Data Processing on Modern Hardware

Jana Giceva

Lecture 3: Cache awareness
for query execution models
Cache awareness for query execution models
The processing model of a database defines *how the system executes the query plan.*

The four main approaches are:

- **Iterator** model (volcano, tuple-at-a-time)
- **Materialization** model (operator-at-a-time, column-at-a-time)
- **Vectorization** model (vector-at-a-time, batch, block-wise)
- **Pushing tuples up** model

There are different trade-offs depending on the workload type and the underlying hardware.

→ cf. Database Systems on Modern CPU Architectures (chapter 5)
Most classical systems implement the **Volcano iterator model**: 

- Operators request tuples from their input using `next()` 
  - On each invocation, the operator returns either a single tuple or `null` if there are no more tuples 

- Data is processed **tuple-at-a-time** in a **pipelined** fashion 
  - Also called the Volcano or pipeline model 

- Each operator keeps its own **state**
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100

for t in child.Next():
    emit(projection(t))

for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2):
        emit(t1 & t2)

for t in child.Next():
    if evalPred(t):
        emit(t)
else:
    emit(null)

if R.hasNext():
    emit(R.next())
else:
    emit(null)

if S.hasNext():
    emit(S.next())
else:
    emit(null)
**Iterator model – Example**

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Diagram:

```
\[ \pi \text{ R.id, S.value} \]
\[ \\Join \text{ R.id = S.id} \]
\[ \sigma \text{ value > 100} \]
```

1. for \( t \) in child.Next():
   - emit(projection(t))

2. for \( t_1 \) in left.Next():
   - buildHashTable(t_1)
   - for \( t_2 \) in right.Next():
     - if probe(t_2): emit(t_1 ⊙ t_2)

3. if R.hasNext():
   - emit(R.next())
   - else emit(null)

Single tuple

if S.hasNext():
   - (S.next())
   - else emit(null)
**Iterator model – Example**

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

---

1. For `t` in `child.Next()`:
   - `emit(projection(t))`

2. For `t_1` in `left.Next()`:
   - `buildHashTable(t_1)`
   - For `t_2` in `right.Next()`:
     - If `probe(t_2)`, `emit(t_1 \& t_2)`

3. If `R.hasNext()`:
   - `emit(R.next())`
   - If not, `emit(null)`

4. For `t` in `child.Next()`:
   - If `evalPred(t)`, `emit(t)`

5. If `S.hasNext()`:
   - `emit(S.next())`
   - If not, `emit(null)`
This is used in almost every RDBMS.
- Allows for tuple pipelining.
- Some operators must block until their children emit all their tuples:
  - Joins, subqueries, sort, group-by, etc.

Implications on cache usage efficiency:
- All operators in a plan run tightly interleaved
  - Their combined instruction footprint may be large
  - Many instruction cache misses
- Operators constantly call each other’s functionality
  - Results in a big function call overhead
- The combined state of the operators may be too large to fit into caches
  - e.g., hash tables, cursors, partial aggregates
  - Results in many data cache misses
Example: TPC-H on MySQL

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 on aggregated tuples without a join

Results taken from MonetDB/X100: Hyper-Pipelining Query Execution *CIDR 2005*
Show results from executing the query

<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_ulongint_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>3.1</td>
<td>23M</td>
<td>64</td>
<td>0.88</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>0.70</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>1.3</td>
<td>17M</td>
<td>38</td>
<td>0.80</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>1.0</td>
<td>11M</td>
<td>38</td>
<td>0.80</td>
<td>Item_func_minus::val</td>
</tr>
<tr>
<td>0.5</td>
<td>5.9M</td>
<td>38</td>
<td>0.80</td>
<td>Item_func_plus::val</td>
</tr>
</tbody>
</table>

Each call only processes a **single tuple** → **millions of calls**

Only **10% of the time** spent on actual query task.

Very low **instructions-per-cycle** (IPC) ratio.
Further observations

Much time spent on field access (e.g., `rec_get_nth_field()`).
- Row-store $\rightarrow$ polymorphic operators.

**Single-tuple functions are hard to optimize (by compiler):**
- Low IPC ratio – empty pipelines make the CPU stall
- Optimization across functions not possible (or very difficult)
- Function call overhead is high
- Vector instructions (SIMD) are hardly applicable

**Example:**
- Let’s consider the `Item_func_plus::val` function from the previous table
- $\frac{38 \text{ instr.}}{0.8 \text{ instr./cycle}} = 48$ cycles vs. 3 instructions for load/add/store assembly
- One explanation for this high cost is the absence of *loop pipelining*, dependent instructions $\rightarrow$ 20 cycles
- High cost of a function (routine) call ($\sim$ 20 cycles) that cannot be amortized
Materialization model

Each operator processes its input all at once and then stores its output all at once (in one buffer)

