Data Processing on Modern Hardware

Jana Giceva

Lecture 2: Cache awareness
Cache Awareness
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

![Graph showing the increasing gap between CPU and memory speeds between 1980 and 2020.](image)
Memory ≠ Memory

Random Access Memory (RAM)

Dynamic RAM (DRAM)
- State kept in capacitor
- Leakage → refreshing needed
- Small capacitor, 1 transistor – high density
- Usage: DIMM (DRAM)

Static RAM (SRAM)
- Bistable latch (0 or 1)
- Cell state stable → no refreshing needed
- 6 transistors – low density, high power
- Usage: CPU-caches
Dynamic RAM is comparably **slow:**
- Memory needs to be **refreshed** periodically (every 64 ms)
- (Dis-)charging a capacitor takes time
- ~ 200 CPU cycles per access

Under certain circumstances, DRAM **can** be reasonably fast:
- DRAM cells are physically organized as a 2-d array.
- The discharge/amplify process done for an **entire row** and more than one word can be read out.
- Several DRAM cells can be used in **parallel.**

We can exploit that by using **sequential access patterns.**
SRAM, in contrast, can be very fast.
- Transistors actively drive output lines, so access to memory is almost instantaneous.

But, SRAM is significantly more expensive (chip space = money).

Therefore, organize memory as a hierarchy and use small, fast memories as caches for slow memory.
Intuition: Cache resemble the buffer manager but are controlled by hardware.
Caches take advantage of the *principle of locality*

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

**Spatial locality:**
- Code often contains loops
- Related data is often spatially close

**Temporal locality:**
- Code may call a function repeatedly, even if it is not spatially close
- Programs tend to reuse data frequently.
Example locality: Data? Instructions?

Temporal locality:
- Data: \( \text{sum} \) referenced in each iteration
- Instructions: cycle through loop repeatedly

Spatial locality:
- Data: array \( a[] \) accessed in stride-1 pattern
- Instructions: reference instructions in sequence

```c
sum = 0;
for (i = 0; i < n; i++)
{
    sum += a[i];
}
return sum;
```
How we access data stored in memory can have significant impact on performance.

```c
int sum_array_col(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            sum += a[i][j];
        }
    }
    return sum;
}

int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```
### Locality example #1

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

### Layout in Memory

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
</tr>
<tr>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td></td>
</tr>
<tr>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Access Pattern:**
1) `a[0][0]`
2) `a[1][0]`
3) `a[2][0]`
4) `a[0][1]`
5) `a[1][1]`
6) `a[2][1]`
7) `a[0][2]`
8) `a[1][2]`
9) `a[2][2]`
10) `a[0][3]`
11) `a[1][3]`
12) `a[2][3]`

**Note:** 76 is just one possible starting address of array `a`.

**M = 3, N = 4**

