main memory capacity was limited to several megabytes
  → Only a small fraction of the database fit in main memory
The Past - Database Systems

- Main memory capacity was limited to several megabytes
  → Only a small fraction of the database fit in main memory
- Disk storage was "huge",
  → Traditional relational DBMSs use disks as primary storage
  → Architectural properties inherited from system R, the first "real" relational DBMS
  → From the 1970’s...
The Past - Database Systems

- Main memory capacity was limited to several megabytes
  → Only a small fraction of the database fit in main memory
- Disk storage was "huge",
  → Traditional relational DBMSs use disks as primary storage
  → Architectural properties inherited from system R, the first "real" relational DBMS
  → From the 1970’s...
- But disk latency is also high
  → Parallel query processing to hide disk latencies
The Present - Computer Architecture

- **Latency**
  - 300 ps
  - 1 ns
  - 3 - 10 ns
  - 10 - 20 ns
  - 50 - 100 ns
  - 5,000,000 - 10,000,000 ns

- **Capacity**
  - 1000 B
  - 64 kB
  - 256 kB
  - 2 - 4 MB
  - 4 - 16 GB
  - 4 - 16 TB

Data taken from [Hennessy and Patterson, 2012]
The Present - Database Systems

- Can have hundreds to thousands of GB main memory
  → Up to $10^6$ times more capacity!
  → Complete database having less than a TB size can be kept in main memory
  → Use **main memory as primary storage** for the database and remove disk access as main performance bottleneck
The Present - Database Systems

• Can have hundreds to thousands of GB main memory
  → Up to $10^6$ times more capacity!
  → Complete database having less than a TB size can be kept in main memory
  → Use main memory as primary storage for the database and remove disk access as main performance bottleneck

• But the architecture of traditional DBMSs is designed for disk-oriented database systems
  → ”30 years of Moore’s law has antiquated the disk-oriented relational architecture for OLTP applications.” [Stonebraker et al., 2007]
Disk-based vs. Main-Memory DBMS

Disk-based DBMS

Main Memory
- Buffered Data

Disk
- Data

Main-Memory DBMS

Main Memory
- Data

Disk
- Replicated Data
ATTENTION: Main-memory storage $\neq$ No Durability
$\rightarrow$ ACID properties have to be guaranteed
$\rightarrow$ However, there are new ways of guaranteeing it, such as a second machine in hot standby
Disk-based vs. Main-Memory DBMS (3)

Having the database in main memory allows us to remove buffer manager and paging → Removing one level of indirection → Results in better performance
Disk-based vs. Main-Memory DBMS (4)

Disk bottleneck is removed as database is kept in main memory → Access to main memory becomes new bottleneck

Disk-based DBMS

- CPU
- Buffered Data
- Old Bottleneck
- Disk
- Data

Main-Memory DBMS

- CPU
- Data
- New Bottleneck
- Replicated Data
The New Bottleneck: Memory Access

Picture taken from [Manegold et al., 2000]
Rethink the Architecture of DBMSs

Even if the complete database fits in main memory, there are significant overheads of traditional, system R like DBMSs:

- Function calls
- A lot of cache misses

→ We need to optimize algorithms and data structures to fully exploit the hardware!
→ Cache-aware data structures, algorithms with cache friendly memory access pattern, ...
Cache Awareness
The following slides are based on the lecture
Data Processing on Modern Hardware
by Jens Teubner from TU Dortmund.
A Motivating Example (Memory Access)

Task: sum up all entries in a two-dimensional array.

Alternative 1:

```c
for (r = 0; r < rows; r++)
    for (c = 0; c < cols; c++)
        sum += src[r * cols + c];
```

Alternative 2:

```c
for (c = 0; c < cols; c++)
    for (r = 0; r < rows; r++)
        sum += src[r * cols + c];
```

Both alternatives touch the same data, but in different order.
A Motivating Example (Memory Access)

cols

total execution time

rows

Sebastian Dorok
Main-Memory Database Management Systems
Last Change: May 13, 2014
Hardware Trends

There is an increasing **gap** between CPU and memory speeds.

- Also called the **memory wall**.
- CPUs spend much of their time **waiting** for memory.
### Memory Hierarchy

<table>
<thead>
<tr>
<th>Technology</th>
<th>Capacity</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>SRAM</td>
<td>&lt; 1 ns</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>SRAM</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>SRAM</td>
<td>&lt; 10 ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>DRAM</td>
<td>70–100 ns</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Some systems also use a 3rd level cache.
- cf. Architecture & Implementation course
  - Caches resemble the buffer manager but are **controlled by hardware**
Principle of Locality

Caches take advantage of the principle of locality.

- The hot set of data often fits into caches.
- 90% execution time spent in 10% of the code.

Spatial Locality:

- Related data is often spatially close.
- Code often contains loops.

Temporal Locality:

- Programs tend to re-use data frequently.
- Code may call a function repeatedly, even if it is not spatially close.
CPU Cache Internals

To guarantee speed, the **overhead** of caching must be kept reasonable.

- Organize cache in **cache lines**.
- Only load/evict **full cache lines**.
- Typical **cache line size**: 64 bytes.

- The organization in cache lines is consistent with the principle of (spatial) locality.
Memory Access

On every memory access, the CPU checks if the respective cache line is already cached.

Cache Hit:
- Read data directly from the cache.
- No need to access lower-level memory.

