Part II

Cache Awareness
Hardware Trends

![Graph showing normalized performance trends for processors and DRAM memory from 1980 to 2005. The x-axis represents the year, and the y-axis represents normalized performance. The graph shows a significant increase in performance over time, with processors showing a more steep increase compared to DRAM memory.]

Source: Hennessy & Patterson, Computer Architecture, 4th Ed.
There is an increasing gap between CPU and memory speeds.

- Also called the memory wall.
- CPUs spend much of their time waiting for 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.
DRAM Characteristics

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 ≡ money)

Therefore:

- Organize memory as a hierarchy.
- Small, fast memories used as caches for slower memory.
Memory Hierarchy

<table>
<thead>
<tr>
<th>CPU</th>
<th>technology</th>
<th>capacity</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>SRAM</td>
<td>bytes</td>
<td>&lt; 1 ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>SRAM</td>
<td>kilobytes</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>main memory</td>
<td>SRAM</td>
<td>megabytes</td>
<td>&lt; 10 ns</td>
</tr>
<tr>
<td></td>
<td>DRAM</td>
<td>gigabytes</td>
<td>70–100 ns</td>
</tr>
</tbody>
</table>

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

Caches take advantage of the principle of locality.

- 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.
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.\(^2\)

\(^2\)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 MB cache, line size: 64 B: 65,536 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**.
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 graph shows the number of cache misses (in millions) for different cache sizes and associativities. The x-axis represents cache size in megabytes (512 kB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB), and the y-axis represents the number of cache misses (in millions).

- The graph includes data for direct-mapped, 2-way associative, 4-way associative, and 8-way associative caches.

Source: Ulrich Drepper. What Every Programmer Should Know About Memory
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**.
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 ($36 - 12 - 6 = 18$).
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**.\(^3\)
  - 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**.

---

\(^3\)**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, 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)

The graph shows the misses per 1000 instructions for different benchmark programs. The x-axis represents the benchmark programs, and the y-axis represents the misses per 1000 instructions. The two bars for each program indicate the misses in the L1 Instruction Cache and the L2 Cache (shared).

The programs are:
- gzip
- vpr
- gcc
- mcf
- crafty
- parser
- eon
- perl.bmk
- gap
- vortex
- bzip2
- twolf
- avg
- TPC-C

The TPC-C benchmark shows a high number of misses in the L1 Instruction Cache, significantly higher than the other programs.
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)

- 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:

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

This query typically involves a full table scan.
Table Scans (NSM)

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):
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>
</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>

Object identifiers (oids) identify matching entries (BUNs).
- Oftentimes, oids can be implemented as **virtual** oids (voids).
  - Not explicitly materialized in memory.

Each column yields one **binary association table (BAT)**.
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 **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: 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_uint_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_fieldExtern_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

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 → polymorphic operators.
- Single-tuple functions hard to optimize (by compiler).
  - Low instructions-per-cycle ratio.
  - Vector instructions (SIMD) hardly applicable.
- Function call overhead.
  - \[
    \frac{38 \text{ instr.}}{0.8 \text{ instr./cycle}} = 48 \text{ cycles} \quad \text{vs.} \quad 3 \text{ instr. for load/add/store assembly.}
  \]
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;
    }
}
```

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
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>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>
<tr>
<td></td>
<td>3,724</td>
<td>365</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?

