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

2

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
  - $s_{\text{school = 'Art \& Science'}}$ Department
  - $s_{\text{school = 'Engineering'}}$ Department
  - …
- How do we partition Student?
  - Same partitioning scheme as Department

3

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 ⊳< R_1, S ⊳< R_2, …, S ⊳< R_n
- Recall the definition of semijoin:
  S ⊳< R_i = $p_{\text{attrs}(S)}$(S ⊳⊲ R_i)

4

Derived horizontal partitioning (slide 3)

- Completeness and reconstructability
  - S = (S ⊳< R_1) ∪ (S ⊳< R_2) ∪ … ∪ (S ⊳< R_n)?
  - Every S tuple must join with some R tuple
- Disjointness
  - (S ⊳< R_i) ∩ (S ⊳< R_j) = Ø for any i ? j?
  - Every S tuple can only join with one R tuple
  - Note: not a precise requirement

» S ⊳⊲ R is a foreign key join (S references R)
  - Example: Student.dept references Department.dept

5

Vertical partitioning

R ? {$p_{\text{attrs}(R_1)}R$, $p_{\text{attrs}(R_2)}R$, …, $p_{\text{attrs}(R_k)}R$ }

attrs(R) = attrs(R_1) ∪ attrs(R_2) ∪ … ∪ attrs(R_k)

attrs(R_i) ∩ attrs(R_j) = key(R) for any i ? j

- Completeness and reconstruction
  - R = R_1 ⊳⊲ R_2 ⊳⊲ … ⊳⊲ R_n
- Disjointness
  - attrs(R_i) ∩ attrs(R_j) = key(R) for any i ? j

» Just like lossless-join decomposition and DSM

6

Attribute affinity matrix

	A_1	A_2	A_3	A_4
A_1	45	0	45	0
A_2	0	80	5	75
A_3	45	5	53	3
A_4	0	75	3	78

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

7

Partitioning according to AAM

- Cluster attributes based on affinity

	A_1	A_3	A_2	A_4
A_1	45	45	0	0
A_3	45	53	5	3
A_2	0	5	80	75
A_4	0	3	75	78

8

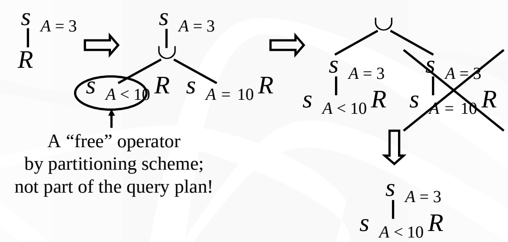
Query rewrite for partitions

- Start with a query plan
- Replace relations by partitions/fragments
- Push $\cup$ and $\bowtie$ up, σ and π down
- Simplify and eliminate unnecessary operations

9

Query rewrite example:

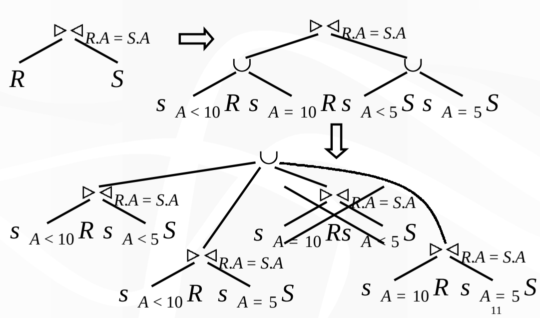
Primary horizontal partitioning



10

Another query rewrite example:

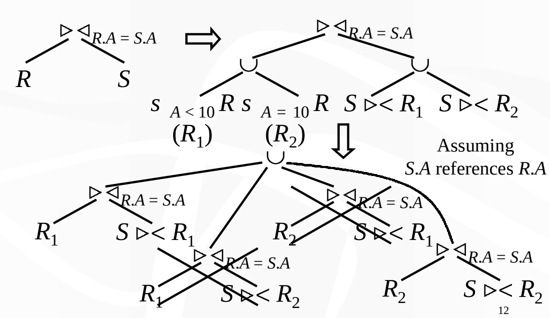
Primary horizontal partitioning



11

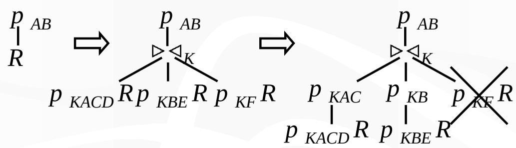
Query rewrite example:

