Main-Memory Database Management Systems

David Broneske
Credits

Part of this lecture are based on content by

- Jens Teubner from TU Dortmund
- Sebastian Breß from TU Berlin
- Sebastian Dorok
We will talk about

- Computer and Database Systems Architecture
- Cache Awareness
- Processing Models
- Storage Models
Computer and Database Systems
Architecture
The Past and the Present
The Past

- **Latency**
  - 5 ns: 200 B
  - 10 ns: 64 KB
  - 100 ns: 32 MB
  - 5,000,000 ns: 2 GB

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

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

- Main-memory capacity is limited to several megabytes
  → Only a small fraction of the database fits in main memory
- And disk storage is "huge",
  → Traditional database systems use disk as primary storage
The Past - Database Systems

- Main-memory capacity is limited to several megabytes
  → Only a small fraction of the database fits in main memory
- And disk storage is "huge",
  → Traditional database systems use disk as primary storage
- But disk latency is high
  → Parallel query processing to hide disk latencies
  → Choose proper buffer replacement strategy to reduce I/O
  → Architectural properties inherited from system R, the first "real" relational DBMS
  → From the 1970’s...
System-R-like Architecture
Overhead Breakdown of RDBMS Shore

![Diagram showing overhead breakdown with bars representing different components:]

- **Buffer Manager**: 34.6%
- **Latching**: 14.2%
- **Locking**: 16.3%
- **Logging**: 11.9%
- **Hand-coded Optimizations**: 16.2%

The diagram is taken from Harizopoulos et al., 2008.
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, 1996]
The Present - Database Systems

- Server machines have up to thousands of gigabyte of main memory available
  → Use main memory as primary storage for the database and remove disk access as main performance bottleneck
The Present - Database Systems

- Server machines have up to thousands of gigabyte of main memory available
  → 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 have antiquated the disk-oriented relational architecture for OLTP applications.” [Stonebraker et al., 2007]
Disk-based vs. Main-Memory DBMS
Disk-based vs. Main-Memory DBMS

- Disk-based DBMS
  - CPU
  - Main Memory
    - Buffered Data
  - Disk
    - Data

- Main-Memory DBMS
  - CPU
  - Main Memory
    - Data
  - Replicated Data

Having the database in main memory allows us to remove buffer manager and paging:
- Remove level of indirection
- Results in better performance
Overhead Breakdown of RDBMS Shore: Payment TXN of TPC-C Benchmark

Picture taken from [Harizopoulos et al., 2008]
Disk-based vs. Main-Memory DBMS

Disk bottleneck is removed as database is kept in main memory
→ Access to main memory becomes new bottleneck
The New Bottleneck: Memory Access

![Graph showing normalized performance over years for Processor and DRAM Memory.]
The New Bottleneck: Memory Access

There is an increasing gap between CPU and memory speeds.

● Also called the memory wall.

● CPUs spend much of their time waiting for memory.

How can we break the memory wall and better utilize the CPU?
The New Bottleneck: Memory Access

→ Caches resemble the buffer manager but are **controlled by hardware**

→ Be aware of the caches!
Cache Awareness
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)
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... locality.
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.*

*Modern CPUs support out-of-order execution and several in-flight cache misses
Example: AMD Opteron

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 (∼ 1 ns)
- L2 hit latency: 7 cycles (∼ 3.5 ns)
- L2 miss latency: 160–180 cycles (∼ 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
  → 4MB cache, line size of 64B:
    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**.
Block Placement: Set-Associative Cache

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

- Direct-mapped
- 2-way associative
- 4-way associative
- 8-way associative

The graph shows the number of cache misses (in millions) for different cache sizes and cache parameter configurations. As the cache size increases, the number of cache misses decreases significantly, indicating improved cache performance.
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**.*
  - Greatly simplifies **data coherency**.

**Write Back**
- Data is only written into the cache.
- A **dirty** flag marks modified cache lines (Uses a status field.)
  - May reduce traffic to lower-level memory.
  - Need to write on eviction of dirty cache lines.

Modern processors usually implement **write back**.

*Write buffers can be used to overcome this problem.

David Broneske | Main-Memory Database Management Systems
Putting It All Together

To compensate for slow memory, systems use caches.

- 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.

*Write buffers can be used to overcome this problem.*
Performance (SPECint 2000)

- L1 Instruction Cache
- L2 Cache (shared)
Performance (SPECint 2000)

![Graph showing instruction cache misses per 1000 instructions for various benchmark programs. The x-axis represents the benchmark programs, and the y-axis represents the number of misses. The graph compares L1 Instruction Cache and L2 Cache (shared).]
Why do DBSs show such poor cache behavior?

Poor code locality:
- Polymorphic functions
  (E.g., resolve attribute types for each processed tuple individually.)

