Jens Teubner, TU Dortmund, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2015
Part II

Cache Awareness
Hardware Trends

- **Source:** Hennessy & Patterson, Computer Architecture, 4th Ed.
- **Data Processing on Modern Hardware · Summer 2015**
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 ≠ Memory

Dynamic RAM (DRAM)

- State kept in capacitor
- Leakage
  → refreshing needed

Static RAM (SRAM)

- Bistable latch (0 or 1)
- Cell state stable
  → no refreshing needed
Dynamic RAM is comparably slow.

- Memory needs to be refreshed periodically (≈ every 64 ms).
- (Dis-)charging a capacitor takes time.

DRAM cells must be addressed and capacitor outputs amplified.

Overall we’re talking about ≈ 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 is done for an entire row.
- Once this is done, more than one word can be read out.

In addition,

- Several DRAM cells can be used in parallel.
  → Read out even more words in parallel.

We can exploit that by using sequential access patterns.
SRAM Characteristics

SRAM, by contrast, can be very fast.
- Transistors actively drive output lines, access almost instantaneous.

But:
- SRAMs are significantly more expensive (chip space $\equiv$ money)

Therefore:
- Organize memory as a hierarchy.
- Small, fast memories used as caches for slower memory.
Some systems also use a 3rd level cache.

- cf. Architecture & Implementation course
  - Caches 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 re-use data frequently.
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.

Block-wise transfers are well-supported by DRAM chips.
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.\(^1\)

\(^1\)Modern CPUs support out-of-order execution and several in-flight cache misses.
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
  - $4 \text{ MB cache, line size: } 64 \text{ B: } 65,536 \text{ cache lines.}$
- Used, *e.g.*, for small TLB caches.
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

\(4 = 12 \mod 8\)
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

The diagram shows the effect of cache size on cache misses for different cache types: direct-mapped, 2-way associative, 4-way associative, and 8-way associative. The cache sizes range from 512 KB to 16 MB. The x-axis represents the cache size, and the y-axis represents the number of cache misses in millions. The bars indicate the number of cache misses for each cache type at each cache size.
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**.

- **tag**: portion of the memory address
- **set index**: portion of the memory address
- **offset**: portion of the memory address

The diagram shows the memory address and its components.
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 \((4 \text{ MB} \div 64 \text{ bytes})\).
- Associativity: **16-way set-associative**.
  - → There are 4,096 sets \((65,536 \div 16 = 4,096)\).
  - → 12-bit set index \((2^{12} = 4,096)\).
- Maximum physical address space: **64 GB**.
  - → 36 address bits are enough \((2^{36} \text{ bytes} = 64 \text{ GB})\)
  - → 18-bit tags \((36 - 12 - 6 = 18)\).

<table>
<thead>
<tr>
<th>tag</th>
<th>set index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 bit</td>
<td>12 bit</td>
<td>6 bit</td>
</tr>
</tbody>
</table>
Block Replacement

When bringing in new cache lines, an existing entry has to be **evicted**.

Different strategies are conceivable (and meaningful):

**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.
- Affects data and code.
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, 2-way set-associative
- **L2 cache**: shared cache, 1 MB, 64 B cache lines, 16-way set-associative, pseudo-LRU policy
- **L1 hit latency**: 2 cycles
- **L2 hit latency**: 7 cycles (for first word)
- **L2 miss latency**: 160–180 cycles
  (20 CPU cycles + 140 cy DRAM latency (50 ns) + 20 cy on mem. bus)
- **L2 cache**: write-back
- **40-bit virtual addresses**

Performance (SPECint 2000)

misses per 1000 instructions

<table>
<thead>
<tr>
<th>benchmark program</th>
<th>L1 Instruction Cache</th>
<th>L2 Cache (shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crafty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perlbmk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vortex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPC-C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Jens Teubner · Data Processing on Modern Hardware · Summer 2015
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.
How can we improve data cache usage?

