Distributed Databases

CPS 216
Advanced Database Systems

Review

Top-down approach to distributed DBMS
• Data partitioning techniques
  – Horizontal partitioning
    • Round-robin, hash, range, predicate-based
    • Derived horizontal partitioning
  – Vertical partitioning
• Query processing and optimization techniques
• Concurrency control and recovery

Derived horizontal partitioning (slide 1)

Example
• Relations
  – Student(SID, name, dept, …)
  – Department(dept, name, school, …)
• Common query: Student ∪ Department
• Department is partitioned according to school
  – school = 'Art & Science' Department
  – school = 'Engineering' Department
  – …
• How do we partition Student?
  –
Derived horizontal partitioning (slide 2)

• If $R$ (owner relation, e.g., Department) is partitioned into:
  $R_1, R_2, \ldots, R_n$
• Then $S$ (member relation, e.g., Student) should be partitioned into $S$ into:
  $S \bowtie R_1, S \bowtie R_2, \ldots, S \bowtie R_n$
• Recall the definition of semijoin:
  $S \bowtie R_i = p_{\text{attrs}(S)}(S \bowtie R_i)$

Derived horizontal partitioning (slide 3)

• Completeness and reconstructability
  – $S = (S \bowtie R_1) \bowtie (S \bowtie R_2) \bowtie \ldots \bowtie (S \bowtie R_n)$
  – Every $S$ tuple must join with some $R$ tuple
• Disjointness
  – $(S \bowtie R_i) \bowtie (S \bowtie R_j) = \emptyset$ for any $i \neq j$
  – Every $S$ tuple can only join with one $R$ tuple
  – Note: not a precise requirement
  » $S \bowtie R$ is a foreign key join ($S$ references $R$)
  – Example: Student.dept references Department.dept

Vertical partitioning

$R \bowtie \{ p_{\text{attrs}(R_1)}(R_1), p_{\text{attrs}(R_2)}(R_2), \ldots, p_{\text{attrs}(R_n)}(R_n) \}$

  $\text{attrs}(R) = \text{attrs}(R_1) \bowtie \text{attrs}(R_2) \bowtie \ldots \bowtie \text{attrs}(R_n)$
  $\text{attrs}(R_i) \bowtie \text{attrs}(R_j) = \text{key}(R)$ for any $i \neq j$

• Completeness and reconstruction
  – $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
• Disjointness
  – $\text{attrs}(R_i) \bowtie \text{attrs}(R_j) = \text{key}(R)$ for any $i \neq j$
  » Just like
Attribute affinity matrix

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>45</td>
<td>0</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>A₂</td>
<td>0</td>
<td>80</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>A₃</td>
<td>45</td>
<td>5</td>
<td>53</td>
<td>3</td>
</tr>
<tr>
<td>A₄</td>
<td>0</td>
<td>75</td>
<td>3</td>
<td>78</td>
</tr>
</tbody>
</table>

- $A_{ij}$: a measure of how “often” $A_i$ and $A_j$ are accessed by the same query

Partitioning according to AAM

- Cluster attributes based on affinity

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>45</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A₂</td>
<td>45</td>
<td>53</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>A₃</td>
<td>0</td>
<td>5</td>
<td>80</td>
<td>75</td>
</tr>
<tr>
<td>A₄</td>
<td>0</td>
<td>3</td>
<td>75</td>
<td>78</td>
</tr>
</tbody>
</table>

Query rewrite for partitions

- Start with a query plan
- Replace relations by partitions/fragments
- Push ? and ?? up, s and p down
- Simplify and eliminate unnecessary operations
Query rewrite example:
Primary horizontal partitioning

Another query rewrite example:
Primary horizontal partitioning

Query rewrite example:
Derived horizontal partitioning
Query rewrite example:
Vertical partitioning

Execution partitioning
- Data partitioned at different sites
- Result wanted at possibly another site
- Where do query operators execute?
  - Approach 1: operators remain local to sites; add send/receive operators to ship intermediate results between sites
    - Inter-operator parallelism
  - Approach 2: redesign operators to exploit intra-operator parallelism

Send/receive operators
Parallel/distributed query operators

- Sort
  - Parallel range-partitioning sort
  - Parallel merge sort
- Join
  - Partitioning join
  - Asymmetric fragment and replicate join
  - General fragment and replicate join
  - Semijoin reducers

Range-partitioning sort

- Range partition $R$ on the sort key $A$, and then sort each partition locally at destination sites

Merge sort

- Sort $R$ locally at source sites, range partition the sorted results and merge them at destination sites
Selecting a partitioning vector

Possible centralized approach using a coordinator

• Each site sends statistics about its partition to coordinator
  – Could be (low, high, number of tuples), or even a histogram
• Coordinator computes and distributes partitioning vector
  – Could be a vector that equally partitions the relation
• Multiple rounds of refinement possible