```
<table>
<thead>
<tr>
<th>tuple-at-a-time</th>
<th>operator-at-a-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>↑</td>
</tr>
</tbody>
</table>
```

**X100 vectorized execution**
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


Section 4.1: MonetDB/X100 architecture

To reduce bandwidth requirements, it uses a vertically fragmented data layout, in some cases enhanced with lightweight...
A first observation is that X100 manages to run all primitives at a very low number of CPU cycles per tuple — for example, which is way better than the 49 cycles per tuple that these fetch-joins are truly efficient, as they cost less than 2 cycles per tuple.

Finally, Table 5 shows that Query 1 uses three columns that are stored in enumerated types (i.e. of 500MB/s, MonetDB/X100 exceeds 7.5GB/s on the same operator.

We now investigate the influence of vector size on performance. X100 uses a default vector size of 1024, but users can override it. Preferably, all vectors together should comfortably fit the CPU cache size, hence they should not be too big. However, with really small vectors, the possibility of exploiting CPU parallelism should not be too big. Nonetheless, with very small vectors the interpretation overhead in the X100 Algebra next() methods disappears. Also, in that case, the impact of interpretation overhead also hits MonetDB/X100 strongly if it uses tuple-at-a-time processing.

A second observation is that since a large part of the data that is being processed by primitives comes from the (coarser) level of X100 algebra operators. X100 implements detailed tracing output generated by running TPC-H Query 1 in detail. Figure 9 shows its translation to help analyze query performance. Table 5 shows the study the performance of MonetDB/X100 on TPC-H. MonetDB/MIL was constrained by the RAM bandwidth really high bandwidth. Where multiplication in MonetDB/X100 is handled in 2.2 cycles per tuple — even relatively complex primitives like aggregations run in 6 cycles per tuple. Notice that a multiplication (of 500MB/s, MonetDB/X100 exceeds 7.5GB/s on the same operator.

Table 4: TPC-H Performance (seconds)

<table>
<thead>
<tr>
<th>Vector size (tuples)</th>
<th>AthlonMP</th>
<th>Itanium2</th>
<th>DB2, 4CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>3.72</td>
<td>0.46</td>
<td>0.1</td>
</tr>
<tr>
<td>2K</td>
<td>0.46</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>4K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>8K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>16K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>32K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>64K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>128K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>256K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>512K</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>1M</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>2M</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>4M</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Please refer to Figure 10: Query 1 performance w.r.t. vector-size impact. The graph shows the time (seconds) on the y-axis and the vector size (tuples) on the x-axis. The graph includes three lines representing AthlonMP, Itanium2, and DB2, 4CPU, respectively. The graph indicates that as the vector size increases, the time taken to process the query decreases, with AthlonMP showing the best performance across all vector sizes.
## 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** ($\approx 1M$ rows)
- Each group, each column **compressed** independently.

<table>
<thead>
<tr>
<th>Row group 1</th>
<th>Row group 2</th>
<th>Row group 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode, compress</td>
<td>Encode, compress</td>
<td>Encode, compress</td>
</tr>
<tr>
<td>Compressed column segments</td>
<td>Compressed column segments</td>
<td>Compressed column segments</td>
</tr>
</tbody>
</table>

**Data Encoding and Compression**

- Each group, each column compressed independently.
- Encoding step transforms column values into a uniform type: a 32-bit or 64-bit integer. Two types of encoding are supported: dictionary based encoding and a value based encoding.
- The results are shown in Table 1. The column store index improves performance dramatically: the query consumes 13 times less CPU time and runs 25 times faster with a warm buffer pool. SQL Server column store technology gives subsecond response times for a star join query against a 1.44 billion row table on a commodity machine. This level of improvement is significant, especially considering that the query enhancements in SQL Server 2008 are provided on top of a pre-existing technology.
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.
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

**Row store** vs. **Column store**

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

```plaintext
Function: next ()