Consider, e.g., a selection query:

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

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

- With every access to a \texttt{l\_shipdate} field, we load a large amount of irrelevant information into the cache.
- Accesses to slot directories and variable-sized tuples incur additional trouble.
Row-Wise vs. Column-Wise Storage

Remember the “Architecture & Implementation” course?

The \(n\)-ary storage model (NSM, row-wise storage) is not the only choice.

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

© Jens Teubner · Data Processing on Modern Hardware · Summer 2015

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>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>
<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).
- Oftentimes, 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)
Commercial databases have just recently announced column-store extensions to their engines:

- **Microsoft SQL Server**:  
  - Represented as “Column Store Indexes”  
  - Available since SQL Server 11  
  - see Larson et al., SIGMOD 2011

- **IBM DB2**:  
  - IBM announced DB2 “BLU Accelerator” last week, a column store that is going to ship with DB2 10.5.  
  - BLU stands for “Blink Ultra”; Blink was developed at IBM Almaden (Raman et al., ICDE 2008).
A hybrid approach is the **PAX (Partition Attributes Accross)** layout:

- Divide each page into **minipages**.
- Group attributes into them.

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
Consequences:

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

- Operators constantly call each other’s functionality.
  - Large \textit{function call overhead}.

- The combined \textit{state} may be too large to fit into caches.
  - \textit{E.g.}, hash tables, cursors, partial aggregates.
  - \textit{Data cache misses}.
Example: Query Q1 from the TPC-H benchmark on MySQL.

```
SELECT l_returnflag, l_linestatus, 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_linestatus
```

Scan query with arithmetics and a bit of aggregation.

Results taken from Peter Boncz, Marcin Zukowski, Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR 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_ulint_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>23M</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>2.9</td>
<td>17M</td>
<td>79</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>17M</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_fieldExtern_bit</td>
</tr>
<tr>
<td><strong>1.0</strong></td>
<td>11M</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>5.9M</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.
- Very low **instructions-per-cycle** (IPC) ratio.

Further:

- Much time spent on **field access** (e.g., `rec_get_nth_field()`).
  - NSM \(\leadsto\) polymorphic operators.
- Single-tuple functions hard to optimize (by compiler).
  - \(\rightarrow\) Low instructions-per-cycle ratio.
  - \(\rightarrow\) Vector instructions (SIMD) hardly applicable.
- Function call overhead.
  - \(\frac{38}{0.8} \text{ instr. per cycle} = 48 \text{ cycles vs. 3 instr. for load/add/store assembly.}\)
Operator-At-A-Time Processing

MonetDB: **operator-at-a-time processing**.

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

Example:

```plaintext
sel_age := people_age.select(30, nil);
sel_id := sel_age.mirror().join(people_age);
sel_name := sel_age.mirror().join(people_name);
tmp := [-](sel_age, 30);
sel_bonus := [*](50, tmp);
```
Operator-At-A-Time Processing

Function call overhead is now replaced by **extremely tight loops**.

**Example:** `batval_int_add(···)` (impl. of `[+](int, BAT[any,int])`)

```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 {
    for (; bp < bq; bp++, bnp++) {
        *bnp = vv;
    }
}
...
```
Tight Loops

These tight loops

- 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**.

Note also:

- **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>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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,724</td>
<td>365</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 ← X100 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 “vector” in MonetDB/X100 terminology.

Choose vector size

- **large enough** to compensate for iteration overhead (function calls, instruction cache misses, …), but
- **small enough** to not thrash data caches.

⚠️ Will there be such a vector size? (Or will caches be thrashed long before iteration overhead is compensated?)
Vectorized execution quickly compensates for iteration overhead.

- 1000 tuples should conveniently fit into caches.
Vectorized Execution in MonetDB/X100

MonetDB/X100 architecture

Query tree...

