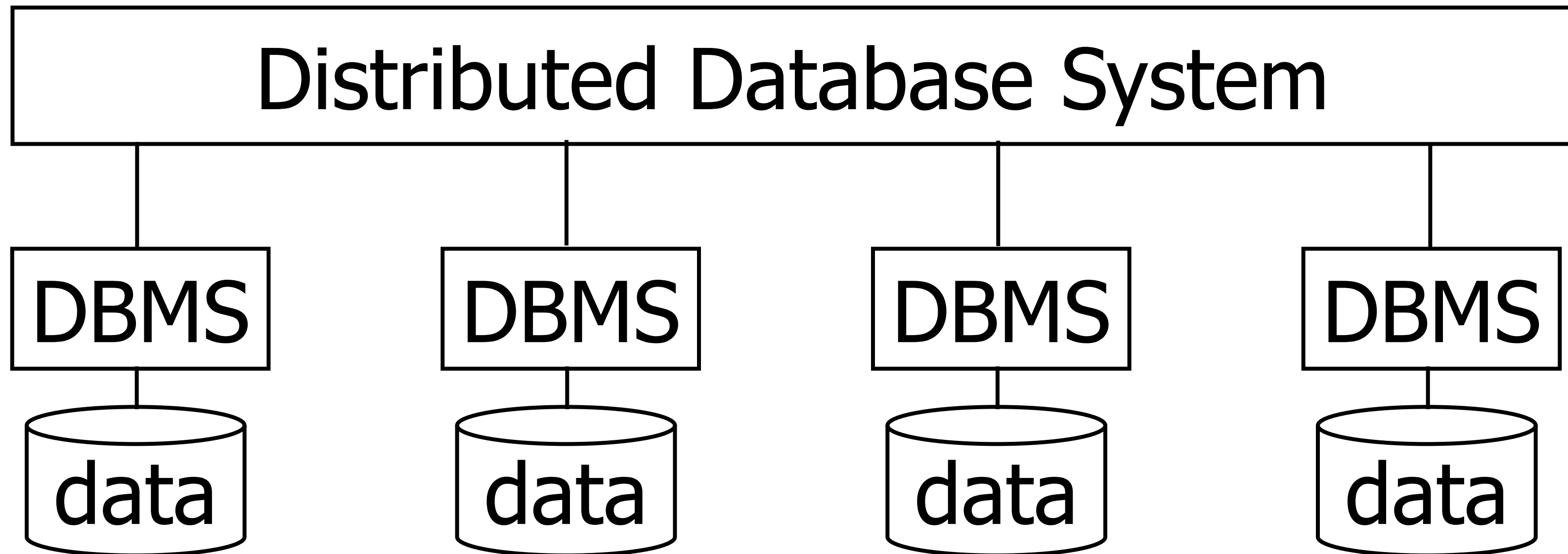
Distributed Databases

Distributed Databases



Data spread and shared across sites

Key advantages

- Fault Tolerance
- High Performance

Distributed DB Challenges

- Local and global transactions
  - A local transaction accesses data in the *single* site.
  - A *global transaction* accesses data in several sites.

→Need to ensure atomicity for global transactions.

- Replication of data items required for improving data availability

→Need to develop distributed locking mechanisms.

- Distributed query processors and data storages.

→Need efficient query optimization techniques.

Outline

- Distributed Commit (2 Phase Commit)
- Distributed Locking
- Distributed Querying
- Distributed Hashing (peer-to-peer systems)

Distributed Commit Protocol

- A global transaction must either be committed at **all** sites, or aborted at all sites.
- Solution: Two-phase commit (2PC) protocol.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model
  - failed sites simply *stop* working [rather than become rogue]
- 2PC initiated by the coordinator C after the last step of the transaction T.

Phase 1: Obtaining a Decision

- C asks all sites to prepare to commit T.
  - C adds record <**prepare** T> to log, and flushes the log.
  - Sends "**prepare** T" messages to others.
- Upon receiving message, each site determines if it can commit:
  - if yes, then add <**ready** T> to the log, flush the log, and send "**ready** T" message to C. *Now, a site cannot abort T unless instructed by C.*
  - if not, add <**Don't commit T**> to the log, and send "**Don't commit T**" message to C

Phase 2: Recording the Decision

- If *C* received a "**ready** *T*" message from all participating sites, then commit, else abort.
- C adds <**commit** *T*> or <**abort** *T*> to the log, flushes the log.
- C informs others of the decision.
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When a site S recovers, it examines its log.