- Set associativity: A memory block can only go into a small number of cache lines (most caches are set-associative).
Why do DBSs show such poor cache behavior?

**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?
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**.
- Each operator keeps its own state.
Tuple-At-A-Time Processing

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

- 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

- **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

- **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.
Example: TPC-H Query Q1 on MySQL

```sql
SELECT l_returnflag, l_linenumber, SUM(l_quantity) AS sum_qty,
       SUM(l_extendedprice) AS sum_base_price,
       SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
       SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge,
       AVG(l_quantity) AS avg_qty, AVG(l_extendedprice) AS avg_price,
       AVG(l_discount) AS avg_disc, COUNT(*) AS count_order
FROM lineitem
WHERE l_shipdate <= DATE '1998-09-02'
GROUP BY l_returnflag, l_linenumber
```

- **Scan query** with **arithmetics** and a bit of aggregation.

Source: MonetDB/X100: Hyper-Pipelining Query Execution. [Boncz et al., 2005]
<table>
<thead>
<tr>
<th>time [sec]</th>
<th>calls</th>
<th>instr./call</th>
<th>IPC</th>
<th>function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.9</td>
<td>846M</td>
<td>6</td>
<td>0.64</td>
<td>ut_fold_ulong_pair</td>
</tr>
<tr>
<td>8.5</td>
<td>0.15M</td>
<td>27K</td>
<td>0.71</td>
<td>ut_fold_binary</td>
</tr>
<tr>
<td>5.8</td>
<td>77M</td>
<td>37</td>
<td>0.85</td>
<td>memcpy</td>
</tr>
<tr>
<td><strong>3.1</strong></td>
<td><strong>23M</strong></td>
<td><strong>64</strong></td>
<td><strong>0.88</strong></td>
<td>Item_sum_sum::update_field</td>
</tr>
<tr>
<td>3.0</td>
<td>6M</td>
<td>247</td>
<td>0.83</td>
<td>row_search_for_mysql</td>
</tr>
<tr>
<td><strong>2.9</strong></td>
<td><strong>17M</strong></td>
<td><strong>79</strong></td>
<td><strong>0.70</strong></td>
<td>Item_sum_avg::update_field</td>
</tr>
<tr>
<td>2.6</td>
<td>108M</td>
<td>11</td>
<td>0.60</td>
<td>rec_get_bit_field_1</td>
</tr>
<tr>
<td>2.5</td>
<td>6M</td>
<td>213</td>
<td>0.61</td>
<td>row_sel_store_mysql_rec</td>
</tr>
<tr>
<td>2.4</td>
<td>48M</td>
<td>25</td>
<td>0.52</td>
<td>rec_get_nth_field</td>
</tr>
<tr>
<td>2.4</td>
<td>60</td>
<td>19M</td>
<td>0.69</td>
<td>ha_print_info</td>
</tr>
<tr>
<td>2.4</td>
<td>5.9M</td>
<td>195</td>
<td>1.08</td>
<td>end_update</td>
</tr>
<tr>
<td>2.1</td>
<td>11M</td>
<td>89</td>
<td>0.98</td>
<td>field_conv</td>
</tr>
<tr>
<td>2.0</td>
<td>5.9M</td>
<td>16</td>
<td>0.77</td>
<td>Field_float::val_real</td>
</tr>
<tr>
<td>1.8</td>
<td>5.9M</td>
<td>14</td>
<td>1.07</td>
<td>Item_field::val</td>
</tr>
<tr>
<td>1.5</td>
<td>42M</td>
<td>17</td>
<td>0.51</td>
<td>row_sel_field_store_in_mysql</td>
</tr>
<tr>
<td>1.4</td>
<td>36M</td>
<td>18</td>
<td>0.76</td>
<td>buf_frame_align</td>
</tr>
<tr>
<td><strong>1.3</strong></td>
<td><strong>17M</strong></td>
<td><strong>38</strong></td>
<td><strong>0.80</strong></td>
<td>Item_func_mul::val</td>
</tr>
<tr>
<td>1.4</td>
<td>25M</td>
<td>25</td>
<td>0.62</td>
<td>pthread_mutex_unlock</td>
</tr>
<tr>
<td>1.2</td>
<td>206M</td>
<td>2</td>
<td>0.75</td>
<td>hash_get_nth_cell</td>
</tr>
<tr>
<td>1.2</td>
<td>25M</td>
<td>21</td>
<td>0.65</td>
<td>mutex_test_and_set</td>
</tr>
<tr>
<td>1.0</td>
<td>102M</td>
<td>4</td>
<td>0.62</td>
<td>rec_get_1byte_offs_flag</td>
</tr>
<tr>
<td>1.0</td>
<td>53M</td>
<td>9</td>
<td>0.58</td>
<td>rec_1_get_field_start_offs</td>
</tr>
<tr>
<td>0.9</td>
<td>42M</td>
<td>11</td>
<td>0.65</td>
<td>rec_get_nth_field_extern_bit</td>
</tr>
<tr>
<td><strong>1.0</strong></td>
<td><strong>11M</strong></td>
<td><strong>38</strong></td>
<td><strong>0.80</strong></td>
<td>Item_func_minus::val</td>
</tr>
<tr>
<td><strong>0.5</strong></td>
<td><strong>5.9M</strong></td>
<td><strong>38</strong></td>
<td><strong>0.80</strong></td>
<td>Item_func_plus::val</td>
</tr>
</tbody>
</table>
Observations