Cache Miss:
- Read full cache line from lower-level memory.
- Evict some cached block and replace it by the newly read cache line.
- CPU stalls until data becomes available.¹

¹Modern CPUs support out-of-order execution and several in-flight cache misses.
Example: **AMD Opteron**  
Data taken from [Hennessy and Patterson, 2006]

Example: AMD Opteron, 2.8 GHz, PC3200 DDR SDRAM

- **L1 cache**: separate data and instruction caches, each 64 kB, 64 B cache lines
- **L2 cache**: shared cache, 1 MB, 64 B cache lines
- **L1 hit latency**: 2 cycles ($\approx 1$ ns)
- **L2 hit latency**: 7 cycles ($\approx 3.5$ ns)
- **L2 miss latency**: 160–180 cycles ($\approx 60$ ns)
Block Placement: Fully Associative Cache

In a **fully associative** cache, a block can be loaded into **any** cache line.

- Offers freedom to block replacement strategy.
- Does not scale to large caches
  - 4 MB cache, line size: 64 B: 65,536 cache lines.
- Used, *e.g.*, for small TLB caches.
Block Placement: Direct-Mapped Cache

In a **direct-mapped** cache, a block has only one place it can appear in the cache.

- **Much** simpler to implement.
- Easier to make **fast**.
- Increases the chance of **conflicts**.

01234567

place block 12 in cache line 4
(4 = 12 mod 8)

01234567890123456789012345678901
A compromise are set-associative caches.

- Group cache lines into sets.
- Each memory block maps to one set.
- Block can be placed anywhere within a set.
- Most processor caches today are set-associative.
Effect of Cache Parameters

Data taken from [Drepper, 2007]

- 512 kB
- 1 MB
- 2 MB
- 4 MB
- 8 MB
- 16 MB

- direct-mapped
- 2-way associative
- 4-way associative
- 8-way associative

Sebastian Dorok
Main-Memory Database Management Systems
Last Change: May 13, 2014
27/113
Block Identification

A tag associated with each cache line identifies the memory block currently held in this cache line.

The tag can be derived from the memory address.

![Diagram showing cache line structure with fields for status, tag, and data, along with byte address and block address with tag, set index, and offset.]
Example: Intel Q6700 (Core 2 Quad)

- Total cache size: **4 MB** (per 2 cores).
- Cache line size: **64 bytes**.
  → 6-bit offset ($2^6 = 64$)
  → There are 65,536 cache lines in total (4 MB ÷ 64 bytes).
- Associativity: **16-way set-associative**.
  → There are 4,096 sets (65,536 ÷ 16 = 4,096).
  → 12-bit set index ($2^{12} = 4,096$).
- Maximum physical address space: **64 GB**.
  → 36 address bits are enough ($2^{36}$ bytes = 64 GB)
  → 18-bit tags (36 − 12 − 6 = 18).
Block Replacement

When bringing in new cache lines, an existing entry has to be **evicted**: Least Recently Used (LRU)
  - Evict cache line whose last access is longest ago.
    → Least likely to be needed any time soon.

First In First Out (FIFO)
  - Behaves often similar like LRU.
  - But easier to implement.

Random
  - Pick a random cache line to evict.
  - Very simple to implement in hardware.

Replacement has to be decided **in hardware** and **fast**.
What Happens on a Write?

To implement memory writes, CPU makers have two options:

**Write Through**

- Data is directly written to lower-level memory (and to the cache).
  
  → Writes will **stall the CPU**.\(^2\)
  
  → Greatly simplifies **data coherency**.

**Write Back**

- Data is only written into the cache.

- A **dirty** flag marks modified cache lines (Remember the status field.)
  
  → May reduce traffic to lower-level memory.
  
  → Need to write on eviction of dirty cache lines.

Modern processors usually implement **write back**.

\(^2\)**Write buffers** can be used to overcome this problem.
Putting it all Together

To compensate for slow memory, systems use caches.

- DRAM provides high capacity, but long latency.
- SRAM has better latency, but low capacity.
- Typically multiple levels of caching (memory hierarchy).
- Caches are organized into cache lines.
- Set associativity: A memory block can only go into a small number of cache lines (most caches are set-associative).

Systems will benefit from locality of data and code.
Performance (SPECint 2000)

- Benchmark programs:
  - gzip, vpr, gcc, mcf, crafty, parser, eon, perlbench, gap, vortex, bzip2, twolf, avg

- Metrics:
  - L1 Instruction Cache
  - L2 Cache (shared)

- Y-axis: misses per 1000 instructions
  - Range: 0 to 20

- X-axis: benchmark program
  - Categories: gzip, vpr, gcc, mcf, crafty, parser, eon, perlbench, gap, vortex, bzip2, twolf, avg
Performance (SPECint 2000)

misses per 1000 instructions

- **L1 Instruction Cache**
- **L2 Cache (shared)**

Benchmark programs:
- gzip
- vpr
- gcc
- mcf
- crafty
- parser
- eon
- perlbench
- gap
- vortex
- bzip2
- twolf
- avg
- TPC-C
Assessment

Why do database systems show such poor cache behavior?

Poor code locality:

• Polymorphic functions
  \((E.g.,\) resolve attribute types for each processed tuple individually.\)

• Volcano iterator model (pipelining)
  Each tuple is passed through a query plan composed of many operators.

Poor data locality:

• Database systems are designed to navigate through large data volumes quickly.

