HyPer-sonic Combined Transaction AND Query Processing

Thomas Neumann

Technische Universität München

December 2, 2011
Motivation

There are different scenarios for database usage:

**OLTP**: Online Transaction Processing
- customers order products, customers make phone calls, etc.
- basically book-keeping, modifies the database
- very high transaction rates, thousands per second

**OLAP**: Online Analytical Processing
- what are the top products, where is the most traffic, etc.
- analytical queries, aggregate large amounts of data
- long running, take seconds or even minutes

Different kinds of requirements
Motivation - OLTP vs. OLAP

OLTP and OLAP have very different requirements

- **OLTP**
  - high rate of small/tiny transactions
  - high locality in data access
  - update performance is critical

- **OLAP**
  - few, but long running transactions
  - aggregates large parts of the database
  - must see a consistent database state the whole time

Traditionally, DBMSs either good at OLTP or good at OLAP
Motivation - Traditional Solution

ETL Extract Transform Load
OLTP Requests /Tx
OLAP Queries

not very satisfying. stale data, redundancy, etc.
Motivation - Hardware Trends

Intel
Tera Scale Initiative
Server with 1 TB main memory
c. 40K Euro from Dell

- main memory grows faster than (business) data
- can afford to keep data in memory
- memory is not just a fast disk
- should make use of this facts

Amazon

Data Volume
Revenue: 25 billion Euro
Avg. Item Price: 15 Euro
c. 1.6 billion order lines per year
c. 54 Bytes per order line
c. 90 GB per year
+ additional data - compression

Transaction Rate
Avg: 32 orders per s
Peak rate: Thousands/s
+ inquiries
HyPer

Our system

Combined OLTP/OLAP system using modern hardware
HyPer - Design

- OLTP performance is crucial
- avoid anything that would slow down OLTP
- OLTP should operate as if there were no OLAP
- OLAP is not that performance sensitive, but needs consistency
- locking/latching is out of question (OLAP would slow down OLTP)

Idea: we are a main memory database. Use hardware support.
HyPer - Pure OLTP workload

- purely main memory, OLTP transactions need a few $\mu$s
- can afford serial execution of transactions (at least initially)
- avoids any concurrency issues
• OLAP sessions need a consistent snapshot over a relatively long time
• use the MMU / OS support to separate OLTP and OLAP
• the *fork* separates OLTP from OLAP, even though they are initially the same
HyPer - Copy on Update

- the MMU detects writes to shared data
- modified pages are copied, both parts have unique copies afterwards
- avoids any interaction between OLTP and OLAP
- like an ultra-efficient shadow paging without the disadvantages
HyPer - Snapshots

We use *fork* to create transaction consistent snapshots

- each OLAP sessions sees one certain point in time
- can do long-running aggregates/analysis
- the data (apparently) stays the same
- if it changes, the MMU makes sure that OLAP does not notice
- eliminates need for latching/locking

And *fork* is cheap!

- only the page table is copied, not the pages themselves
- some care is needed to scale to large memory sizes
- but can *fork* 40GB in 2.7ms
HyPer - Using the Cores

• we allow parallelism if we know transactions operate on separate data
• requires data flow analysis, serialize if not sure
• allows for utilizing more than one core on the OLTP side
• multiple OLAP sessions, each copies just what is needed
• logging is needed for ACID properties
• backups for fast restart
Query Processing

Most DBMS offer a *declarative* query interface

- the user specifies the only desired result
- the exact evaluation mechanism is up the the DBMS
- for relational DBMS: SQL

For execution, the DBMS needs a more imperative representation

- usually some variant of relational algebra
- describes the the real execution steps
- set oriented, but otherwise quite imperative
Query Processing (2)

Example translation into relational algebra:

**SQL**

```sql
select *
from R1,R3,
    (select R2.z,count(*)
    from R2
    where R2.y=3
    group by R2.z) R2
where R1.x=7
    and R1.a=R3.b
    and R2.z=R3.c
```

**Execution Plan**

- algebraic expression describes execution strategy
- physical algebra contains more information omitted here (access path, join algorithms etc.)
Query Processing (3)

How to evaluate such an execution plan?