- Only **single tuple** processed in each call; **millions of calls**.
- Only 10% of the time spent on actual query task.
- Low instructions-per-cycle (IPC) ratio.

- Much time spent on field access (e.g., rec get nth field()).
  - Polymorphic operators
- Single-tuple functions hard to optimize (by compiler).
  → Low instructions-per-cycle ratio.
  → Vector instructions (SIMD) hardly applicable.
- Function call overhead (e.g., Item func plus::val()).
  \[
  \frac{38 \text{ instr.}}{0.8 \text{ instr. per cycle}} = 48 \text{ cycles vs. } 3 \text{ instr. for load/add/store assembly}\]

* Depends on underlying hardware
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.
Operator-At-A-Time Processing

Function call overhead is now replaced by extremely tight loops.

Example: `batval_int_add(···)`

```c
:
  if (vv != int_nil) {
    for (; bp < bq; bp++, bnp++) {
      REGISTER int bv = *bp;
      if (bv != int_nil) {
        bv = (int) OP(bv,+,vv);
      }
      *bnp = bv;
    }
  } else {
    ...
  }
  ...
```
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
    $\rightarrow$ loop unrolling
    $\rightarrow$ vectorization (use of SIMD instructions)
  - can leverage modern CPU features (hardware prefetching).
- Function calls are now out of the critical code path.
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**.
- **No** per-tuple field extraction or type resolution.
  - **Operator specialization**, e.g., for every possible type.
  - Implemented using **macro expansion**.
  - Possible due to column-based storage.
<table>
<thead>
<tr>
<th>result size</th>
<th>time [ms]</th>
<th>bandwidth [MB/s]</th>
<th>MIL statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9M</td>
<td>127</td>
<td>352</td>
<td>s0 := select (l_shipdate, ...) . mark ();</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>505</td>
<td>s1 := join (s0, l_returnag);</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>506</td>
<td>s2 := join (s0, l_linestatus);</td>
</tr>
<tr>
<td>5.9M</td>
<td>235</td>
<td>483</td>
<td>s3 := join (s0, l_extprice);</td>
</tr>
<tr>
<td>5.9M</td>
<td>233</td>
<td>488</td>
<td>s4 := join (s0, l_discount);</td>
</tr>
<tr>
<td>5.9M</td>
<td>232</td>
<td>489</td>
<td>s5 := join (s0, l_tax);</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>507</td>
<td>s6 := join (s0, l_quantity);</td>
</tr>
<tr>
<td>5.9M</td>
<td>290</td>
<td>155</td>
<td>s7 := group (s1);</td>
</tr>
<tr>
<td>5.9M</td>
<td>329</td>
<td>136</td>
<td>s8 := group (s7, s2);</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>s9 := unique (s8 . mirror ());</td>
</tr>
<tr>
<td>5.9M</td>
<td>206</td>
<td>440</td>
<td>r0 := [+] (1.0, s5);</td>
</tr>
<tr>
<td>5.9M</td>
<td>210</td>
<td>432</td>
<td>r1 := [-] (1.0, s4);</td>
</tr>
<tr>
<td>5.9M</td>
<td>274</td>
<td>498</td>
<td>r2 := [*] (s3, r1);</td>
</tr>
<tr>
<td>5.9M</td>
<td>274</td>
<td>499</td>
<td>r3 := [*] (s12, r0);</td>
</tr>
<tr>
<td>4</td>
<td>165</td>
<td>271</td>
<td>r4 := {sum} (r3, s8, s9);</td>
</tr>
<tr>
<td>4</td>
<td>165</td>
<td>271</td>
<td>r5 := {sum} (r2, s8, s9);</td>
</tr>
<tr>
<td>4</td>
<td>163</td>
<td>275</td>
<td>r6 := {sum} (s3, s8, s9);</td>
</tr>
<tr>
<td>4</td>
<td>163</td>
<td>275</td>
<td>r7 := {sum} (s4, s8, s9);</td>
</tr>
<tr>
<td>4</td>
<td>144</td>
<td>151</td>
<td>r8 := {sum} (s6, s8, s9);</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>196</td>
<td>r9 := {count} (s7, s8, s9);</td>
</tr>
</tbody>
</table>