• Navigating an index tree, e.g., by design means to “randomly” visit any of the (many) child nodes.
Processing Models

Or how to improve the instruction cache effectiveness?
Processing Models

There are basically two alternative processing models that are used in modern DBMSs:

- **Tuple-at-a-time volcano model** [Graefe, 1990]
  - Operator requests next tuple, processes it, and passes it to the next operator

- **Operator-at-a-time bulk processing** [Manegold et al., 2009]
  - Operator consumes its input and materializes its output
Tuple-At-A-Time Processing

Most systems implement the **Volcano iterator model**:

- Operators request tuples from their input using `next()`.
- Data is processed **tuple at a time**.
- “pipelining”
- Each operator keeps its own **state**.
- ↗ DB implementation course

![Diagram of tuple-at-a-time processing]

```plaintext
Operator 1
next () —> tuple

Operator 2
next () —> tuple

Operator 3
next () —> tuple

... tuple

Database
```
Tuple-At-A-Time Processing - Consequences

- Pipeline-parallelism
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results
Tuple-At-A-Time Processing - Consequences

- **Pipeline-parallelism**
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results

- All operators in a plan run tightly interleaved.
  - Their combined instruction footprint may be large.
  - Instruction cache misses.
Tuple-At-A-Time Processing - Consequences

- **Pipeline-parallelism**
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results

- All operators in a plan run **tightly interleaved**.
  - Their combined instruction footprint may be large.
  - Instruction cache misses.

- Operators constantly call each other’s functionality.
  - Large function call overhead.
Tuple-At-A-Time Processing - Consequences

• Pipeline-parallelism
  → Data processing can start although data does not fully reside in main memory
  → Small intermediate results

• All operators in a plan run tightly interleaved.
  → Their combined instruction footprint may be large.
  → Instruction cache misses.

• Operators constantly call each other’s functionality.
  → Large function call overhead.

• The combined state may be too large to fit into caches.
  • E.g., hash tables, cursors, partial aggregates.
  → Data cache misses.
Operator-At-A-Time Processing

- Operators consume and produce full tables.
- Each (sub-)result is **fully materialized** (in memory).
- **No** pipelining (rather a sequence of statements).
- Each operator runs exactly once.

![Diagram](attachment:image.png)
Operator-At-A-Time Consequences

• Parallelism: **Inter-operator** and **intra-operator**
Operator-At-A-Time Consequences

- Parallelism: **Inter-operator** and **intra-operator**
- Function call overhead is now replaced by **extremely tight loops** that
  - conveniently **fit into instruction caches**,
  - can be **optimized** effectively by modern compilers
    → **loop unrolling**
    → **vectorization** (use of SIMD instructions)
  - can leverage modern CPU features (**hardware prefetching**).
Operator-At-A-Time Consequences

- Parallelism: **Inter-operator** and **intra-operator**
- Function call overhead is now replaced by **extremely tight loops** that
  - conveniently **fit into instruction caches**,
  - can be **optimized** effectively by modern compilers
    - loop unrolling
    - **vectorization** (use of SIMD instructions)
  - can leverage modern CPU features (**hardware prefetching**).
- Function calls are now **out of the critical code path**.
Tuple-At-A-Time vs. Operator-At-A-Time

The operator-at-a-time model is a two-edged sword:

😊 Cache-efficient with respect to code and operator state.
😊 Tight loops, optimizable code.
Tuple-At-A-Time vs. Operator-At-A-Time

The **operator-at-a-time model** is a two-edged sword:

- **ㄇ** Cache-efficient with respect to code and operator state.
- **ㄇ** Tight loops, optimizable code.

- **放心 Data won’t fully fit into cache.**
  - → Repeated scans will fetch data **from memory** over and over.
  - → Strategy falls apart when intermediate results no longer fit into main memory.
Tuple-At-A-Time vs. Operator-At-A-Time

The operator-at-a-time model is a two-edged sword:

- Cache-efficient with respect to code and operator state.
- Tight loops, optimizable code.

- Data won’t fully fit into cache.
  - Repeated scans will fetch data from memory over and over.
  - Strategy falls apart when intermediate results no longer fit into main memory.

Can we aim for the middle ground between the two extremes?

```
tuple-at-a-time ← vectorized execution → operator-at-a-time
```
Vectorized Execution Model

Idea:

• Use Volcano-style iteration,

but:

• for each `next()` call **return a large number of tuples**
  → a so called “vector”
Vectorized Execution Model

Idea:

- Use Volcano-style iteration,

but:

- for each \texttt{next()} call \texttt{return a large number of tuples}
  → a so called “vector”

Choose vector size

- \textbf{large enough} to compensate for iteration overhead (function calls, instruction cache misses, \ldots), but

- \textbf{small enough} to not thrash data caches.
Vector Size ↔ Instruction Cache Effectiveness

- Vectorized execution quickly compensates for iteration overhead.
- 1000 tuples should conveniently fit into caches.

Source: [Zukowski, 2009]
### Comparison of Execution Models

<table>
<thead>
<tr>
<th>Execution Model</th>
<th>Tuple</th>
<th>Operator</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>Function calls</td>
<td>many</td>
<td>extremely few</td>
<td>very few</td>
</tr>
<tr>
<td>Attribute access</td>
<td>complex</td>
<td>direct</td>
<td>direct</td>
</tr>
<tr>
<td>Most time spent on</td>
<td>interpretation</td>
<td>processing</td>
<td>processing</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>poor</td>
<td>good</td>
<td>very good</td>
</tr>
<tr>
<td>Compiler optimizations</td>
<td>limited</td>
<td>applicable</td>
<td>applicable</td>
</tr>
<tr>
<td>Materialization overhead</td>
<td>very cheap</td>
<td>expensive</td>
<td>cheap</td>
</tr>
<tr>
<td>Scalability</td>
<td>good</td>
<td>limited</td>
<td>good</td>
</tr>
</tbody>
</table>

Source [Zukowski, 2009]
Storage Models

Or how to improve the data cache effectiveness?
Consider, e.g., a selection query:

```
SELECT COUNT(*)
FROM lineitem
WHERE l_shipdate = "2009-09-26"
```

This query typically involves a **full table scan**.
Query Processing - Full Table Scan

Tuples are represented as records stored sequentially on a database page.

```
<table>
<thead>
<tr>
<th>record</th>
<th>l_shipdate</th>
</tr>
</thead>
</table>
```

With every access to an \( l\_\text{shipdate} \) field, we load a large amount of irrelevant information into the cache.
Query Processing - Full Table Scan

Tuples are represented as records stored sequentially on a database page.

