**Query Optimization Techniques for Partitioned Tables**

Herodotos Herodotou, Nedyalko Borisov, Shivnath Babu

Department of Computer Science, Duke University

### Trends in Partitioning
- Growing usage due to increased data sizes
- Growing user control due to new SQL extensions
- DBA may have limited control over partitioning scheme
- Diverse mix of partitioning schemes

### Partitioning Schemes
- Multidimensional (Hierarchical) partitioning
- Tables partitioned on the same key, but with different ranges
- Range partitioning with non-equi ranges

### Problem Definition
- Given a partitioning scheme and a query, find the optimal query execution plan

### Challenges
- Dealing with plan space explosion
- Incorporating into state-of-the-art optimizers
- Partitions as physical or logical properties?
- Supporting a wide range of partitioning conditions

### Sample Query
- SELECT *
  FROM Sales S,
  Payments P,
  Loans L
WHERE S.id = L.id
AND S.id = P.id
AND P.id > 25

### Partition-aware Query Optimization

#### Phase 1: Matching
**Goal:** Identify partition-wise join pairs that can generate output records

**Partition Index Tree (PIT):**
- Core idea: associate each partition with intervals
- Functionalities: index intervals, efficient lookups
- Implementation: augmented red-black tree

**Matching Algorithm**
- Inputs: Table S, Table P, Join condition J
- Step 1: Convert partitioning conditions to intervals
- Step 2: Build PIT with intervals of S
- Step 3: Probe PIT with intervals of P
- Output: Partition-wise join pairs (S_i, P_i)

**Output:**
- Partition Join Pairs
  - (S_i, P_i)
  - (S_j, P_j)

**Probe:**
- (10, 20)

**Notation:**
- Partition: [Interval, Max]

### Phase 2: Clustering
**Goal:** Minimize number of partition-wise join pairs to avoid overheads
- Optimization overheads: enumerating join operators, accessing catalog, calculating cardinality estimates
- Execution-time overheads: accessing the same partition multiple times

**Clustering Algorithm**
- Input: Partition join pairs (output from Matching Phase)
- Step 1: Build bipartite join partition graph
- Step 2: Find connected components using Breadth First Search
- Output: Cluttered join pairs

**Intermediate:**
- Join Partition Graph
  - (S_{12}, P_i)
  - (S_{20}, P_j)

**Output:**
- Cluttered Join Pairs
  - (S_{12}, P_{13})
  - (S_{20}, P_{13})

### Phase 3: Path Creation & Selection
**Goal:** Create and cost partition-wise join paths for child tables

**Bottom-up Query Optimization**
- Find & retain best access paths
- Find & retain best 2-way join paths per interesting order
- Find & retain best 3-way join paths per interesting order and so on

**Extended enumeration**
- Create and cost union of partition-wise joins
- Retain best path per interesting order
- Not enough – Not considering entire plan space (e.g., cannot create 3-way joins)

**Treating Partitions as Physical Properties**
- Interesting partitions in joins can make later joins less expensive
- Retain best path for each interesting order and partition
- Not enough – Not considering entire plan space (e.g., cannot create partition-wise joins with different join orders)

**Treating Partitions as Logical Properties**
- Property 1: Interesting orders independent across child joins
- Property 2: Child joins can have different join orders/operators
- Property 3: Search space includes full extended plan space
- Optimality guarantee: Our bottom-up optimizer will find the optimal plan in the extended plan space

### Experimental Evaluation
- **Prototype using PostgreSQL 8.3.7**
- **TPC-H Benchmark (scale 30)**

**Evaluation Methodology**
- DBA has full Vs. limited control over partitioning scheme
- State-of-the-art Vs. Our partition-aware optimizer

**Evaluation knobs**
- Partitioning scheme
- Number/size of partitions
- Data size

**Optimizer evaluation metrics**
- Execution time
- Optimization time
- Memory utilization

**Experimental Setup**
- **State-of-the-art**
- **Partition-aware**

**Table Partitioning**

<table>
<thead>
<tr>
<th>Trends in Partitioning</th>
<th>Challenges</th>
<th>Sample Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing usage due to increased data sizes</td>
<td>Dealing with plan space explosion</td>
<td>SELECT * FROM Sales S, Payments P, Loans L WHERE S.id = L.id AND S.id = P.id AND P.id &gt; 25</td>
</tr>
<tr>
<td>Growing user control due to new SQL extensions</td>
<td>Incorporating into state-of-the-art optimizers</td>
<td></td>
</tr>
<tr>
<td>DBA must have limited control over partitioning scheme</td>
<td>Partitions as physical or logical properties?</td>
<td></td>
</tr>
<tr>
<td>Diverse mix of partitioning schemes</td>
<td>Supporting a wide range of partitioning conditions</td>
<td></td>
</tr>
</tbody>
</table>

**Partition-aware Query Optimization**

**Phase 1: Matching**

<table>
<thead>
<tr>
<th>Goal</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify partition-wise join pairs that can generate output records</td>
<td>Convert partitioning conditions to intervals</td>
<td>Build PIT with intervals of S</td>
<td>Probe PIT with intervals of P</td>
<td>Output: Partition-wise join pairs (S_i, P_i)</td>
<td></td>
</tr>
</tbody>
</table>

**Matching Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table S, Table P, Join condition J</td>
<td>Convert partitioning conditions to intervals</td>
<td>Build PIT with intervals of S</td>
<td>Probe PIT with intervals of P</td>
<td>Output: Partition-wise join pairs (S_i, P_i)</td>
<td></td>
</tr>
</tbody>
</table>

**Phase 2: Clustering**

<table>
<thead>
<tr>
<th>Goal</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimize number of partition-wise join pairs to avoid overheads</td>
<td>Optimize partitions: enumerating join operators, accessing catalog, calculating cardinality estimates</td>
<td>Execution-time overheads: accessing the same partition multiple times</td>
<td></td>
</tr>
</tbody>
</table>

**Clustering Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition join pairs (output from Matching Phase)</td>
<td>Build bipartite join partition graph</td>
<td>Find connected components using Breadth First Search</td>
<td>Output: Cluttered join pairs</td>
</tr>
</tbody>
</table>

**Phase 3: Path Creation & Selection**

<table>
<thead>
<tr>
<th>Goal</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create and cost partition-wise join paths for child tables</td>
<td>Find &amp; retain best access paths</td>
<td>Find &amp; retain best 2-way join paths per interesting order</td>
<td>Find &amp; retain best 3-way join paths per interesting order and so on</td>
<td>Extended enumeration</td>
<td>Treating partitions as logical properties</td>
</tr>
</tbody>
</table>

**Bottom-up Query Optimization**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype using PostgreSQL 8.3.7</td>
<td>Find &amp; retain best access paths</td>
<td>Find &amp; retain best 2-way join paths per interesting order</td>
<td>Find &amp; retain best 3-way join paths per interesting order and so on</td>
<td>Extended enumeration</td>
<td>Treating partitions as logical properties</td>
</tr>
</tbody>
</table>

**Experimental Setup**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype using PostgreSQL 8.3.7</td>
<td>Find &amp; retain best access paths</td>
<td>Find &amp; retain best 2-way join paths per interesting order</td>
<td>Find &amp; retain best 3-way join paths per interesting order and so on</td>
<td>Extended enumeration</td>
<td>Treating partitions as logical properties</td>
</tr>
</tbody>
</table>