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

The output can be either a whole tuple (row-store) or subsets of columns (column-store).
Materialization model – Example

\[
\begin{array}{c}
\text{SELECT R.id, S.cdate} \\
\text{FROM R JOIN S} \\
\text{ON R.id = S.id} \\
\text{WHERE S.value > 100}
\end{array}
\]

\[
\begin{array}{c}
\pi R.id, S.value \\
\shuffle R.id = S.id \\
\sigma \text{ value > 100}
\end{array}
\]

1. \[
\text{out} = [] \\
\text{for } t \text{ in } \text{child}.\text{Output}(): \\
\quad \text{out}.\text{append}(\text{projection}(t)) \\
\text{return out}
\]

2. \[
\text{out} = [] \\
\text{for } t_1 \text{ in } \text{left}.\text{Output}(): \\
\quad \text{buildHashTable}(t_1) \\
\text{for } t_2 \text{ in } \text{right}.\text{Output}(): \\
\quad \text{if probe}(t_2): \text{out}.\text{append}(t_1 \land t_2) \\
\text{return out}
\]

3. \[
\text{out} = [] \\
\text{for } t \text{ in } R \\
\quad \text{out}.\text{append}(t) \\
\text{return out}
\]

\[
\begin{array}{c}
\text{out} = [] \\
\text{for } t \text{ in S} \\
\quad \text{out}.\text{append}(t) \\
\text{return out}
\end{array}
\]

All tuples
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100

\[ \pi \]
\[ \bowtie \]
\[ \sigma \]

\( R \)
\( S \)

1. \[ out = [] \]
   \[ for t in child.Output(): \]
   \[ out.append(projection(t)) \]
   \[ return out \]

2. \[ out = [] \]
   \[ for t_1 in left.Output(): \]
   \[ buildHashTable(t_1) \]
   \[ for t_2 in right.Output(): \]
   \[ if probe(t_2): out.append(t_1 \bowtie t_2) \]
   \[ return out \]

3. \[ out = [] \]
   \[ for t in R \]
   \[ out.append(t) \]
   \[ return out \]

4. \[ out = [] \]
   \[ for t in child.Output(): \]
   \[ if evalPred(t): out.append(t) \]
   \[ return out \]

5. \[ out = [] \]
   \[ for t in S \]
   \[ out.append(t) \]
   \[ return out \]
Materialization model – Example

SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100

\[
\pi \begin{pmatrix} R.id, S.value \end{pmatrix} \Join \sigma \text{value} > 100
\]

- **1**: 
  ```python
  out = []
  for t in child.Output():
      out.append(projection(t))
  return out
  ```

- **2**: 
  ```python
  out = []
  for t1 in left.Output():
      buildHashTable(t1)
  for t2 in right.Output():
      if probe(t2): out.append(t1 \Join t2)
  return out
  ```

- **3**: 
  ```python
  out = []
  for t in R
      out.append(t)
  return out
  ```

- **4**: 
  ```python
  out = []
  for t in child.Output():
      if evalPred(t): out.append(t)
  return out
  ```

- **5**: 
  ```python
  out = []
  for t in S
      out.append(t)
  return out
  ```
Materialization model – analysis

Much fewer number of function calls

Due to such operator-at-a-time processing, its 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*)
- Far less expensive function calls are now *out of the critical code path*
The materialization (operator-at-a-time) model is a two-edged sword:
- Cache-efficient with respect to code and operator state
- Tight loops, optimizable code

But, each operator reads in and out everything, so
- Data won’t fully fit into the cache:
  - Repeated scans will fetch data from memory over and over
  - Strategy falls apart when intermediate (materialized) results no longer fit in memory/caches

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

**Iterator vs. Materialization model**

- tuple-at-a-time (iterator model)
- vector-at-a-time (vectorization model)
- operator-at-a-time (materialization model)
**Vectorization model**

**Idea:** use volcano-style iteration

**But** for each `next()` call return a *batch of tuples* instead of a single tuple

- Vector in MonetDB/X100 terminology
- The operator’s internal loop processes multiple tuples at a time
- The size of the batch can vary based on the hardware and query properties
Vectorization model – Example

\[
\begin{align*}
\pi & \quad R.id, S.value \\
\bowtie & \quad R.id = S.id \\
\sigma & \quad \text{value} > 100
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad R.id, S.cdate \\
\text{FROM} & \quad R \ JOIN S \\
\text{ON} & \quad R.id = S.id \\
\text{WHERE} & \quad S.value > 100
\end{align*}
\]

1. \[\text{out} = []\]
   \[\text{for } t \text{ in } \text{child.Output}():\]
   \[\phantom{\text{out}} = \text{out.append(projection(t))}\]
   \[\text{if } |\text{out}| > \text{n}: \text{emit(out)}\]