```c
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
</tbody>
</table>
```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}

M = 3, N = 4

Access Pattern: 1) \( a[0][0] \) 2) \( a[0][1] \) 3) \( a[0][2] \) 4) \( a[0][3] \) 5) \( a[1][0] \) 6) \( a[1][1] \) 7) \( a[1][2] \) 8) \( a[1][3] \) 9) \( a[2][0] \) 10) \( a[2][1] \) 11) \( a[2][2] \) 12) \( a[2][3] \)

Layout in Memory

<table>
<thead>
<tr>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[0][3]</th>
<th>a[1][0]</th>
<th>a[1][1]</th>
<th>a[1][2]</th>
<th>a[1][3]</th>
<th>a[2][0]</th>
<th>a[2][1]</th>
<th>a[2][2]</th>
<th>a[2][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>92</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: 76 is just one possible starting address of array a
Executing the program for a 20'000 x 20'000 matrix gives an order of magnitude difference:

- **16.98 sec** for `sum_array_col` vs **1.71 sec** for `sum_array_row`

Quick check with `perf` measuring the cpu cycles and cache misses confirms the importance of writing *cache-friendly* code.
Cache Internals – recap
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* is 64 bytes.

- The organization in cache lines in consistent with the principle of (spatial) locality.
- Block-wise transfers are well-supported by DRAM chips.
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 cache line 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
Big difference between the cost of cache hit and a cache miss

- Could be 100x speed difference between accessing cache and main memory (in clock cycles)

**Miss rate (MR)**

- Fraction of memory references not found in cache: \( \frac{\text{misses}}{\text{accesses}} = 1 - \text{Hit rate} \)

**Hit time (HT)**

- Time to deliver a cache line from the cache to the processor

**Miss penalty (MP)**

- Additional time required because of a miss

Average time to access memory (considering both hits and misses): \( HT + MR \times MP \)
Cache performance

Big *difference* between the *cost* of *cache hit* and a *cache miss*
- Could be 100x speed difference between accessing cache and main memory (in clock cycles)

- Average time to access memory (considering both hits and misses): $HT + MR \times MP$

- 99% hit rate is twice as good as 97% hit rate
  - Assume HT of 1 *cycle*, and MP of 100 *clock cycles*
  - 97%: $1 + (1 - 0.97) \times 100 = 1 + 3 = 4$ *cycles*
  - 99%: $1 + (1 - 0.99) \times 100 = 1 + 1 = 2$ *cycles*
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:
  - For 4MB cache, line size of 64B
    → 65,536 cache lines
- Used, e.g., for small translation lookaside buffer (TLB) caches.
In a **direct mapped** cache, a block can be loaded into **exactly one** cache line.

- **Much** simpler to implement
- Easier to make it **fast**.
- But, it increases the chance of **conflicts**.

![Diagram of a direct mapped cache]

Place block #15 in cache line 7

7 = 15 mod 8
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 caches today are set-associative.
When bringing in new cache lines, an existing entry has to be \textit{evicted}.

No choice for direct-mapped caches.

Possible replacement strategies for fully- and set-associative caches:

- \textbf{Least Recently Used (LRU)}
  - Evict cache line whose last access was done longest time ago.
  - Due to temporal locality, it is least likely to be needed any time soon.

- \textbf{First In First Out (FIFO)}
  - Behaves often similar to LRU
  - But, it is easier to implement.

- \textbf{Random}
  - Pick a random cache line to evict.
  - Very simple to implement in hardware.

Replacement has to be done in \textbf{hardware} and \textbf{fast}. Hardware usually implements \textit{not most recently used}. 
Types of Cache Misses: 3 C’s!

**Compulsory (cold) miss:**
- Occurs on *first* access to a block.

**Conflict miss:**
- Occurs when the cache is large enough, but multiple blocks all *map to the same slot*.
- Can also happen due to bad alignment of struct elements.
- Direct-mapped caches have more conflict misses than N-way set-associative caches.

**Capacity miss:**
- Occurs when the set of active cache blocks (the *working set*) is *larger than the cache*.
- Note: fully-associative caches have only compulsory and capacity misses.
What happens on a write-hit?

Multiple copies of data exist (in cache and memory). What is the problem with that?

**Write-through:**
- Write immediately to memory and all caches in between
- Memory is always consistent with the cache copy and simplifies *data coherency*
- But each write will *stall the CPU* *
- *Slow:* what if the same value (or line!) is written several times?

**Write-back:**
- *Defer writing* to memory until cache line is evicted (replaced)
- Needs a *dirty bit* that indicates that the line is different from memory
- Has *higher performance* but is more complex to implement.

Modern processors usually implement *write back*.

* Write buffers can be used to overcome this problem.
What happens on a write-miss?

**Write-allocate** (load into cache, update line in cache):
- Good if more writes to the location will follow
- More complex to implement
- May evict an existing value
- Common with *write-back caches*.

**No-write-allocate** (writes immediately to memory):
- Simpler to implement
- Slower code (bad if value is consistently re-read)
- Seen with *write-through caches*.
Effect of Cache Parameters

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

cache misses (millions)

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

src: Ulrich Drepper. What Every Programmer Should Know About Memory
Real caches: Intel core i7-5960X

All caches have a cache line size of 64 bytes.

L1 instruction-cache (i-cache) and data-cache (d-cache):
- 32 KiB, 8-way set-associative
- i-cache: no writes, d-cache: write-back
- Access: 4 cycles

L2 unified cache:
- 256 KiB, 8-way set-associative
- Private, write-back
- Access: 11 cycles

L3 unified cache: (shared among multiple cores)
- 8 MiB, 16-way set-associative
- Shared, write-back
- Access: 30-40 cycles

Slower, but more likely to hit
Write code that has locality

- **Spatial**: access data contiguously
- **Temporal**: make sure access to the same data is not too far apart in time

How to achieve this?

- Adjust memory access in *code* (software) to improve miss rate (MR)
  - Requires knowledge of both how caches work as well as your system’s parameters
- Proper choice of algorithm
- Loop transformations
  - Cf. parallel programming class. We’ll cover them in a few weeks
Cache performance analysis
Example: matrix multiplication