- the algebraic expression describes the intended evaluation strategy
- but it is not directly executable
- before executing, most DBMS perform code generation

What “code generation” means differs between systems

- some simply annotate the algebraic tree, and then interpret it
- some generate bytecode for a VM
- and some really generate code
- e.g., System R generated machine code (but had portability issues)

What is the best evaluation strategy on modern machines?
Iterator Model

The classical evaluation strategy is the **iterator model** (sometimes called Volcano Model, but actually much older [Lorie 74])

- each algebraic operator produces a *tuple stream*
- a consumer can *iterate* over its input streams
- interface: open/next/close
- each *next* call produces a new tuple
- all operators offer the same interface, implementation is opaque
Iterator Model (2)

Example:

\[ \begin{align*}
\sigma_x &= 7 \\
\quad & \downarrow 1. \text{next} \\
\sigma_y &= 3 \\
\end{align*} \]
Iterator Model (2)

Example:

\[
\begin{align*}
\sigma_{x=7} & \quad \sigma_{y=3} \\
R_1 & \quad R_2 \\
\bigwedge_{a=b} & \quad \bigwedge_{z=c} \\
1. \text{ next} & \quad 2. \text{ next}
\end{align*}
\]
Iterator Model (2)

Example:

\[ \sigma_{x=7} \]

1. next

2. next

3. next

\[ \Gamma_{z; count(*)} \]

\[ \sigma_{y=3} \]

\[ R_1 \]

\[ R_2 \]

\[ R_3 \]
Iterator Model (2)

Example:

\[
\begin{align*}
\sigma_{x=7} & \\
\times_{a=b} & \\
\sigma_{y=3} & \\
\times_{z=c} & \\
\end{align*}
\]

R₁

R₂

R₃
Iterator Model (2)

Example:

\[ a = b \]

\[ \sigma_{x=7} \]

\[ \sigma_{y=3} \]

\[ \Gamma_{z;\text{count}(\ast)} \]

\[ R_1 \]

\[ R_2 \]

\[ R_3 \]
Iterator Model (2)

Example:

\begin{align*}
\sigma_x &= 7 \\
R_1 &\leftarrow \text{tuple} \\
R_2 &\quad \sigma_y &= 3 \\
R_3 &\quad z; \text{count}(*)
\end{align*}
Iterator Model (2)

Example:

\[ \text{tuple} \]

\[ \sigma_{x=7} \]

\[ \sigma_{y=3} \]

\[ \Gamma_{z; \text{count}(\ast)} \]

\[ R_1 \]

\[ R_2 \]

\[ R_3 \]

\[ a=b \]

\[ z=c \]
Iterator Model (2)

Example:

\[
\begin{align*}
\sigma_{x=7} & \implies 1. \text{ next} \\
\Join_{a=b} & \implies 1000. \text{ next} \\
\sigma_{y=3} & \\
\Join_{z=c} & \\
\Gamma_{z;\text{count}(*)} & \\
\end{align*}
\]
Iterator Model (2)

Example:

\[ \sigma_{x=7} \]

1. next

\[ \forall_{a=b} \]

1000. next

\[ \forall_{z=c} \]

1001. next

\[ \Gamma_{z;\text{count}(\ast)} \]

\[ \sigma_{y=3} \]

\[ R_1 \]

\[ R_2 \]

\[ R_3 \]
Iterator Model (2)

Example:

\[ \sigma_{x=7} \]

\[ \Gamma_{z; \text{count}(\ast)} \]

\[ \sigma_{y=3} \]

\[ 1. \text{next} \]

\[ 1001. \text{next} \]

\[ R_1 \]

\[ R_2 \]

\[ R_3 \]

\[ \times_{a=b} \]

\[ \times_{z=c} \]

\[ \text{etc.} \]
Data-Centric Query Execution

HyPer does not use the classical iterator model

Why does the iterator model (and its variants) use the operator structure for execution?

- it is convenient, and feels natural
- the operator structure is there anyway
- but otherwise the operators only describe the data flow
- in particular operator boundaries are somewhat arbitrary

What we really want is **data centric** query execution

- data should be read/written as rarely as possible
- data should be kept in CPU registers as much as possible
- the code should center around the data, not the data move according to the code
- increase locality, reduce branching
Data-Centric Query Execution (2)
Processing is oriented along pipeline fragments.