2. \[\text{out} = []\]
   \[\text{for } t_1 \text{ in } \text{left.Output}():\]
   \[\phantom{\text{out}} = \text{buildHashTable(t_1)}\]
   \[\text{for } t_2 \text{ in } \text{right.Output}():\]
   \[\phantom{\text{out}} = \text{if probe(t_2): out.append(t_1 \bowtie t_2)}\]
   \[\text{if } |\text{out}| > \text{n}: \text{emit(out)}\]

3. \[\text{out} = []\]
   \[\text{while } \text{R.hasNext()} \&\& |\text{out}| < \text{n}\]
   \[\phantom{\text{out}} = \text{out.append(R.next())}\]
   \[\text{emit(out)}\]

\[\text{out} = []\]
\[\text{for } t \text{ in } \text{child.Output}():\]
\[\phantom{\text{out}} = \text{if evalPred(t): out.append(t)}\]
\[\text{if } |\text{out}| > \text{n}: \text{emit(out)}\]

\[\text{out} = []\]
\[\text{while } \text{S.hasNext()} \&\& |\text{out}| < \text{n}\]
\[\phantom{\text{out}} = \text{out.append(S.next())}\]
\[\text{emit(out)}\]
Vectorization model – Example

\[
\begin{align*}
\text{SELECT} & \quad \text{R.id, S.cdate} \\
\text{FROM} & \quad \text{R JOIN S} \\
\text{ON} & \quad \text{R.id = S.id} \\
\text{WHERE} & \quad \text{S.value > 100}
\end{align*}
\]

\[
\begin{align*}
\pi \quad & \text{R.id, S.value} \\
\bowtie \quad & \text{R.id = S.id} \\
\sigma \quad & \text{value > 100}
\end{align*}
\]

1. `out = []`  
   `for t in child.Output():`  
   `out.append(projection(t))`  
   `if |out| > n: emit(out)`

2. `out = []`  
   `for t1 in left.Output():`  
   `buildHashTable(t1)`  
   `for t2 in right.Output():`  
   `if probe(t2): out.append(t1ψt2)`  
   `if |out| > n: emit(out)`

3. `out = []`  
   `while R.hasNext() & |out| < n`  
   `out.append(R.next())`  
   `emit(out)`

4. `out = []`  
   `for t in child.Output():`  
   `if evalPred(t): out.append(t)`  
   `if |out| > n: emit(out)`

5. `out = []`  
   `while S.hasNext() & |out| < n`  
   `out.append(S.next())`  
   `emit(out)`
Vectorization model – analysis

Uses the best of both worlds (iterator and materialization models):
- Reduces the number of invocations per operator
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples

Imperative to choose a vector size that is:
- Large enough to amortize the iteration overhead (e.g., function calls, instruction cache misses, etc),
- 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?
Observations:

- Vectorized execution quickly compensates for iteration overhead
- 1000 tuples should conveniently fit into caches
Vectorized execution in MonetDB/X100

Source: M. Zukowski, Balancing Vectorized Query Execution with Bandwidth Optimized Storage, PhD thesis, CWI Amsterdam, 2009
Microsoft SQL Server supports vectorized ("batched" in MS jargon) execution since version 11.

- Storage via new **column-wise index** (with compression and prefetching improvements)

- New operators with **batch-at-a-time processing**

- 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, the optimizer treats batch processing as new **physical property** (like being sorted) to combine operators in a proper way.
SQL Server: Performance

Performance impact (TPC-DS, scale factor 100, ~ 100GB)

Per-Ake Larson et al. SQL Server Column Store Indexes. SIGMOD 2011
Vectorized execution in PostgreSQL

- Organize query plan into **execution groups**
- Add **buffer operator** between execution groups
- The buffer operator provides tuple-at-a-time interface to the outside, but **batches up** tuples internally.
- Similar to the example we covered previously

```c
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 next tuple in buffer;
```
Buffer operators in PostgreSQL

## Comparison of processing models

Overview of the discussed execution models

<table>
<thead>
<tr>
<th>Execution model</th>
<th>iterator (tuple)</th>
<th>materialization (operator)</th>
<th>vectorization (vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>query plans</td>
<td>simple</td>
<td>complex</td>
<td>simple</td>
</tr>
<tr>
<td>instruction 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>

*src: M. Zukowski, Balancing Vectorized Query Execution with Bandwidth Optimized Storage, PhD thesis, CWI Amsterdam, 2009*
References

- Various papers cross-referenced in the slides:
  - Boncz et al. *MonetDB/X100: Hyper-Pipelining Query Execution* CIDR 2005
  - Larson et al. *SQL Server Column Store Indexes*. SIGMOD 2011
  - Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask*. VLDB 2018

- Lecture: *Database Systems on Modern CPU Architectures* by Prof. Thomas Neumann (TUM)
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Advanced Databases* by Prof. Andy Pavlo (CMU)

- Check out the code from Timo Kersten and play around with the TPC-H queries from Typer and Tectorwise (TW):
  - [https://github.com/TimoKersten/db-engine-paradigms](https://github.com/TimoKersten/db-engine-paradigms)