```c
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++) { // move along rows of a
        for (j = 0; j < n; j++) { // move along columns of b
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
        }
    }
}
```

\[
\begin{align*}
c & \quad = \\
    \begin{bmatrix}
        a_{ij} \\
    \end{bmatrix} & \quad \times \\
    \begin{bmatrix}
        b_{kj} \\
    \end{bmatrix}
\end{align*}
\]
Cache miss analysis

Assume:
- Square matrix \((n \times n)\), elements are double, \(\text{sizeof}(\text{double})=8\)
- Cache-line is 64 bytes
- Single matrix row does not fit in the cache

First iteration:
- \(\frac{n}{8} + n = \frac{9n}{8}\) misses
- Afterwards in cache: (schematic)
  - Thrashing cached items before using them.
- Total misses: \(\frac{9n}{8} \times n^2 = \frac{9}{8} n^3\)
Can get the same result of matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)

For example, multiply two $4 \times 4$ matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

with $B$ defined similarly.

$$AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}.$$
Matrices of size $n \times n$, split into 4 blocks of size $r$ ($n = 4r$)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} \times B_{k2}$$

Multiplication operates on small “block” matrices

- Choose size so that they fit in the cache
- This technique called “cache blocking”
Blocked Matrix Multiply

```c
/* move by rxr BLOCKS now */
for (i = 0; i < n; i+=r)
    for (j = 0; j < n; j+=r)
        for (k = 0; k < n, k+=r)
            /* block matrix multiplication */
            for (ib = i; ib < i+r; ib++)
                for (jb = j; jb < j+r; jb++)
                    for (kb = k; kb < k+r; kb++)
                        c[ib*n + jb] += a[ib*n + kb] * b[kb*n + jb]
```

Blocked version of the naïve algorithm

- \( r \) = block matrix size (assume \( r \) divides \( n \) evenly)

6 nested loops may seem less efficient, but leads to a much faster code!!
Assume:

- Square matrix \((n \times n)\), elements are \text{double}, \text{sizeof(double)}=8
- Cache-line size is 64 bytes
- Single matrix row does not fit in the cache
- Three blocks \((r \times r)\) fit into cache: \(3r^2 < \text{cache size}\)

First (block) iteration:

- \(\frac{r^2}{8}\) misses for each block
- \(\frac{n}{r} \times 2 \times \frac{r^2}{8} = \frac{nr}{4}\) (again omitting matrix \(c\))

Afterwards in cache (schematic):
Cache Miss Analysis (Blocked)

Assume:
- Square matrix \((n \times n)\), elements are double, `sizeof(double)=8`
- Cache-line size is 64 bytes
- Single matrix row does not fit in the cache
- Three blocks \((r \times r)\) fit into cache: \(3r^2 \lt \text{cache size}\)

First (block) iteration:
- \(\frac{r^2}{8}\) misses for each block
- \(\frac{n}{r} \times 2 \times \frac{r^2}{8} = \frac{nr}{4}\) (again omitting matrix \(c\))

Total misses:
- \(\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{(4r)}\).
Matrix Multiply Summary

Naïve: \((9/8) \times n^3\)
Blocked: \(1/(4r) \times n^3\)

- If \(r = 8\), difference is \(4 \times 8 \times 9/8 = 36x\)
- If \(r = 16\), difference is \(4 \times 16 \times 9/8 = 72x\)

Blocking optimization only works if the blocks fit in the cache
- Suggests larger possible block size up to limit \(3r^2 \leq \text{cache size}\)

Matrix multiplication has inherent temporal locality:
- Input data: \(3n^2\), computation \(2n^3\)
- Every array element used \(O(n)\) times!
- But program has to be written properly
Cache-Friendly Code

Programmer can optimise for cache performance
- How data structures are organised
- How data are accessed:
  - Nested loop structure
  - Blocking is a general technique

All systems favour “cache-friendly code”
- Getting absolute optimum performance is very platform specific
  - Cache sizes, cache block size, associativity, etc.
- Can get most of the advantage with generic code:
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)
  - Focus on inner loop cycle
- Don’t optimize too much prematurely. Check the hotspots with a profiling tool like perf.
To compensate for slow memory, systems use caches:

- **DRAM** provides high capacity, but long latency \(\rightarrow\) main memory
- **SRAM** has better latency, but low capacity \(\rightarrow\) CPU caches
- Typically multiple levels of caching (memory hierarchy)
- Caches are organized into *cache lines* (smallest granularity for moving data blocks)
- **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* (temporal and spatial):
- Affects both data *and* code
- Concrete layout of caches in systems may be different, but locality always helps!