With every access to a l_shipdate field, we load a large amount of irrelevant information into the cache.
Row-Wise Storage

Row-wise storage \( (n\text{-}ary\ storage\ model,\ NSM):\)

Row-wise storage:

\[
\begin{array}{cccc}
\text{p} & \text{b} & \text{c} & \text{d} \\
\hline
\text{p}_1 & \text{b}_1 & \text{c}_1 & \text{d}_1 \\
\text{p}_2 & \text{b}_2 & \text{c}_2 & \text{d}_2 \\
\text{p}_3 & \text{b}_3 & \text{c}_3 & \text{d}_3 \\
\text{p}_4 & \text{b}_4 & \text{c}_4 & \text{d}_4 \\
\end{array}
\]

Column-wise storage (decomposition storage model, DSM):

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
\text{a}_1 & \text{b}_1 & \text{c}_1 & \text{d}_1 \\
\text{a}_2 & \text{b}_2 & \text{c}_2 & \text{d}_2 \\
\text{a}_3 & \text{b}_3 & \text{c}_3 & \text{d}_3 \\
\text{a}_4 & \text{b}_4 & \text{c}_4 & \text{d}_4 \\
\end{array}
\]
Row-Wise vs. Column-Wise Storage

Row-wise storage (n-ary storage model, NSM):

Column-wise storage (decomposition storage model, DSM):
Column-Wise Storage

- All data loaded into caches by a "l_shipdate scan" is now actually relevant for the query.
  → Less data has to be fetched from memory.
  → Amortize cost for fetch over more tuples.
  → If we’re really lucky, the full (l_shipdate) data might now even fit into caches.

- The same arguments hold, by the way, also for disk-based systems.

- Additional benefit: Data compression might work better.
MonetDB: Binary Association Tables

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

<table>
<thead>
<tr>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>
MonetDB: Binary Association Tables

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>o₂</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>o₃</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>o₄</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

- Each column yields one **binary association table (BAT)**.
- **Object identifiers** (oids) identify matching entries (BUNs).
MonetDB: Binary Association Tables

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>$o_2$</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>$o_3$</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>$o_4$</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>$o_2$</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>$o_3$</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>$o_4$</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

- Each column yields one **binary association table (BAT)**.
- **Object identifiers** (oids) identify matching entries (BUNs).
- Often, oids can be implemented as **virtual oids (voids)**.
  - → Not explicitly materialized in memory.
Tuple recombination can cause considerable cost.

- Need to perform many joins.
- Workload-dependent trade-off.

→ MonetDB: positional joins (thanks to void columns)
Main-Memory DBMSs for OLAP
Online Analytical Processing (OLAP)
Online Analytical Processing (OLAP)

read-mostly
Online Analytical Processing (OLAP)

analysis mostly on **few columns** and over **all rows**
Online Analytical Processing (OLAP)

query response time is important
Main-Memory DBMSs for OLAP

- read-mostly
  - Data is typically not modified or updated instead new data is just inserted
  - Append new data via bulk loading
Main-Memory DBMSs for OLAP

- **read-mostly**
  - Data is typically not modified or updated instead new data is just inserted
  - Append new data via bulk loading

- **analysis mostly on few columns and over all rows**
  - Use column-oriented data layout
  - Yields better data cache utilization
Main-Memory DBMSs for OLAP

• read-mostly
  → Data is typically not modified or updated instead new data is just inserted
  → Append new data via bulk loading

• analysis mostly on few columns and over all rows
  → Use column-oriented data layout
  → Yields better data cache utilization

• query response time is important
  → Bulk processing to improve instruction cache effectiveness
  → Reduce amount of data to process by
    • lightweight compression techniques
    • optimizing query processing
Main-Memory DBMSs for OLAP

- Main memory as primary storage
- Column-oriented data layout
- Lightweight compression
- Optimized query processing

Main-Memory DBMS for **efficient** OLAP
Main-Memory DBMSs for OLAP

- Main memory as primary storage  
- Column-oriented data layout
- Lightweight compression
- Optimized query processing

Main-Memory DBMS for efficient OLAP
Data Compression - Motivation

Reduce size of data
Data Compression - Motivation

Reduce size of data

→ **Reduced costs** for storage as we need less storage space to store the same amount of data
→ **More data** can be stored using the same amount of storage space
→ Better utilization of **memory bandwidth**
Data Compression - Requirements

• Lossless compression → otherwise we generate data errors
• Lightweight (de-)compression → otherwise (de-)compression overhead would outweigh our possible performance potentials
• Enable processing of compressed values → no additional overhead for decompression
Data Compression - Requirements

• Lossless compression
  → otherwise we generate data errors
Data Compression - Requirements

- Lossless compression
  → otherwise we generate data errors

- Lightweight (de-)compression
  → otherwise (de-)compression overhead would outweigh our possible performance potentials
Data Compression - Requirements

• Lossless compression
  → otherwise we generate data errors

• Lightweight (de-)compression
  → otherwise (de-)compression overhead would outweigh our possible performance potentials

• Enable processing of compressed values
  → no additional overhead for decompression
Classification of compression techniques

• General idea: Replace data by representation that needs less bits than original data
Classification of compression techniques

- General idea: Replace data by representation that needs less bits than original data
- Granularity:
  - Attribute values, tuples, tables, pages
  - Index structures
Classification of compression techniques