Source: MonetDB/X100: Hyper-Pipelining Query Execution. [Boncz et al., 2005]
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 `next()` call return a large number of tuples → a so called “vector”

Choose vector size
- **large enough** to compensate for iteration overhead (function calls, instruction cache misses, . . .), but
- **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>instr. cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>function calls</td>
<td>many complex</td>
<td>extremely few</td>
<td>very few direct</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*?
Vectorized Execution Model

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

- Row Stores

- Column Stores
Row Stores

a.k.a. row-wise storage or n-ary storage model, NSM:
Column Stores

a.k.a. column-wise storage or decomposition storage model, DSM:
The Effect on Query Processing

Consider, e.g., a selection query:

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

This query typically involves a **full table scan**.
A Full Table Scan in a Row Store

In a row-store, all **rows** of a table are stored sequentially on a database page.
A Full Table Scan in a Row Store

In a row-store, all **rows** of a table are stored sequentially on a database page.
A Full Table Scan in a Row Store

In a row-store, all rows of a table are 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.
A “Full Table Scan” in a Column Store

In a column store, all values of one column are stored sequentially on a database page.

```
l_shipdate(s)
```
A “Full Table Scan” in a Column Store

In a column store, all values of one **column** are stored sequentially on a database page.

```
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
    | l_shipdate(s) |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
    | cache block boundaries |
+---+---+---+---+---+---+---+---+
```

All data loaded into caches by a “l_shipdate scan” is now actually **relevant** for the query.
Column Store Advantages

- 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 \( l_{\text{shipdate}} \) 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.
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
- 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

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

<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>

Dictionary Encoding and RLE
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)

### Example

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

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

### Dictionary

<table>
<thead>
<tr>
<th>Country</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>01</td>
</tr>
<tr>
<td>Germany</td>
<td>001</td>
</tr>
<tr>
<td>Russia</td>
<td>0001</td>
</tr>
<tr>
<td>China</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
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

Picture taken from [Raman et al., 2008]
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
Columns Store Trade Offs

**Tuple recombination** can cause considerable cost.
- Need to perform **many joins**.
- Workload-dependent trade-off.

*Figure 2: Varying the Number of Projected Attributes*

Source: [Copeland and Khoshafian, 1985]
An example: Binary Association Tables in MonetDB

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).
- Often, oids can be implemented as virtual oids (voids).

→ Not explicitly materialized in memory.
Materialization Strategies

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

```sql
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]
Conclusion

- **Row stores** store complete tuples sequentially on a database page
- **Column stores** store all values of one column sequentially on a database page
- Depending on the workload column stores or row stores are more advantageous
  - Tuple reconstruction is overhead in column stores
  - Analytical workloads that process few columns at a time benefit from column stores
→ One data storage approach is not optimal to serve all possible workloads
Examples of systems

HDD

Data Management System for Relational Data
- Vertica
- hadapt
- Neo4j
- TITAN
- Google Spanner
- Asterix
- InfiniteGraph
- MySQL
- Cassandra
- Oracle

Data Management System for Graph Data
- Apache Hama

Data Management System for Streams

NVM

Data Management System for Relational Data
- FAWN
- SILT
- SkimpyStash
- Clustrix

Data Management System for Graph Data
- GraphChi
- GraphLab

Data Management System for Streams

DRAM

Data Management System for Relational Data
- H-Store
- ORACLE
- GraphX

Data Management System for Graph Data
- GPS
- Trinity
- GraphX

Data Management System for Streams
- GridGain
- S4
- Storm
- Spark

Generic Data Processing Engine
- Dryad
- epic

HDD-based Big Data Storage System
- LogBase
- hadoop

Memory-based Big Data Storage System
- Aerospike
- MemC3
- Memepic

Generic Data Processing Engine
- Piccolo
- Mammoth

Memory-based Big Data Storage System
- Redis
- FaRM

Memory-based Big Data Storage System
- Memepic
Take home messages

- **Modern hardware** offers performance improvements for DB applications → disk vs. main-memory access speed

- **Rethinking the architecture of DBMSs** to adapt them on changes in hardware pays off → single- vs. multi-threaded OLTP engines

- **New DBMS architectures** mean that we have to
  - solve old problems again → durability and availability
  - optimize for different things → cache effectiveness
References I


In ICDE, pages 466–475.


In SIGMOD, pages 268–279.

What Every Programmer Should Know About Memory.

References II

In SIGMOD, pages 981–992.

Morgan Kaufmann, 4 edition.

Morgan Kaufmann, 5 edition.

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

DB2 with BLU Acceleration: So Much More Than Just a Column Store.
PVLDB, 6(11):1080–1091.
References III

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

In ICDE, pages 60–69.

In VLDB, pages 1150–1160.