Processing Models

Or how to improve the instruction cache effectiveness?
Processing Models

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

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

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

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

- Operators request tuples from their input using `next()`.
- Data is processed **tuple at a time**.
- Each operator keeps its own state.
Tuple-At-A-Time Processing - Consequences

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

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

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

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

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

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

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

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

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

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

```
SELECT l_returnflag, l_linenstatus, 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_linenstatus
```

• Scan query with arithmetics and a bit of aggregation.

Source: MonetDB/X100: Hyper-Pipelining Query Execution.

[Boncz et al., 2005]
<table>
<thead>
<tr>
<th>time [sec]</th>
<th>calls</th>
<th>instr./call</th>
<th>IPC</th>
<th>function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.9</td>
<td>846M</td>
<td>6</td>
<td>0.64</td>
<td>ut_fold_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>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><strong>2.9</strong></td>
<td>17M</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

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

\(^4\) Depends on underlying hardware
Observations

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

• Much time spent on **field access** (e.g., rec_get_nth_field ()).
  • Polymorphic operators
• Single-tuple functions hard to optimize (by compiler).
  → Low instructions-per-cycle ratio.
  → Vector instructions (SIMD) hardly applicable.
• Function call overhead (e.g., Item_func_plus::val ()).
  • \[
    \frac{38 \text{ instr.}}{0.8 \text{ instr.}} = 48 \text{ cycles vs. } 3 \text{ instr. for load/add/store assembly}^4
  \]

^4 Depends on underlying hardware
Operator-At-A-Time Processing

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

Result

```
Operator 1
  tuples
  Operator 2
    tuples
    Operator 3
      tuples
      ...
Database
```
Operator-At-A-Time Processing

Function call overhead is now replaced by extremely tight loops.

Example: batval_int_add(···)

```c
...
if (vv != int_nil) {
   for (; bp < bq; bp++, bnp++) {
      REGISTER int bv = *bp;
      if (bv != int_nil) {
         bv = (int) OP(bv,+,vv);
      }
      *bnp = bv;
   } else {
      ...
   }
```
Operator-At-A-Time Consequences

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

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

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

- Parallelism: **Inter-operator** and **intra-operator**
- Function call overhead is now replaced by **extremely tight loops** that
  - conveniently **fit into instruction caches**,
  - can be **optimized** effectively by modern compilers
    → **loop unrolling**
    → **vectorization** (use of SIMD instructions)
  - can leverage modern CPU features (**hardware prefetching**).
- Function calls are now **out of the critical code path**.
- **No** per-tuple field extraction or type resolution.
  - **Operator specialization**, *e.g.*, for every possible type.
  - Implemented using **macro expansion**.
  - Possible due to column-based storage.
<table>
<thead>
<tr>
<th>MIL statement</th>
<th>time [ms]</th>
<th>bandwidth [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s0 := select (l_shipdate, ...).mark ();</code></td>
<td>127</td>
<td>352</td>
</tr>
<tr>
<td><code>s1 := join (s0, l_returnag);</code></td>
<td>134</td>
<td>505</td>
</tr>
<tr>
<td><code>s2 := join (s0, l_linenestatus);</code></td>
<td>134</td>
<td>506</td>
</tr>
<tr>
<td><code>s3 := join (s0, l_extprice);</code></td>
<td>235</td>
<td>483</td>
</tr>
<tr>
<td><code>s4 := join (s0, l_discount);</code></td>
<td>233</td>
<td>488</td>
</tr>
<tr>
<td><code>s5 := join (s0, l_tax);</code></td>
<td>232</td>
<td>489</td>
</tr>
<tr>
<td><code>s6 := join (s0, l_quantity);</code></td>
<td>134</td>
<td>507</td>
</tr>
<tr>
<td><code>s7 := group (s1);</code></td>
<td>290</td>
<td>155</td>
</tr>
<tr>
<td><code>s8 := group (s7, s2);</code></td>
<td>329</td>
<td>136</td>
</tr>
<tr>
<td><code>s9 := unique (s8.mirror ());</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>r0 := [+] (1.0, s5);</code></td>
<td>206</td>
<td>440</td>
</tr>
<tr>
<td><code>r1 := [-] (1.0, s4);</code></td>
<td>210</td>
<td>432</td>
</tr>
<tr>
<td><code>r2 := [*] (s3, r1);</code></td>
<td>274</td>
<td>498</td>
</tr>
<tr>
<td><code>r3 := [*] (s12, r0);</code></td>
<td>274</td>
<td>499</td>
</tr>
<tr>
<td><code>r4 := {sum} (r3, s8, s9);</code></td>
<td>165</td>
<td>271</td>
</tr>
<tr>
<td><code>r5 := {sum} (r2, s8, s9);</code></td>
<td>165</td>
<td>271</td>
</tr>
<tr>
<td><code>r6 := {sum} (s3, s8, s9);</code></td>
<td>163</td>
<td>275</td>
</tr>
<tr>
<td><code>r7 := {sum} (s4, s8, s9);</code></td>
<td>163</td>
<td>275</td>
</tr>
<tr>
<td><code>r8 := {sum} (s6, s8, s9);</code></td>
<td>144</td>
<td>151</td>
</tr>
<tr>
<td><code>r9 := {count} (s7, s8, s9);</code></td>
<td>112</td>
<td>196</td>
</tr>
</tbody>
</table>

Source: MonetDB/X100: Hyper-Pipelining Query Execution.