Derived horizontal partitioning



12

Query rewrite example:
Vertical partitioning



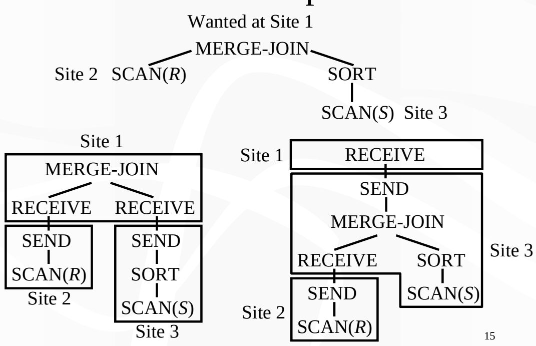
13

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

14

Send/receive operators



15

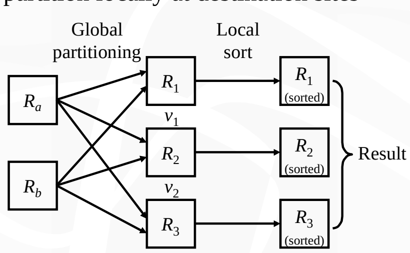
Parallel/distributed query operators

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

16

Range-partitioning sort

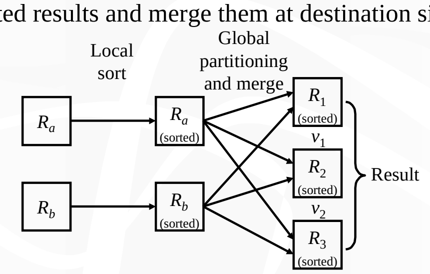
- Range partition R on the sort key A, and then sort each partition locally at destination sites



17

Merge sort

- Sort R locally at source sites, range partition the sorted results and merge them at destination sites



18

Selecting a partitioning vector

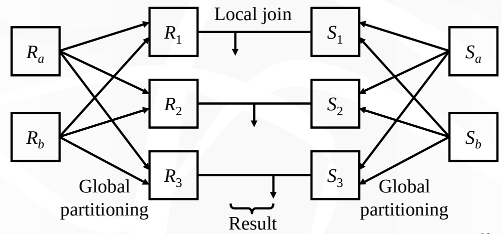
Possible centralized approach using a coordinator

- 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

19

Partitioning join

- Partition both *R* and *S* according to join key, and then join corresponding partitions locally



20

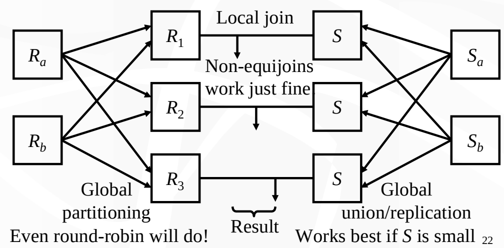
More on partitioning join

- 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

21

Asymmetric fragment & replicate join

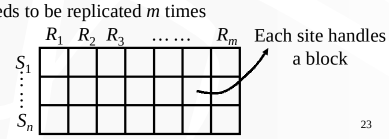
- Partition *R*, replicate *S*, and then join each partition of *R* with a replica of *S* locally



22

General fragment & replicate join

- Suppose $m \times n$ 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



23

Semijoin reducer

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

Site 1 Site 2

- Naïve 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
  - $R \bowtie S = (R \ltimes S) \bowtie S = R \bowtie (S \ltimes R)$
    $= (R \ltimes S) \bowtie (S \ltimes R)$
  - Use semijoins to reduce the number of tuples that need to be shipped to join at another site

24

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

- Communication costs
  - Naïve: sizeof(R)
  - Semijoin: sizeof($p_A S$) + sizeof($R \ltimes S$)
  - Greater savings if there is a local selection on S

25

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 $S.A$ value to an offset from 0 to $n - 1$
  - Bloom-filer is lossy and may generate false positives
    - Example: $a \in p_A S$, $b \notin p_A S$, hash(a) = hash(b) = 1; R tuples with value b are sent to S—unnecessary but harmless
  - Similar to the idea of signature files

26

Full reducer

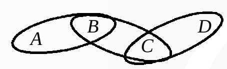
$R_1 \bowtie \dots \bowtie R_n$

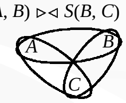
- R_i is reduced if $R_i = p_{\text{attrs}(R_i)}(R_1 \bowtie \dots \bowtie R_n)$
- A series of semijoins is called a full reducer if every R_i is reduced after executing the semijoins
  - That is, there are no dangling tuple at all!
- Full reducer for $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$
  - S ?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

27

Join hypergraph

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





- 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)

28

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 ? $S \ltimes R$ ← S is reduced w.r.t. R
  - Generate a full reducer for the remaining hypergraph
  - R ? $R \ltimes S$

Now R is reduced w.r.t. S, and w.r.t. other relations through S

Other relations are reduced w.r.t. R through S; S is further reduced w.r.t. other relations

29

Next time

- Optimizing distributed queries
- Concurrency control and recovery

- Bottom-up approach to building a distributed database

- Data warehousing

30