Cache Awareness for Data Processing
The Memory Mountain

Aggressive prefetching

Ridges of temporal Locality

Slopes of spatial Locality

Core i7 Haswell 2.1 GHz
32 KiB L1 cache
256 KiB L2 cache
8 MiB L3 cache
64 B block size

Read throughput (MB/s)
Size (bytes)
Stride (x8 bytes)
Working data set size (increasing)
Decreasing spatial locality
Cache-Friendly Code

Programmer can optimise for cache performance

- How data structures are organised (alignment and layout)
- How data are accessed:
  - Nested loop structure
  - Blocking is a general technique

All systems favour “cache-friendly code”

- Can get most of the advantage with generic code:
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)
  - Focus on inner loop cycle
First, we will look into data cache usage and how we can improve it. Afterwards, we will go over a few techniques that improve on instruction cache usage.
Question

Why do database systems show such poor data-cache behavior?
Caches for data processing

How can we improve data cache usage?

- Requires going back to different data storage models and query execution models.
- And thinking both in terms of temporal- and spatial-locality

Let’s consider as an example the following selection query:

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

Which typically involves a full table scan.
Tuples are represented as **records** stored sequentially on a database page.

- With every access to `l_shipdate` field, we load a large amount of *irrelevant* data into the cache.
- Accesses to slot directories and variable sized tuples incur additional trouble.
- Especially present in OLAP workloads.
Improving data cache locality
Data alignment

Word and cache *aligned attributes* with *padding* are essential to enable the CPU to access elements without any unexpected behavior or additional work
Structure represented as block of memory:
- Big enough to hold all of the fields

Fields ordered according to declaration order
- Even if another ordering would be more compact

Compiler determines overall size + positions of fields
- Machine-level programs has no understanding of the structures in the source code
Memory Alignment in x86-64

For good memory system performance, Intel recommends data to be aligned

- Memory is accessed in word-chunks, so it is inefficient to load/store values that span word boundaries and especially cache-line boundaries.

- However, the x86-64 hardware will work correctly regardless of alignment of data.

_Aligned_ means that any primitive object of \( K \) bytes must have an address that is multiple of \( K \).

<table>
<thead>
<tr>
<th>( K )</th>
<th>Type</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
<td>No restrictions</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td>Lowest bit must be zero: ( ...0_2 )</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
<td>Lowest 2 bits zero: ( ...00_2 )</td>
</tr>
<tr>
<td>8</td>
<td>long, double</td>
<td>Lowest 3 bits zero: ( ...000_2 )</td>
</tr>
<tr>
<td>16</td>
<td>long double</td>
<td>Lowest 4 bits zero: ( ...0000_2 )</td>
</tr>
</tbody>
</table>
Structures and Alignment

Aligned Data:
- Primitive data type requires $K$ bytes
- Address must be multiple of $K$

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

Even though it is not packed, this padded data structure will result in better performance.
Alignment of Structs

Compiler will do the following:

- Maintains declared **ordering** of fields in struct
- Each **field** must be aligned **within** the struct (*may insert padding*)
  - `offsetof` can be used to get actual field offset
- Overall struct must be **aligned** according to largest field
- Total struct **size** must be multiple of its alignment (*may insert padding*)
  - `sizeof` should be used to get true size of structs
- For strings and other variable-length data
  - split the string into length and data: fixed size header and variable size tail.
  - header contains pointers to tail.
  - place variable data at the end of the **struct** (consider as alignment 1)
  - Cf. Database Systems on Modern CPU Architectures (Access Paths)
How you can save space

The compiler must respect the order elements are declared in

- Sometimes the programmer can save space by declaring large data types first

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

12 bytes

8 bytes
Data alignment

**Task:** test effect of padding and alignment when inserting tuples in an array (single socket, 4 hw-threads)

