An Adaptive Query Execution Engine for Data Integration

Zachary Ives, Daniela Florescu, Marc Friedman, Alon Levy, Daniel S. Weld

University of Washington

Slides by Peng Li, Modified by Rachel Pottinger,
Modified by April Webster

Outline
- Motivation for Tukwila
- Tukwila Architecture
- How is Tukwila Adaptive?
- Interleaved Optimization and Execution
- Adaptive Query Operators: dynamic collectors
  & double pipelined hash join

The Motivation for Tukwila:
Key characteristics of the data integration problem:
- No statistics
- Unpredictable data transfer rates
- Overlap & redundancy of data
- Quick initial results more important than total plan cost

Conclusion: Adaptivity is critical to performance!
Traditional static query processing inadequate.

Discussion #1
- Why does optimizing initial answers matter more in data integration?
- Can you imagine needing it elsewhere?

Tukwila Architecture

How is Tukwila adaptive?
- Between the optimizer & the execution engine via "Interleaved planning & execution"
  - Compensates for lack of information and unpredictable data transfer rates
- Within the execution engine via "Adaptive query operators"
  - Manage overlapping data sources (dynamic collectors)
  - Produce initial results quickly (double pipelined hash join)
Interleaved Optimization & Execution

Query plans can be reoptimized or rescheduled by the query optimizer after the execution of each fragment by the query execution engine based on the event-condition-action rules.

The Optimizer

Novel characteristics of Tukwila’s optimizer:
- The optimizer generates a query plan and a set of rules to define adaptive behaviour
- The query plan need not be complete - if essential statistics are missing or uncertain a partial query plan can be produced
- The optimizer conserves the state of its search space when it calls the execution engine; can resume optimization incrementally

The Query Plan

- **Query plan** = a partially-ordered set of fragments + a set of rules
- **Fragment** = a fully pipelined tree of physical operators
- **Rules** = describe when and how to modify the implementation of certain operators at runtime if needed; detect opportunities for re-optimization.

The fragment is the source of adaptivity
  - At the end of each fragment, the rest of the plan can be re-optimized or rescheduled according to the rules

The Rules

Implement one of four "adaptive" behaviours:
1. **Re-optimization**: if the optimizer's cardinality estimate for the fragment’s result is significantly different from the actual size → re-involve optimizer
2. **Contingent planning**: the execution engine checks properties of the result to select the next fragment
3. **Rescheduling**: if a source times out
4. **Adaptive operators**: overflow resolution for the double pipelined join; collector implementation

Query Execution

The query plan (represented as an operator tree) is executed using the top-down “iterator” model:

```
    Orders
      ↓
    Join
      ↓
    Select
      ↓
    Read
```

```
“Show which orders have been delivered”
```

- **Control flow**
  - Iterator (top-down)
    - Most common database model
    - Control flows from the node down to the leaves within each fragment

Query Execution

- Multiple fragments; end at materialization points
- Execution engine generates events when execution state changes (e.g., fragment completes)
- Events trigger rules
  - E.g., Re-optimize remainder (terminate current plan & reinvoke optimizer, sending back statistics)
Dynamic Collector

- Dynamic collectors provide a policy for guiding access to overlapping data sources
  - Provided by the optimizer based on estimates of the "overlap relationships" between sources
    - Data source access order
    - Potential fallback sources
  - Implemented by query execution engine by contacting data sources in parallel
    - Is flexible – can contact only some of the sources

Double Pipelined Hash Join

- Main features:
  - Each source relation has its own hash table in memory
  - As a tuple comes in, add to its own hash table and probe opposite hash table
- Benefits:
  - Produces results as soon as a tuple is received; time to first output tuple minimized!
  - Symmetric – no "inner" relation to wait for (as in nested loop joins & hash joins)
- Drawbacks:
  - Requires memory for two has tables
  - Data-driven, bottom-up execution model (but execution is top-down)

Example

<table>
<thead>
<tr>
<th>Orders</th>
<th>TrackNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>01-23-45</td>
</tr>
<tr>
<td>1235</td>
<td>02-90-85</td>
</tr>
<tr>
<td>1399</td>
<td>02-90-85</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UPS</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrackNo</td>
<td></td>
</tr>
<tr>
<td>01-23-45</td>
<td>In Transit</td>
</tr>
<tr>
<td>02-90-85</td>
<td>Delivered</td>
</tr>
<tr>
<td>03-99-10</td>
<td>Delivered</td>
</tr>
</tbody>
</table>

Double-Pipelined Hash Join: Adapting to the Iterator Model

- Use multiple threads with queues
  - Each child (A or B) reads tuples until full, then sleeps & awakens parent
  - Join sleeps until awakened, then:
    - Joins tuples from QA or QB, returning all matches as output
    - Wakes owner of queue

Double-Pipelined Hash Join: Handling Memory Overflow

- May not be able to fit hash tables in memory
  - Only feasible recovery strategy is flush portion of hash table to disk when system runs out of memory
- Two algorithms implemented in Tukwila:
  - Incremental Left Flush - read only from right, flush from left
  - Incremental Symmetric Flush - choose a bucket and flush from both sources

Conclusions

Tukwila’s main contributions:

- Tukwila achieves adaptivity by segmenting a query plan into fragments and interleaving the execution of these with reoptimization
- Tukwila minimizes the time to the first output tuple through its use of the double pipelined hash join
Discussion #2

- How do you evaluate the double pipelined hash join?
- Is it efficient?
- Would you use it if you were not doing data integration, why or why not?