- **Select**
  - `select_le_date_col_date_val`
  - `1998−09−02`

- **Project**
  - `vat_price`
  - `map_mul_flt_val_flt_col`
  - `3`

- **Aggregate**
  - `hash table maintenance`
  - `aggr_sum_flt_col`

- **Scan**
  - `shipdate`
  - `returnflag`
  - `exprice`

- **Vectors**
  - contain multiple values of a single attribute

- **Primitives**
  - process entire vectors at a time

- **Operators**
  - process sets of tuples represented as aligned vectors

Effect on Query Execution Time

![Graph showing the effect on query execution time.](image)

**Figure 10:** Query 1 performance w.r.t. vector-size.

**Table 4:** TPC-H Performance (seconds)

<table>
<thead>
<tr>
<th>Vector size (tuples)</th>
<th>AthlonMP</th>
<th>Itanium2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.72</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>0.46</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>16</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>32</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>64</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>128</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>256</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>512</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>4K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>8K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>16K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>32K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>64K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>128K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>256K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>512K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>4M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Table 5:** Example Trace of Query 1

<table>
<thead>
<tr>
<th>Vector size (tuples)</th>
<th>AthlonMP</th>
<th>Itanium2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.72</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>0.46</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>16</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>32</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>64</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>128</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>256</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>512</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>4K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>8K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>16K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>32K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>64K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>128K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>256K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>512K</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>4M</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Figure 9:** Query 1 in X100 Algebra.

**Table 5:** Example Trace of Query 1
## Comparison of Execution Models

Overview over discussed execution models:

<table>
<thead>
<tr>
<th>execution model</th>
<th>tuple</th>
<th>operator</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>query plans</td>
<td>simple</td>
<td>complex</td>
<td>simple</td>
</tr>
<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</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>

Microsoft SQL Server supports vectorized ("batched" in MS jargon) execution since version 11.

- Storage via new **column-wise index**.
  - Includes **compression** and **prefetching improvements**.
- New operators with **batch-at-a-time processing**.
  - Can combine row- and batch-at-a-time operators in one plan.
  - CPU-optimized implementations.

↑ Per-Åke Larson *et al.* SQL Server Column Store Indexes. *SIGMOD 2011.*
- Tables divided into row groups ($\approx 1$ million rows)
- Each group, each column compressed independently.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode, compress</td>
<td>Encode, compress</td>
<td>Encode, compress</td>
<td>Compressed column segments</td>
</tr>
</tbody>
</table>

Figure 1: Converting rows to column segments

Figure 2: Storing column segments

The dictionary based encoding transforms a set of distinct values into a uniform type: a 32-bit or 64-bit integer. Two types of encoding are supported: a dictionary based encoding and a value based encoding.
Segment Organization

- **Segment directory** keeps track of segments.
- Segments are stored as **BLOBs** (“binary large objects”)
  - Re-use existing SQL Server functionality.
- Statistics (min/max values) for each segment.

---

**Segment Organization**

![Segment Organization Diagram]

- Blobs
  - Row group 1
  - Row group 2
  - Row group 3

---

- **Segment directory** keeps track of segments.
- Segments are stored as **BLOBs** (“binary large objects”)
  - Re-use existing SQL Server functionality.
- Statistics (min/max values) for each segment.
I/O Optimizations

Column-store indexes are designed for scans.

- **Compression** (RLE, bit packing, dictionary encoding)
  - Re-order row groups for best compression.

- Segments are forced to be contiguous on disk.
  - Unlike typical page-by-page storage.
  - Pages and segments are automatically prefetched.

<table>
<thead>
<tr>
<th>data set</th>
<th>uncompressed</th>
<th>column-store idx</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmetics</td>
<td>1,302</td>
<td>88.5</td>
<td>14.7</td>
</tr>
<tr>
<td>SQM</td>
<td>1,431</td>
<td>166</td>
<td>8.6</td>
</tr>
<tr>
<td>Xbox</td>
<td>1,045</td>
<td>202</td>
<td>5.2</td>
</tr>
<tr>
<td>MSSales</td>
<td>642,000</td>
<td>126,000</td>
<td>5.1</td>
</tr>
<tr>
<td>Web Analytics</td>
<td>2,560</td>
<td>553</td>
<td>4.6</td>
</tr>
<tr>
<td>Telecom</td>
<td>2,905</td>
<td>727</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Batched Execution

Similar to the X100/Vectorwise execution model, batch operators in SQL Server can process batches of tuples at once.