- Log contains <**commit** T>: Execute **redo** (T)
- Log contains <**abort** T>: Executes **undo** (T)
- Log contains <**ready** T>: Consult C.
  - If T committed, **redo** (T)
  - If T aborted, **undo** (T)
- Log contains no control records concerning T
  - T *must* have been aborted by C. So, execute **undo** (T)

Coordinator Failure

- If coordinator fails, then a new "coordinator" elected. T's fate is decided based on:
    1. If a site contains <**commit** *T*> record, then *T* must be committed.
    2. If a site contains <**abort** *T*> record, then *T* must be aborted.
    3. If some site does not contain a <**ready** *T*> record, then abort *T*.
    4. Remaining case: all active sites must have a <**ready** *T*> record, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case, must wait for *C* to recover, to find decision.

Outline

- Distributed Commit (2 Phase Commit)
- Distributed Locking
- Distributed Querying
- Distributed Hashing (peer-to-peer systems)

Distributed Locking Protocols

- Data may be replicated across sites.
- Whenever written, we assume all replicas of any item are updated

Possible Locking Strategies:

- One copy of each data
  - Single Lock Manager
  - Distributed Lock Manager
- Replicated copies of each data
  - Primary copy
  - General case

One copy: Single Lock Manager

- System maintains a *single* lock manager at some site S.
- Lock requests sent to S, which determines if the lock can be granted.

One copy: Distributed Lock Manager

- Lock managers at each site
  - Lock managers control access to local data items
- Robust to failures
- Deadlock detection is more complicated

Replicated Items: Primary Copy

- Choose one copy to be the *primary copy*
  - Stored at *primary site*
- Lock requests sent at the primary site.
  - Implicitly lock granted on all replicas
- Simple implementation.
- Not robust.

Replicated Items: Majority Protocol

- General Case: Distributed lock managers, and multiple replicas.
  - If Q is replicated at n sites, then a lock request is sent to more than $n/2$ sites.
  - Need grants from at least $n/2$ sites.
  - Write on *all* replicas.
- Robust.

Majority Protocol (Drawbacks)

- Drawback
  - Requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests.
  - Potential for deadlock even with single item - e.g., each of 3 transactions may have locks on 1/3 of the replicas.

Replicated copies: Biased Protocol

- Treat read and write locks differently. Note that majority needed only for write locks.
- E.g.,
  - Shared locks. Get a lock from any one site.
  - Exclusive locks. Get a lock from all sites.
- In general, s locks for shared locks and x locks for exclusive locks are sufficient, as long as $(s + x) > n$ and $(2x) > n$.

Site-Weighted Biased Protocol

- Each site is assigned a weight.
  – Let S be the total of all site weights
- Choose two values: read quorum Q_r and write quorum Q_w
  – Such that $Q_r + Q_w > S$ and $2*Q_w > S$
  – Quorums can be chosen (and S computed) separately for each item
- Each read (write) must lock enough replicas that the sum of the site weights is >= Q_r (Q_w)

Outline

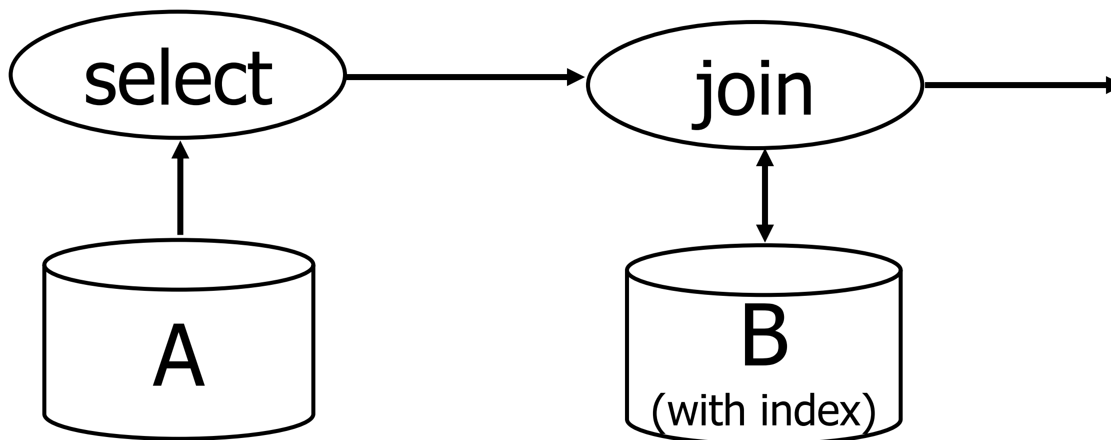
- Distributed Commit (2 Phase Commit)
- Distributed Locking
- Distributed Querying
- Distributed Hashing (peer-to-peer systems)

Distributed Query Processing

- Minimize cost of a data transmission.
- Maximize computation parallelism.