- General idea: Replace data by representation that needs less bits than original data
- Granularity:
  - Attribute values, tuples, tables, pages
  - Index structures
- Code length:
  - **Fixed code length**: All values are encoded with same number of bits
  - **Variable code length**: Number of bits differs (e.g., correlate number of used bits with value frequency; Huffman Encoding)
Dictionary Encoding

- Use a dictionary that contains data values and their surrogates
- Surrogate can be derived from values’ dictionary position
- Applicable to row- and column-oriented data layouts
Bit Packing

- Surrogate values do not have to be multiples of one byte
- Example: 16 distinct values can be effectively stored using 4 bit per surrogate → 2 values per byte

→ Processing of compressed values is not straightforward
Run Length Encoding

- Reduce size of sequences of same value
- Store the value and an indicator about the sequence length
- Applicable to column-oriented data layouts
- Sorting can further improve compression effectiveness

### Example

<table>
<thead>
<tr>
<th>...</th>
<th>State</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bavaria</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hesse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hesse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saxony</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sx. Anhalt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sx. Anhalt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thuringia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thuringia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bavaria</td>
</tr>
<tr>
<td>2</td>
<td>Hesse</td>
</tr>
<tr>
<td>1</td>
<td>Saxony</td>
</tr>
<tr>
<td>2</td>
<td>Sx. Anhalt</td>
</tr>
<tr>
<td>3</td>
<td>Thuringia</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

---

Run Length Encoding (RLE)

### Dictionary Encoding and RLE

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Common Value Suppression

- A common value is scattered across a column (e.g., *null*)
- Use a data structure that indicates whether a common value is stored at a given row index or not
  - Yes: Common value is stored here
  - No: Lookup value in the dictionary (using prefix sum)
- Applicable to column-oriented data layouts

<table>
<thead>
<tr>
<th>State</th>
<th>Common Value</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thuringia</td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td><em>NULL</em></td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>Saxony</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td><em>NULL</em></td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>Hesse</td>
<td>1</td>
<td>0011</td>
</tr>
<tr>
<td>Bavaria</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td><em>NULL</em></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Sx. Anhalt</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><em>NULL</em></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Exception Values:

- Bavaria: 0000
- Hesse: 0001
- Saxony: 0010
- Sx. Anhalt: 0011
- Thuringia: 0100
Bit-Vector Encoding

- Suitable for columns that have low number of distinct values
- Use bit string for every column value that indicates whether the value is present at a given row index or not
- Length of bit string equals number of tuples
- Used in Bitmap-Indexes
Delta Coding

- Store difference to precedent value instead of the original value
- Applicable to column-oriented data layouts
- Sorting can further improve compression effectiveness

<table>
<thead>
<tr>
<th>Postal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>39104</td>
</tr>
<tr>
<td>39106</td>
</tr>
<tr>
<td>39108</td>
</tr>
<tr>
<td>39130</td>
</tr>
<tr>
<td>80336</td>
</tr>
<tr>
<td>80339</td>
</tr>
<tr>
<td>80807</td>
</tr>
<tr>
<td>80809</td>
</tr>
<tr>
<td>80933</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>39104</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>41206</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>468</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>124</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
**Frequency Compression**

- **Idea:** Exploit data skew
- **Principle:**
  - More frequent values are encoded with fewer bits using
  - Less frequent values are encoded with more bits using
- **Use prefix codes (e.g., Huffman Encoding)**

![Diagram](image)
Frequency Compression - Column Store

Keep track where encoded values start and end
→ Use fixed code length for a partition rather than unique code for every value (i.e., Huffman Encoding)

**Region 1**
**Bank 1:** 3 bit tuples in 128-bit words, with 2 bits of padding

**Region 2**
**Bank 1:** 8 bit tuples in 64-bit words (no padding)

**Tuple Map**
101001000... (1st, 3rd, 6th entries are in region 1)

Picture taken from [Raman et al., 2013]
Frequency Compression - Row Store

Apply frequency compression on each column and perform additional delta coding

→ Introduces overhead for reading $n$th column value (2 ns per column value)
Frequency Partitioning

- Developed for IBM’s BLINK project [Raman et al., 2008]
- Similar to frequency compression of tuples in a row-oriented data layout
- But, partitioning tuples regarding column values
  → Overhead is reduced as within one partition code length is fixed
Frequency Partitioning: Principle

Original Table

Col 1
Col 2

Number of Occurrences
Frequencies of Values in Col 2
Common Values Rare values

Column Partitions for Col 2

Column 2 Values

Frequencies of Values in Col 1
Column Partitions For Col 1

Column 1 Values
Table partitioned into Cells

Picture taken from [Raman et al., 2008]
Frequency Partitioning: Partitioning

- Sort column values by frequency and partition into a fixed number of intervals
- Interval size is a power of two
- Optimal partitioning of column using dynamic computing: Number of values with high frequency is less than number of values with lower frequency
- Optimal partitioning across columns using greedy strategy: Which column benefits most from an additional partition?
Data Compression - Summary

• General idea: Replace data by representation that needs less bits than original data

• Discussed approaches:
  • **Fixed code length**: Dictionary Encoding, RLE, Common Value Suppression
  • **Variable code length**: Delta Coding, Frequency Compression, Frequency Partitioning