Partitioning join

• Partition both $R$ and $S$ according to join key, and then join corresponding partitions locally

More on partitioning join

• Same partition function for both $R$ and $S$
  – Can be either range or hash partitioning
• Equijoins work best
• Any type of local join algorithm can be used
• Several possible variants, e.g.
  – Partition $R$; partition $S$; join
  – Partition $R$ and build a hash table for $R$; partition $S$ and join
Asymmetric fragment & replicate join

- Partition $R$, replicate $S$, and then join each partition of $R$ with a replica of $S$ locally

![Diagram showing the asymmetric fragment & replicate join process]

General fragment & replicate join

- Suppose $m \neq n$ sites participate in join
- Partition $R$ into $R_1, R_2, \ldots, R_m$
- Partition $S$ into $S_1, S_2, \ldots, S_n$
- Each site receives a copy of $R_i$ and a copy of $S_j$ and joins them locally
  - Each $R_i$ needs to be replicated $n$ times
  - Each $S_j$ needs to be replicated $m$ times

Semijoin reducer

$R(A, B) \bowtie S(A, C)$

- Naive strategy: ship $R$ Site 2 and join it there with $S$
- Problem
  - All $R$ tuples are shipped, but few actually join
  - Lots of bandwidth wasted in sending useless $R$ tuples!
- Idea
  - $R \bowtie S = (R \bowtie ? S) \bowtie S = R \bowtie (S \bowtie ? R)$
  - Use semijoins to reduce the number of tuples that need to be shipped to join at another site
Semijoin reducer in action

\[ R(A, B) \bowtie S(A, C) \]

Site 1 Site 2

- Site 2 computes \( p_A S \) and sends it to Site 1
- Site 1 computes \( R \bowtie S = R \bowtie p_A S \) and sends it to Site 2
- Site 2 computes \( R \bowtie S = (R \bowtie S) \bowtie S \)

- Communication costs
  - Naïve: \( \text{sizeof}(R) \)
  - Semijoin: \( \text{sizeof}(p_A S) + \text{sizeof}(R \bowtie S) \)
  - Greater savings if there is a local selection on \( S \)

Semijoin reducer tricks

- Encode \( p_A S \) as a bitmap
  - One bit for each possible value in the domain of \( A \)
  - What if the domain is too big? What if we only want to send \( n \) bits?
- Encode \( p_A S \) as a bloom-filter of \( n \) bits
  - Hash each \( S \cdot A \) value to an offset from 0 to \( n - 1 \)
  - Bloom-filer is lossy and may generate false positives
    - Example: \( a \bowtie p_A S, b \bowtie p_A S, \text{hash}(a) = \text{hash}(b) = 1 \); \( R \) tuples with value \( b \) are sent to \( S \)—unnecessary but harmless
  - Similar to the idea of signature files

Full reducer

\[ R_1 \bowtie \ldots \bowtie R_n \]

- \( R_i \) is reduced if \( R_i = p_{\text{attrs}}(R_1 \bowtie \ldots \bowtie R_n) \)
- A series of semijoins is called a full reducer if every \( R_i \) is reduced after executing the semijoins
  - That is, there are no dangling tuple at all!
- Full reducer for \( R(A, B) \bowtie S(B, C) \bowtie T(C, D) \)
  - \( S \bowtie ? \bowtie ? \bowtie R \)
  - \( T \bowtie ? \bowtie ? \bowtie S \)
  - \( S \bowtie ? \bowtie ? \bowtie R \bowtie ? \bowtie S \)
- Full reducer for \( R(A, B) \bowtie S(B, C) \bowtie T(C, A) \)
  - None!
Join hypergraph

- A node is an attribute; matching join attributes share the same node
- A hyperedge connects attributes from the same relation
- For hyperedges $E$ and $F$, if the attributes in $E - F$ are unique to $E$ (not in any other hyperedge), then $E$ is an ear
- A join hypergraph is acyclic if we can continue removing ears until there is nothing left
  - That is, the graph is really a tree (think of ears as leaves)

Full reducer for acyclic hypergraph

- Theorem: A join has a full reducer iff the join hypergraph is acyclic
- Algorithm
  - Remove an ear $R$; say it hangs off $S$
  - $S \not\rightarrow R \Leftrightarrow S$ is reduced w.r.t. $R$
  - Generate a full reducer for the remaining hypergraph
  - $R \not\rightarrow S \Leftrightarrow$ Other relations are reduced w.r.t. $R$ through $S$
  - $S$ is further reduced w.r.t. other relations through $S$

Next time

- Optimizing distributed queries
- Concurrency control and recovery
- Bottom-up approach to building a distributed database
- Data warehousing