// Read a batch of input tuples if buffer is empty.
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
  return next tuple in buffer ;
```

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
Buffer Operators in PostgreSQL

Our experience indicates that the threshold is not very sensitive to the choice of operator. By changing the predicate in the operator “TableScan,” we can control the cardinality of the output of the table scan operator, thus adjusting its footprint. This is because it is a foreign-key join and the optimizer can “miss” the join. Figure 11 shows the query performance of the experiments.

For the buffered plan, footprint analysis suggests two execution groups (marked with boxes). Note that there is no obvious benefit shown for even larger buffer sizes. This is because the data is allocated (or accessed) sequentially, hardware prefetching hides most of the L2 data cache miss latency. These results are repeated for different buffer sizes. Figure 13 shows the execution time breakdown for different buffer sizes.

Another buffering parameter is the size of the array used to store intermediate tuples. The size is set during operator initialization. The number of reduced trace cache misses is roughly proportional to \( \frac{1}{\text{buffersize}} \) and thus incurs more L2 data cache misses. It is tempting to conclude that these L2 misses may be important, but the total memory requirement is less than L2 cache size. Therefore, there is a relatively large amount of computation that we can achieve good query performance with a moderate buffer size. Figure 14 shows a query that joins the tables “lineitem” and “order.”

We use a two-table join query to demonstrate how buffer operators can be used in more complex situations. Figure 15 shows the performance of Query 1, with different buffer sizes. TableScan is invoked. Figure 11 shows the query performance of the experiments.

For the buffered plan, footprint analysis suggests two execution groups (marked with boxes). Note that there is no obvious benefit shown for even larger buffer sizes. This is because the data is allocated (or accessed) sequentially, hardware prefetching hides most of the L2 data cache miss latency. These results are repeated for different buffer sizes. Figure 13 shows the execution time breakdown for different buffer sizes.

A bigger buffer size means that the child of the buffer operator added above the “IndexScan” operator, even though its footprint is larger than the L1 instruction cache. This is because it is a foreign-key join and the optimizer can “miss” the join. Figure 11 shows the query performance of the experiments.

For the buffered plan, footprint analysis suggests two execution groups (marked with boxes). Note that there is no obvious benefit shown for even larger buffer sizes. This is because the data is allocated (or accessed) sequentially, hardware prefetching hides most of the L2 data cache miss latency. These results are repeated for different buffer sizes. Figure 13 shows the execution time breakdown for different buffer sizes.
In-Memory Joins

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

```
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 $R \Join S$,

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

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

1. **build**

2. **probe**

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

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

- **Partition** $R$ and $S$ into $h_1$, $h_2$, etc.
- **Build** hash tables $h_1$, $h_2$, etc.
- **Probe** $h_1$ against $h_2$, etc.

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

Build/probe now contained within caches:
- 15/21 instructions per tuple (build/probe)
- ≈ 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)
for all input tuples \( t \) do
\[
h \leftarrow \text{hash}(t.\text{key})
\]
\[
\text{out}[\text{pos}[h]] \leftarrow t
\]
\[
\text{pos}[h] \leftarrow \text{pos}[h] + 1
\]
end for

→ Expensive beyond \( \approx 2^8 - 2^9 \) partitions.
Multi-pass partitioning ("radix partitioning")

**Diagram Description:**
- **Step 1:** Partitioning
- **Step 2:** Building
- **Step 3:** Probing

Each pass involves:
- **Pass 1:** Scan and partition
- **Pass 2:** Build and probe

**Key Elements:**
- **R:** Source table
- **S:** Result table
- **h1,1, h1,2:** Hash functions
- **r1, r2, r3, r4:** Records
- **s1, s2, s3, s4:** Hash tables

**Legend:**
- **scan**
- **partition**
- **build**
- **probe**

**Notes:**
- One hash table per partition

---

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
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 vs. radix bits for single-pass and two-pass partitioning. The graph indicates that two-pass partitioning generally has a higher throughput compared to single-pass partitioning, especially as the radix bits increase.](image-url)
Hash join is $O(N \log N)$!
for all input tuples \( t \) do
\[
\begin{align*}
    h &\leftarrow \text{hash}(t.\text{key}) \\
    \text{copy } t \text{ to } \text{out}[\text{pos}[h]] \\
    \text{pos}[h] &\leftarrow \text{pos}[h] + 1
\end{align*}
\]
end for

Naïve partitioning (cf. slide 78)

Software-Managed Buffers

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

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

Throughput [million tuples/sec] vs. radix bits

- Single-pass partitioning
- Two-pass partitioning
- SW-managed buffers

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
Plugging it together

Blanas et al.: 86.4 / 64.6 cy/tpl

cycles per output tuple

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 2: 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

- 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

- 977 MiB
- e.g., Nehalem: 25 cy/tpl \( \approx \) 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.
Joins and Column-Based Storage

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.