• Improvements: bit packing, partitioning

• Benefits for main-memory DBMSs:
  • Reduced storage requirements
  • Better memory bandwidth utilization
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>In-memory size in MiB</th>
<th>Compression Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MonetDB</td>
<td>SAP</td>
<td>HANA</td>
</tr>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>2,662.777</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>4,495.983</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.076</td>
</tr>
<tr>
<td>4</td>
<td>HG00096</td>
<td>base</td>
<td>1,030.635</td>
<td>0.771</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>1.758</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>2,614.746</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>1,782.782</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.003</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>107.552</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.010</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>11,666.458</td>
</tr>
</tbody>
</table>
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>MonetDB</th>
<th>SAP HANA</th>
<th>HANA</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>2,662.777</td>
<td>64.599</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>4,495.983</td>
<td>109.073</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.076</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>HG00096</td>
<td>base</td>
<td>1,030.635</td>
<td>0.771</td>
<td>0.075</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>1.758</td>
<td>0.043</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>2,614.746</td>
<td>274.983</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>1,782.782</td>
<td>187.489</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.003</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>107.552</td>
<td>249.396</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.010</td>
<td>0.024</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>11,666.458</td>
<td>59.087</td>
<td></td>
</tr>
</tbody>
</table>
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>In-memory size in MiB</th>
<th>Compression Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>64.599</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>109.073</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.002</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>base</td>
<td>1,030.635</td>
<td>0.075</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>0.043</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>274.983</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>187.489</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.001</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>249.396</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.024</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>59.087</td>
</tr>
</tbody>
</table>
Main-Memory DBMSs for OLAP

Main memory as primary storage ✓
✓ Column-oriented data layout ✓
✓ Lightweight compression ✓
 ○ Optimized query processing

Main-Memory DBMS for efficient OLAP
Query Processing - Motivation

• For analytical applications (OLAP), most of the time a small set of columns is needed to answer a query
  → Read only columns that are needed to answer a query
  → Column-oriented data layout is highly suitable

• Beware for queries accessing all columns of a table
  → high materialization costs
Materialization Strategies

- Recall: In a column-oriented data layout each column is stored separately
- Consider, e.g., a selection query:

```
SELECT l_shipdate, l_linenumber
FROM lineitem
WHERE l_shipdate < "2009-09-26"
AND l_linenumber < "10000"
```

→ Query accesses and returns values of two columns
→ Materialization (tuple reconstruction) during query processing necessary
Early Materialization
Reconstruct tuples as soon as possible

Picture taken from [Abadi et al., 2007]

SPC ... Scan, Predicate, Construct
Late Materialization
Postpone tuple reconstruction to the latest possible time

Picture taken from [Abadi et al., 2007]
Advantages of Early and Late Materialization

Early Materialization (EM):
- Reduces access cost if one column has to be accessed multiple times during query processing
- ↗ [Abadi et al., 2007]

Late Materialization (LM):
- Reduces amount of tuples to reconstruct
- LM allows processing of columns as long as possible
  → Processing of compressed data
  → LM improves cache effectiveness
- ↗ [Abadi et al., 2008]
Example: Invisible Join [Abadi et al., 2008]

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
    AND lo.suppkey = s.suppkey
    AND lo.orderdate = d.datekey
    AND c.region = ASIA
    AND s.region = ASIA
    AND d.year >= 1992
    AND d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```
Example: Invisible Join  [Abadi et al., 2008]

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
    supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.region = ASIA
  AND s.region = ASIA
  AND d.year >= 1992
  AND d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```
Example: Invisible Join - Hash  [Abadi et al., 2008]

**Apply** region = 'Asia' on Customer table

<table>
<thead>
<tr>
<th>custkey</th>
<th>region</th>
<th>nation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Asia</td>
<td>China</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Europe</td>
<td>France</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Asia</td>
<td>India</td>
<td></td>
</tr>
</tbody>
</table>

Hash table with keys 1 and 3

**Apply** region = 'Asia' on Supplier table

<table>
<thead>
<tr>
<th>suppkey</th>
<th>region</th>
<th>nation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Asia</td>
<td>Russia</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Europe</td>
<td>Spain</td>
<td></td>
</tr>
</tbody>
</table>

Hash table with key 1

**Apply** year in [1992, 1997] on Date table

<table>
<thead>
<tr>
<th>dateid</th>
<th>year</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01011997</td>
<td>1997</td>
<td></td>
</tr>
<tr>
<td>01021997</td>
<td>1997</td>
<td></td>
</tr>
<tr>
<td>01031997</td>
<td>1997</td>
<td></td>
</tr>
</tbody>
</table>

Hash table with keys 01011997, 01021997, and 01031997
Example: Invisible Join  [Abadi et al., 2008]

```
SELECT c.nation, s.nation, d.year, 
    sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo, 
    supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey 
    AND lo.suppkey = s.suppkey 
    AND lo.orderdate = d.datekey 
    AND c.region = ASIA 
    AND s.region = ASIA 
    AND d.year >= 1992 
    AND d.year <= 1997 
GROUP BY c.nation, s.nation, d.year 
ORDER BY d.year asc, revenue desc;
```
Example: Invisible Join - Probe [Abadi et al., 2008]

Fact Table

<table>
<thead>
<tr>
<th>orderkey</th>
<th>custkey</th>
<th>suppkey</th>
<th>orderdate</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>01011997</td>
<td>43256</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>01011997</td>
<td>33333</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>01021997</td>
<td>12121</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>01021997</td>
<td>23233</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>01021997</td>
<td>45456</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>01031997</td>
<td>43251</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>01031997</td>
<td>34235</td>
</tr>
</tbody>
</table>