[Boncz et al., 2005]
Tuple-At-A-Time vs. Operator-At-A-Time

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

😊 Cache-efficient with respect to **code** and **operator state**.

😊 Tight loops, optimizable code.
Tuple-At-A-Time vs. Operator-At-A-Time

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

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

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

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

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

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

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

tuple-at-a-time \(\leftrightarrow\) vectorized execution \(\uparrow\) operator-at-a-time
Vectorized Execution Model

Idea:

- Use Volcano-style iteration,
  but:
  - for each `next()` call **return a large number of tuples**
    → a so called “vector”
Vectorized Execution Model

Idea:

• Use Volcano-style iteration,

but:

• for each next () call return a large number of tuples
  → a so called “vector”

Choose vector size

• large enough to compensate for iteration overhead (function calls, instruction cache misses, . . . ), but

• small enough to not thrash data caches.
5.1.3 Processing unit size

Comparing to the tuple-at-a-time and column-at-a-time models, the vectorized model provides a granularity of operation that falls between these two extremes. As a result, there are situations in which some logic that is usually executed for every tuple, can be executed on a per-vector base. A simple example is data partitioning, when the result partition sizes are not known in advance. The code for dividing a vector of N tuples into P partitions using the hash values could be as follows:

```c
for (i = 0; i < N; i++) {
    group = hash_values[i] % P;
    *(part[group]++) = values[i];
    if (part[group] == part_end[group])
        overflow(group);
}
```

Note that the overflow check is necessary for each tuple if we do not know the partition sizes in advance. While this check is usually false, we can still remove it from the loop, by exploiting the fact that in most cases the buffers for the destination groups are much larger than the size of the vector. As a result, we can check if every group buffer still contains enough tuples before processing each vector.

```c
for (i = 0; i < P; i++)
    if (part[i] >= part_sentinel[i])
```

Source: [Zukowski, 2009]

- Vectorized execution quickly compensates for iteration overhead.
- 1000 tuples should conveniently fit into caches.
## Comparison of Execution Models

<table>
<thead>
<tr>
<th>execution model</th>
<th>tuple</th>
<th>operator</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr. cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>function calls</td>
<td>many</td>
<td>extremely few</td>
<td>very few</td>
</tr>
<tr>
<td>attribute access</td>
<td>complex</td>
<td>direct</td>
<td>direct</td>
</tr>
<tr>
<td>most time spent on</td>
<td>interpretation</td>
<td>processing</td>
<td>processing</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>poor</td>
<td>good</td>
<td>very good</td>
</tr>
<tr>
<td>compiler optimizations</td>
<td>limited</td>
<td>applicable</td>
<td>applicable</td>
</tr>
<tr>
<td>materialization overhead</td>
<td>very cheap</td>
<td>expensive</td>
<td>cheap</td>
</tr>
<tr>
<td>scalability</td>
<td>good</td>
<td>limited</td>
<td>good</td>
</tr>
</tbody>
</table>

Source [Zukowski, 2009]
Storage Models

Or how to improve the data cache effectiveness?
Data Storage Approaches

- Row-stores
- Column-stores
Row-stores

a.k.a. row-wise storage or \( n \)-ary storage model, NSM:

![Diagram showing row-stores on pages 0 and 1]
Column-stores

a.k.a. column-wise storage or decomposition storage model, DSM:
The effect on query processing

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**.
A full table scan in a row-store

In a row-store, all \textbf{rows} of a table are stored sequentially on a database page.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{tuple_diagram}
\caption{Tuple representation in a row-store}
\end{figure}

Sebastian Dorok
Main-Memory Database Management Systems
Last Change: May 11, 2015
55/116
A full table scan in a row-store

In a row-store, all **rows** of a table are stored sequentially on a database page.

```
<table>
<thead>
<tr>
<th>l_shipdate</th>
<th>tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

A full table scan in a row-store

In a row-store, all rows of a table are stored sequentially on a database page.

With every access to a l_shipdate field, we load a large amount of irrelevant information into the cache.
A "full table scan" on a column-store

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

l_shipdate(s)
A "full table scan" on a column-store

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

l_shipdate(s)

All data loaded into caches by a “l_shipdate scan” is now actually relevant for the query.
Column-store advantages

- All data loaded into caches by a “l_shipdate scan” is now actually relevant for the query.
  - Less data has to be fetched from memory.
  - Amortize cost for fetch over more tuples.
  - If we’re really lucky, the full (l_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.
Column-store trade-offs

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

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

MonetDB makes this explicit in its data model.

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

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

MonetDB makes this explicit in its data model.

• **All** tables in MonetDB have two columns (“head” and “tail”).

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

• Each column yields one **binary association table (BAT)**.

• **Object identifiers** ($oids$) identify matching entries (BUNs).
An example: Binary Association Tables in MonetDB

MonetDB makes this explicit in its data model.

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

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

<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>

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

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