- Can mix batch- and row-based processing in one plan.
- Typical pattern:
  - Scan, pre-filter, project, aggregate data early in the plan using batch operators.
  - Row operators may be needed to finish the operation.
- Good for scan-intensive workloads (OLAP), not for point queries (OLTP workloads).
- Internally, optimizer treats batch processing as new physical property (like sortedness) to combine operators in a proper way.
Performance impact (TPC-DS, scale factor 100, ≈ 100 GB):

- Q1: 100 ms
- Q2: 1 s
- Q3: 10 s
- Q4: 100 s

execution time

query number (TPC-DS)

row store

column store

source: Larson et al., SQL Server Column Store Indexes. SIGMOD 2011 (elapsed times, warm buffer pool).
Alternative: Buffer Operators

A similar effect can be achieved in a less invasive way by placing buffer operators in a pipelined execution plan.

- Organize query plan into execution groups.
- Add buffer operator between execution groups.
- Buffer operator provides tuple-at-a-time interface to the outside,
  - but batches up tuples internally.

A buffer operator can be plugged into every Volcano-style engine.

```
1 Function: next ()

   // Read a batch of input tuples if buffer is empty.
2 if empty and !end-of-tuples then
   while !full do
      append child.next () to buffer ;
   if end-of-tuples then
      break ;
   // Return tuples from buffer
7 return next tuple in buffer ;
```
Buffer Operators in PostgreSQL

Figure 14: Query 3
In-Memory Joins

After plain select queries, let us now look at join queries:

```sql
SELECT COUNT (*)
FROM orders, lineitem
WHERE o_orderkey = l_orderkey
```

(We want to ignore result construction for now, thus only count result tuples.)

We assume:

- no exploitable order,
- no exploitable indices (input might be an intermediate result), and
- an equality join predicate (as above).
Hash Join

**Hash join** is a good match for such a situation.

To compute $R \times S$,

1. **Build a hash table** on the “outer” join relation $R$.  
2. **Scan** the “inner” relation $S$ and **probe** into the hash table for each tuple $s \in S$.

```
1 Function: hash_join (R, S)
            // Build Phase
2 foreach tuple r ∈ R do
3         insert s into hash table H ;
            // Join Phase
4 foreach tuple s ∈ S do
5         probe H and append matching tuples to result ;
```
Hash Join

$R$ scan $\rightarrow h \rightarrow \vdots \rightarrow h \rightarrow \vdots \rightarrow h \leftarrow S$ scan

1. build
2. probe

✓ $O(N)$ (approx.)
✓ Easy to **parallelize**
Parallel Hash Join

- **R** \(\rightarrow h \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow b_k \rightarrow S\)

- **Build**
- **Probe**

✓ Protect using locks; **very low contention**
Random access pattern
→ Every hash table access a cache miss

Cost per tuple (build phase):
- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

\{ hash join is severely latency-bound \}
Partitioned Hash Join

Thus: **partitioned hash join**  [Shatdal et al. 1994]

![Diagram of partitioned hash join](attachment:image.png)

- **Partition**: Divide the input data into partitions.
- **Build**: Distribute the partitions to the processing units.
- **Probe**: Perform the join operation on each partition.

(parallelism: assign partitions to threads → no locking needed)
Cache Effects

Build/probe now contained within caches:
- 15/21 instructions per tuple (build/probe)
- \( \approx 0.01 \) cache misses per tuple
- almost no TLB misses