Hash table with keys 1 and 3

Hash table with key 1

Hash table with keys 01011997, 01021997, and 01031997

Bitwise And

matching fact table bitmap for cust. dim. join

fact table tuples that satisfy all join predicates
Example: Invisible Join  [Abadi et al., 2008]

```
SELECT c.nation, s.nation, d.year, 
       sum(lo.revenue) as revenue 
FROM customer AS c, lineorder AS lo, 
     supplier AS s, dwdate AS d 
WHERE lo.custkey = c.custkey 
  AND lo.suppkey = s.suppkey 
  AND lo.orderdate = d.datekey 
  AND c.region = ASIA 
  AND s.region = ASIA 
  AND d.year >= 1992 
  AND d.year <= 1997 
GROUP BY c.nation, s.nation, d.year 
ORDER BY d.year asc, revenue desc;
```
Example: Invisible Join - Materialize [Abadi et al., 2008]
Star-Schema-Benchmark Performance

Figure taken from [Abadi et al., 2008]
Star-Schema-Benchmark Performance (2)

Impact of column-store optimizations (ranked according to previous experiment)

1. Late materialization
   → process only needed data

2. Compression
   → better memory bandwidth utilization

3. Bulk processing
   → improves instruction-cache effectiveness

4. Invisible Join
   → optimized join processing that makes use of late materialization
Main-Memory DBMSs for OLAP

Main memory as primary storage ✔
+ Column-oriented data layout ✔
+ Lightweight compression ✔
+ Optimized query processing ✔

Main-Memory DBMS for efficient OLAP
Main-Memory DBMSs for OLTP
The following slides are based on the lecture *Transaction Management in Main-Memory DBMSs* by Sebastian Breß.
Online Transaction Processing (OLTP)
Online Transaction Processing (OLTP)

mixed workloads, read and write
Online Transaction Processing (OLTP)

query mostly over all columns but only a few rows
Online Transaction Processing (OLTP)

query **throughput** is important
Main-Memory DBMSs for OLTP

- mixed workloads, **read and write**
  → Data layout needed that allows for fast updates and inserts
Main-Memory DBMSs for OLTP

- mixed workloads, read and write
  → Data layout needed that allows for fast updates and inserts
- query mostly over all columns but only a few rows
  → Use row-oriented data layout
  → Yields better data cache utilization
Main-Memory DBMSs for OLTP

- mixed workloads, read and write
  → Data layout needed that allows for fast updates and inserts

- query mostly over all columns but only a few rows
  → Use row-oriented data layout
  → Yields better data cache utilization

- query throughput is important
  → Fully exploit modern hardware by minimizing functional overhead introduced by traditional, disk-based DBMSs
  - Single-threaded vs. multi-threaded execution
  - Logging vs. high availability
Single-Threaded vs. Multi-Threaded Execution

- System R like systems use multi-threading to hide disk latencies and to increase throughput

- Required functionality
  1. Ensure serializability of transactions
     → Use locking
  2. Protect data structures for concurrent access
     → Use latching

- Additional mechanisms introduce significant overhead
Single-Threaded vs. Multi-Threaded Execution (2)

• A single transaction’s execution time is roughly in the order of hundreds of microseconds

• If TXN are executed serially, we:
  1. Can remove locking, because serializability is guaranteed
  2. Can remove latching, because data structures are not accessed concurrently
→ Execute one transaction at a time is often faster!
Logging vs. High Availability

- Writing a log file introduces significant overhead

- Failover/rebuild is as efficient as using a redo log for recovery
  [Lau and Madden, 2006]
  1. Undo only required for running TXNs, undo log can be held in main-memory structure
  2. No need for redo, because of network recovery from remote site
     → If node restarts after system failure, the database is updated with data from the operational node

→ No persistent log file needed!
Overhead Breakdown of RDBMS Shore

- **buffer manager**: 34.6%
- **locking**: 16.3%
- **latching**: 14.2%
- **logging**: 11.9%
- **hand-coded optimizations**: 16.2%

Picture taken from [Harizopoulos et al., 2008]
Overhead Breakdown of RDBMS Shore: Payment TXN of TPC-C Benchmark

Picture taken from [Harizopoulos et al., 2008]
Alternative Design Considerations – Summary

• System R like systems introduce a lot of overhead if data can be kept in memory [Harizopoulos et al., 2008]:
  1. Logging
  2. Locking
  3. Latching
  4. Buffer management

• A redesigned engine such as H-Store can outperform traditional, system R like RDBMSs
  → Stonebraker and others observed a performance improvement by a factor of 82 [Stonebraker et al., 2007]
H-Store

• Next-generation OLTP system

• Operates on a distributed cluster of shared-nothing machines

• Data resides entirely in main memory

• Row-oriented storage
H-Store

- H-Store site: basic operational entity, single-threaded execution of TXNs, assigned to one core
  - Sites independent from each other
  - No sharing of data structures or memory with co-located sites running on the same machine

- H-Store node: physical machine, hosts multiple sites
  \( \approx \# \text{CPU cores} \)

- H-Store instance: cluster of two or more nodes
H-Store: Architecture

Deployment Framework
- Database Schema
- Cluster Information
- Stored Procedures
- Sample Workload

Transaction Initiator
- OLTP Application
- H-Store API

Messaging Fabric
- Transaction Manager
  - Stored Procedure Executor
  - Query Execution Engine
  - System Catalogs

Compiled Stored Procedures
- Query Plans
- Physical Layout

Main Memory Storage Manager
- Other Cluster Execution Nodes

Deployment Time

Runtime Time

Picture taken from [Kallman et al., 2008]
Main-Memory DBMSs for OLTP and OLAP
Main-Memory DBMSs for OLTP and OLAP

Combining OLTP and OLAP in one system leads to conflicting requirements!
Main-Memory DBMSs for OLTP and OLAP

read-mostly $\not\leftarrow$ read AND write
Main-Memory DBMSs for OLTP and OLAP

query over

FEW columns but ALL rows \lor ALL columns but FEW rows
Main-Memory DBMSs for OLTP and OLAP

query over
query response time ≈ query throughput
Design Considerations

• Data layout:
  → Which data layout to use for storing data?