Corresponding code fragments:

- initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and $\Gamma_z$
- for each tuple $t$ in $R_1$
  - if $t.x = 7$
    - materialize $t$ in hash table of $\bowtie_{a=b}$
- for each tuple $t$ in $R_2$
  - if $t.y = 3$
    - aggregate $t$ in hash table of $\Gamma_z$
- for each tuple $t$ in $\Gamma_z$
  - materialize $t$ in hash table of $\bowtie_{z=c}$
- for each tuple $t_3$ in $R_3$
  - for each match $t_2$ in $\bowtie_{z=c}[t_3.c]$
    - for each match $t_1$ in $\bowtie_{a=b}[t_3.b]$
      - output $t_1 \circ t_2 \circ t_3$
Data-Centric Query Execution (3)

The algebraic expression is translated into query fragments.

Each operator has two interfaces:

1. produce
   - asks the operator to produce tuples and push it into

2. consume
   - which accepts the tuple and pushes it further up

Note: only a mental model!

- the functions are not really called
- they only exist conceptually during code generation
- each “call” generates the corresponding code
- operator boundaries are blurred, code centers around data
- we generate machine code at compile time
- initially using C++, now using LLVM
Evaluation

We used a combined TPC-C and TPC-H benchmark (12 warehouses)

- TPC-C transactions are unmodified
- TPC-H queries adapted to the combined schema
- OLTP and OLAP runs in parallel
## TPC-C+H Performance

<table>
<thead>
<tr>
<th>Query No.</th>
<th>1 query session (stream)</th>
<th>3 query sessions (streams)</th>
<th>MonetDB no OLTP</th>
<th>VoltDB no OLAP only OLTP results from VoltDB web page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>one query session (stream)</td>
<td>3 query sessions (streams)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OLTP</td>
<td>OLTP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>single threaded OLTP</td>
<td>5 OLTP threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OLTP throughput</td>
<td>OLTP throughput</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Query resp. times (ms)</td>
<td>Query resp. times (ms)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>67</td>
<td>71</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>163</td>
<td>212</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>66</td>
<td>73</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>194</td>
<td>226</td>
<td>6003</td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>1276</td>
<td>1564</td>
<td>5930</td>
<td></td>
</tr>
<tr>
<td>Q6</td>
<td>9</td>
<td>17</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>Q7</td>
<td>1151</td>
<td>1466</td>
<td>1713</td>
<td></td>
</tr>
<tr>
<td>Q8</td>
<td>399</td>
<td>593</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>Q9</td>
<td>206</td>
<td>249</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>Q10</td>
<td>1871</td>
<td>2260</td>
<td>6209</td>
<td></td>
</tr>
<tr>
<td>Q11</td>
<td>33</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Q12</td>
<td>156</td>
<td>170</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>Q13</td>
<td>185</td>
<td>229</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>Q14</td>
<td>122</td>
<td>156</td>
<td>722</td>
<td></td>
</tr>
<tr>
<td>Q15</td>
<td>528</td>
<td>792</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td>Q16</td>
<td>1353</td>
<td>1500</td>
<td>3562</td>
<td></td>
</tr>
<tr>
<td>Q17</td>
<td>159</td>
<td>168</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>Q18</td>
<td>108</td>
<td>119</td>
<td>2505</td>
<td></td>
</tr>
<tr>
<td>Q19</td>
<td>103</td>
<td>183</td>
<td>1698</td>
<td></td>
</tr>
<tr>
<td>Q20</td>
<td>114</td>
<td>197</td>
<td>750</td>
<td></td>
</tr>
<tr>
<td>Q21</td>
<td>46</td>
<td>50</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>Q22</td>
<td>7</td>
<td>9</td>
<td>141</td>
<td></td>
</tr>
</tbody>
</table>

Dual Intel X5570 Quad-Core-CPU, 64GB RAM, RHEL 5.4
- we only have to replicate the working set
Conclusion

• main memory databases change the game
• very high throughput, transactions should never wait
• minimize latching and locks to get best performance
• use MMU support instead to separate OLTP and OLAP
• compiled, data-centric queries for excellent performance

HyPer is a very fast hybrid OLTP/OLAP system
• top performance for both OLTP and OLAP
• full ACID support

It is indeed possible to build a combined OLTP/OLAP system!