Partitioning is now critical

- Many partitions, far apart
- Each one will reside on its own page
- Run out of TLB entries (100–500)
Cost of Partitioning

\[
\text{for all input tuples } t \text{ do}
\begin{align*}
    h & \leftarrow \text{hash}(t\cdot\text{key}) \\
    \text{out}[\text{pos}[h]] & \leftarrow t \\
    \text{pos}[h] & \leftarrow \text{pos}[h] + 1
\end{align*}
\text{end for}
\]

\[\rightarrow \text{Expensive beyond } \approx 2^8-2^9 \text{ partitions.}\]
Multi-pass partitioning ("radix partitioning")

one hash table per partition

pass 1
① partition
② build
③ probe
① partition

pass 2

R

\( h_{1,1} \)
\( h_{1,2} \)
\( r_1 \rightarrow h_2 \)
\( r_2 \)
\( r_3 \)
\( r_4 \)
\( h_2 \)

S

\( h_{1,1} \)
\( h_{1,2} \)
\( s_1 \)
\( s_2 \)
\( s_3 \)
\( s_4 \)
Multi-pass partitioning ("radix partitioning")

In practice:

- $h_1, \ldots, h_P$ use same hash function but look at different bits.
Two-pass partitioning

![Graph showing throughput versus radix bits for single-pass and two-pass partitioning. The graph illustrates that two-pass partitioning generally has a lower throughput compared to single-pass partitioning as the radix bits increase.]
Hash join is $O(N \log N)$!
for all input tuples \( t \) do
\[
h \leftarrow \text{hash}(t.\text{key})
\]
copy \( t \) to \( \text{out}[\text{pos}[h]] \)
\[
\text{pos}[h] \leftarrow \text{pos}[h] + 1
\]
end for

Naïve partitioning (cf. slide 78)

for all input tuples \( t \) do
\[
h \leftarrow \text{hash}(t.\text{key})
\]
\[
\text{buf}[h][\text{pos}[h] \mod \text{bufsiz}] \leftarrow t
\]
if \( \text{pos}[h] \mod \text{bufsiz} = 0 \) then
\[
\text{copy} \ \text{buf}[h] \ \text{to} \ \text{out}[\text{pos}[h] - \text{bufsiz}]
\]
end if
\[
\text{pos}[h] \leftarrow \text{pos}[h] + 1
\]
end for

Software-Managed Buffers

→ TLB miss only every \( \text{bufsiz} \) tuples
→ Choose \( \text{bufsiz} \) to match cache line size
Software-Managed Buffers

The diagram illustrates the throughput in million tuples per second for different radix bits for single-pass, two-pass partitioning, and software-managed buffers. The throughput decreases as the radix bits increase for all methods, with software-managed buffers showing the highest throughput for a given number of radix bits.
Plugging it together

Blanas et al.: 86.4 / 64.6 cy/tpl

Nehalem: 4 cores/8 threads; 2.26 GHz

Sandy Bridge: 8 cores/16 threads; 2.7 GHz

AMD Bulldozer: 16 cores; 2.3 GHz

Niagara T2: 8 cores/64 threads; 1.2 GHz

256 MiB × 4096 MiB

e.g., Nehalem: 25 cy/tpl ≈ 90 million tuples per second
Another Workload Configuration

<table>
<thead>
<tr>
<th>CPU</th>
<th>Cores/Threads</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>4/8</td>
<td>2.26 GHz</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>8/16</td>
<td>2.7 GHz</td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>16</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td>Niagara 2: T2</td>
<td>8</td>
<td>1.2 GHz</td>
</tr>
</tbody>
</table>

- **977 MiB × 977 MiB**
- **e.g., Nehalem: 25 cy/tpl ≈ 90 million tuples per second**
Overall performance is influenced by a number of parameters:

- input data volume
- cluster size / number of clusters
- number of passes (plus number of radix bits per pass)

An **optimizer** has to make the right decisions at runtime.

- Need a detailed **cost model** for this.
With column-based storage, a single join is not enough.

- Joining BATs for key attributes yields a **join index**.
- **Post-project** BATs for all remaining attributes.
Positional lookup?

- Makes post-projection joins “random access” 😊

Thus:

- **(Radix-)Sort** by oids of larger relation
  - Positional lookups become cache-efficient.

- **Partially cluster** by oids before positional join of smaller relation
  - Access to smaller relation becomes cache-efficient, too.