```c
struct S1
{
    int primary_key;
    long timestamp;
    char color[2];
    int zipcode;
} *p;
```

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Alignment</td>
<td>0.523 MB/s</td>
</tr>
<tr>
<td>Padding</td>
<td>11.7 MB/s</td>
</tr>
<tr>
<td>Reordering + Padding</td>
<td>814.8 MB/s</td>
</tr>
</tbody>
</table>

→ src: [CMU-DB Alignment Experiment](#) by Tianyu Li
Storage model: Option 1 row-store

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

- Ideal for OLTP where txns tend to operate only on an individual entry and insert- or update-heavy workloads.
- Good for:
  + Inserts, updates, and deletes.
  + Queries that need the entire tuple.
  + Index-oriented physical storage.
- Bad for:
  - Scanning large portions of the table and/or a subset of the attributes.

![Diagram of row-wise storage]

Ideal for OLTP where txns tend to operate only on an individual entry and insert- or update-heavy workloads.

Good for:
+ Inserts, updates, and deletes.
+ Queries that need the entire tuple.
+ Index-oriented physical storage.

Bad for:
- Scanning large portions of the table and/or a subset of the attributes.
Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.

**Good for:**
- Only reads the data that it needs.
- Amortizes cost for fetching data from memory.
- Better for compression.

**Bad for:**
- Point queries, inserts, updates, and deletes because of tuple splitting/stitching.

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

→ Copeland and Khoshafian. A Decomposition Storage Model. *SIGMOD 1985*

Page 0

Page 1
**Tuple identification**
- Fixed length offsets – each value is the same length for an attribute
- Embedded Tuple IDs – each value is stored with its tuple id in a column

**Example:** MonetDB makes this explicit in its data model with Binary Association Tables
- 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>

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>John</td>
</tr>
<tr>
<td>o2</td>
<td>Angelina</td>
</tr>
<tr>
<td>o3</td>
<td>Scott</td>
</tr>
<tr>
<td>o4</td>
<td>Nancy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>34</td>
</tr>
<tr>
<td>o2</td>
<td>31</td>
</tr>
<tr>
<td>o3</td>
<td>35</td>
</tr>
<tr>
<td>o4</td>
<td>33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>m</td>
</tr>
<tr>
<td>o2</td>
<td>f</td>
</tr>
<tr>
<td>o3</td>
<td>m</td>
</tr>
<tr>
<td>o4</td>
<td>f</td>
</tr>
</tbody>
</table>

- Each column yields one **binary association table (BAT)**, with oids to identify matching entries
- Often, the oids can be implemented as virtual oids (voids) → not explicitly materialized in memory
Column stores: tuple reconstruction

**Tuple re-combination** can cause considerable overhead:

- Need to perform many joins
- Workload-dependent trade-off
- MonetDB positional joins (thanks to void columns)

→ Column-stores vs. row-stores: How different are they really? *SIGMOD 2008*
Column Stores in Commercial DBMS

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: MonetDB, VectorWise, Vertica

2010s: Almost all commercial databases added extensions to their engines

- Microsoft SQL Server (since SQL Server 11)
  - Column Store Indexes (Larson et al. SIGMOD 2011)
- Oracle
  - Dual-format in-memory option (Lahiri et al. ICDE 2015)
- IBM DB2 (since DB2 10.5)
  - BLU Accelerator (Raman et al. VLDB 2013), enhancing Blink (Raman et al. ICDE 2008)

Edgar F. Codd Innovation Award, and ACM SIGMOD Systems Award (MonetDB)

ACM SIGMOD Test of Time Award for C-Store
One can also store data in a hybrid format:

- **PAX (Partition Attributes Across) layout:**
  - Divide each page into mini-pages and group attributes into them
  - *Weaving Relations for Cache Performance* by Ailamaki et al. (VLDB 2001)

- **Hybrid storage model**
  - Store new data in NSM for fast OLTP
  - Migrate data to DSM for more efficient OLAP
  - *Fractured mirrors* (Oracle, IBM), *Delta Store* (SAP Hana)

- **Recent research states that DSM can be used efficiently for hybrid workloads**
  - *Optimal Column Layout for Hybrid Workloads* by Athanassoulis et al. (VLDB 2019)
Books:
- “Computer Systems: A Programmer’s Perspective” (3rd edition) by Bryant and O’Hallaron
- “What Every Programmer Should Know About Memory” by Ulrich Drepper

Lecture: Introduction to Computer Architecture by myself (Imperial College London)
Lecture: Data Processing on Modern Hardware by Prof. Jens Teubner (TU Dortmund)
Lecture: Advanced Databases by Prof. Andy Pavlo (CMU)

Various research papers references in the slides:
- “A Decomposition Storage Model” by Copeland and Khoshafian SIGMOD 1985
- “Column-stores vs. row-stores: How different are they really?” by Abadi et al. SIGMOD 2008
- “Weaving Relations for Cache Performance” by Ailamaki et al. VLDB 2001
- “Optimal Column Layout for Hybrid Workloads” by Athanassoulis et al. VLDB 2019