• Data processing:
  → How to process analytical and transactional workloads simultaneously?
Data Layout

Different data stores are advantageous for different database application

- Column stores are advantageous for OLAP workloads
  → Aggregation over complete columns, just read needed data
- Row stores are advantageous for OLTP workloads
  → Update, write, record access
Data Layout

Different data stores are advantageous for different database application

- Column stores are advantageous for OLAP workloads
  → Aggregation over complete columns, just read needed data
- Row stores are advantageous for OLTP workloads
  → Update, write, record access

→ Single data layouts seem not capable for OLTP and OLAP processing at the same time
→ Use a hybrid data layout/ storage engine:

1. Storage Advisor [Rösch et al., 2012]
2. Delta Store [Sikka et al., 2012]
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

- Idea: Cost-based recommendation of data layout based on
  - Workload (online and offline)
  - Data schema
  - Data characteristics

- Goal: Minimize runtime of given (offline) workload
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

- Idea: Cost-based recommendation of data layout based on
  - Workload (online and offline)
  - Data schema
  - Data characteristics
- Goal: Minimize runtime of given (offline) workload
Storage Advisor for Hybrid-Store Databases  [Rösch et al., 2012]

Recommendations are made for complete tables
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

Recommendations are made for complete tables **but also for**

- **rows**
- **columns**

Pictures taken from [Rösch et al., 2012]
**Delta Store** [Sikka et al., 2012]

- Idea: Store records according to their life cycle stage:
  - New records → row store
  - Old records → column store
- Concrete storage layout is transparent to database operations
Delta Store [Sikka et al., 2012]

- Idea: Store records according to their life cycle stage:
  - New records → row store
  - Old records → column store
- Concrete storage layout is transparent to database operations
Data Processing

- OLAP and OLTP applications have different optimization goals → Response time vs. throughput
- Problems when processing both workloads at the same time:
  - OLAP queries are postponed until OLTP queries
    → \$ \downarrow \text{Response time}
  - OLTP queries have to wait for long lasting OLAP queries
    → \$ \downarrow \text{query throughput}

→ Decouple workloads in one DBMS!
HyPer [Kemper and Neumann, 2011]

• Supports efficient hybrid OLTP/OLAP processing

• Keeps all data in main memory

• OLTP queries run in one process, OLAP queries run on a virtual memory snapshot from the OLTP process

• Create snapshot using the *fork* system call of the Linux kernel

• Assumption: We do not need the latest consistent state for OLAP, because analysis result is hardly influenced by a couple of updates
HyPer: Copy on Write Mechanism

When a OLTP query attempts to modify a page:

- The linux kernel creates a new page containing the old data for the OLAP process → transparently to the DBMS!
- Then, the OLTP process can modify the page → OLAP process always has a transaction consistent snapshot
HyPer: Database Architecture

HyPer is a hybrid OLTP&OLAP main memory Database System as efficient as dedicated OLTP main memory DBMS (e.g., VoltDB, TimesTen) and as fast as dedicated OLAP main memory DBMS (e.g., MonetDB, TREX). Picture taken from Kemper and Neumann, 2011.
HyPer: Concept for Hybrid OLTP/OLAP

Picture taken from [Kemper and Neumann, 2011]
HyPer: Concept for Hybrid OLTP/OLAP

Picture taken from [Kemper and Neumann, 2011]
HyPer: Persistency and Logging

Picture taken from [Kemper and Neumann, 2011]
HyPer: Multi Node Support

Picture taken from [Kemper and Neumann, 2011]
Summary

- Rethinking the architecture of DBMSs to adapt them on changes in hardware pays off [Harizopoulos et al., 2008]

- Modern transactional systems keep their operational data in main memory
  → We have to optimize for different things than 30 years ago

→ New architectures mean that we have to solve old problems again, but for the current hardware landscape and the applications of today and tomorrow!
CoGaDB

CoGaDB (Column Oriented GPU accelerated DBMS):

• Main-memory DBMS
• Column store
• Lightweight compression techniques such as Dictionary Encoding, RLE, Bit-Vector Encoding
• Late materialization → optimized query processing (e.g., Invisible Join)
• Co-processor acceleration
• (Hybrid) query optimization
Invitation

Your are invited to join our research on main-memory databases and databases on new hardware, e.g., in form of:

• Bachelor or master thesis

• “Scientific Project: Data Management on new Hardware”

• Scientific individual project

Contact: david.broneske@iti.cs.uni-magdeburg.de
References I

Column-stores vs. row-stores: how different are they really?
In *SIGMOD*, pages 967–980.

Materialization Strategies in a Column-Oriented DBMS.
In *ICDE*, pages 466–475.

A decomposition storage model.
In *SIGMOD*, pages 268–279.

What Every Programmer Should Know About Memory.

Encapsulation of Parallelism in the Volcano Query Processing System.
In *SIGMOD*, pages 102–111.

OLTP through the looking glass, and what we found there.
In *SIGMOD*, pages 981–992.
References II

Morgan Kaufmann, 2 edition.

Morgan Kaufmann, 4 edition.

Morgan Kaufmann, 5 edition.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., 
PVLDB, 1(2):1496–1499.

HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. 
In ICDE, pages 195–206.

An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. 
In VLDB, pages 703–714.

Database Architecture Evolution: Mammals flourished long before Dinosaurs became extinct.

DB2 with BLU Acceleration: So Much More Than Just a Column Store.

How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations.
In *VLDB*, pages 858–869.

Constant-Time Query Processing.
In *ICDE*, pages 60–69.

A Storage Advisor for Hybrid-Store Databases.
*PVLDB*, 5(12):1748–1758.
References IV

