Traditional Query Optimization

Static optimization

- Iterate over series of plans to find best
- Execute plan until finished
- No opportunity to change plan once started
Problems With Traditional Query Optimization

Run-time fluctuations
- Wide-area data sources
- Long-running queries
- Correlations between predicates and order of tuples
- Variation in available resources (cloud computing)
## Alternatives To Traditional Query Optimization

**Time-based re-planning**
- Reevaluate plan every x minutes

**Pipeline-based re-planning**
- Execute one node of the plan tree, then reevaluate remaining nodes

**Tuple-based re-planning**
- Construct plan per-tuple
- Static vs. dynamic planning

**Eddies**
- Tuple-based dynamic planning
Adapting the Join Order using an Eddy

Query processing as routing of tuples through operators

A traditional query plan

\[
\text{R} \rightarrow \text{NLJ}_A \rightarrow \text{NLJ}_B \rightarrow \text{NLJ}_C
\]

(c₁, s₁) → (c₂, s₂) → (c₃, s₃)

Pipelined query execution using an eddy

An eddy operator
- Intercepts tuples from sources and output tuples from operators
- Executes query by routing source tuples through operators

Encapsulates all aspects of adaptivity in a “standard” dataflow operator: measure, model, plan and actuate.
Synchronization Barriers

Point at which one data source has to wait for another

– slowlow/fasthi join example
Moments of Symmetry

Point at which order of input can be changed
- Nested loops join
- Symmetric hash join
An R Tuple: \( \mathbf{r_1} \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
</tr>
</tbody>
</table>
An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

**ready** bit \( i \):

1 → operator \( i \) can be applied
0 → operator \( i \) can’t be applied
Eddies [AH’00]

An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

done bit i:
- 1 → operator i has been applied
- 0 → operator i hasn’t been applied

[Diagram of Eddy with operators and NLJ nodes]
Eddies [AH’00]

An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

Used to decide validity and need of applying operators.

Eddy

Operator 1

Operator 2

Operator 3

result

R

\( r_1 \)
Eddies [AH’00]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>AnameA</td>
<td>...</td>
<td>000</td>
<td>111</td>
</tr>
</tbody>
</table>

if done = 111, send to output
Adapting order is easy
- Just change the operators to which tuples are sent
- Can be done on a per-tuple basis
- Can be done in the middle of tuple’s “pipeline”

How are the routing decisions made?
Using a routing policy
Naïve Routing

Naïve routing

- Tuples enter eddy with low priority
- Tuples marked high priority after going through first operator
- Avoid clogging
- Benefits from fluid dynamic with static selectivity
- Each operator has a limited-size input queue
Lottery routing

- Operators receive tickets with tuples
- Operators lose tickets when returning tuples to eddy
- High tickets = high selectivity
- History from window-based policy