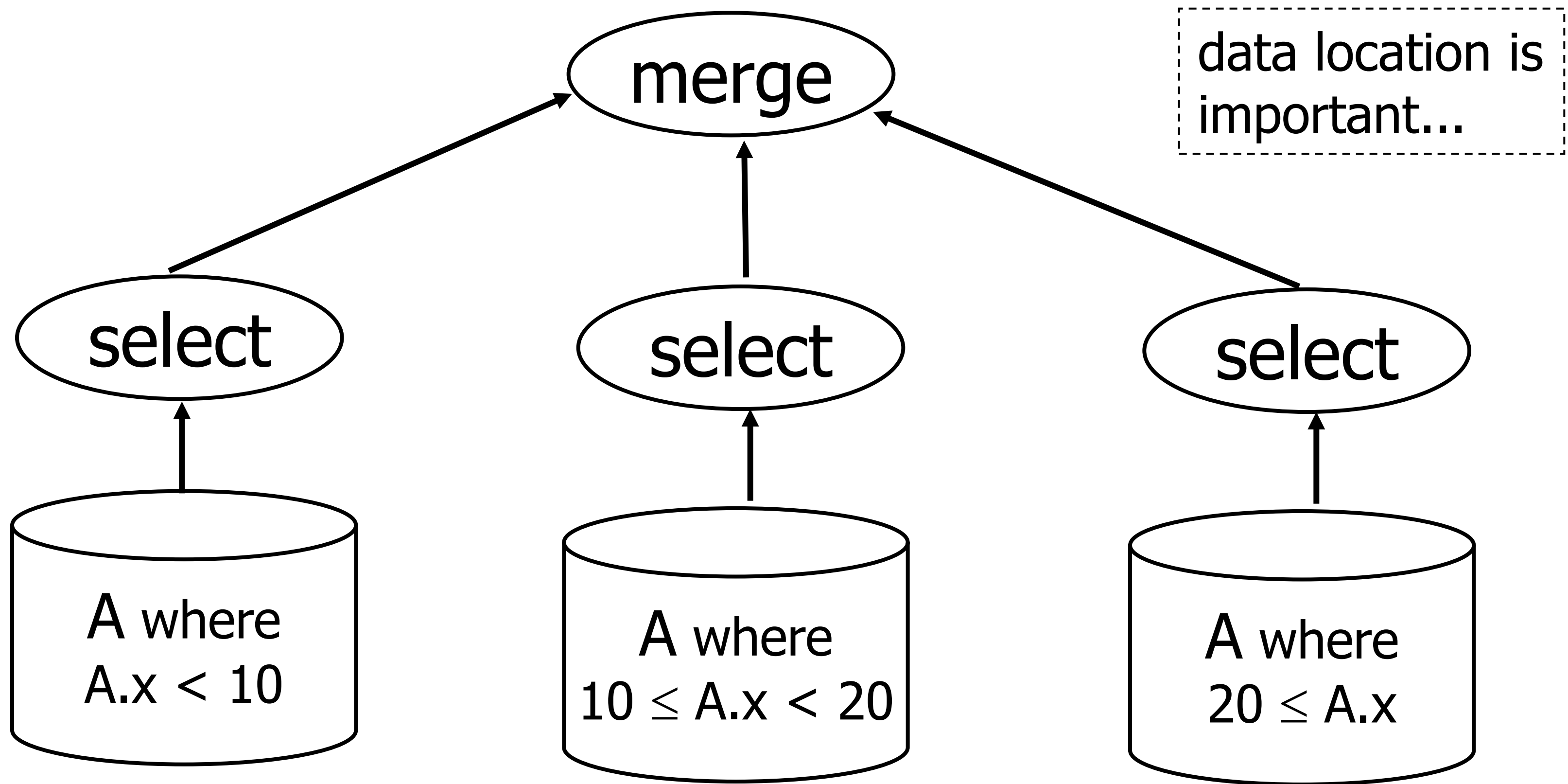
Parallelism I: Pipelining

- $T_1 \leftarrow$ SELECT *
  FROM A
  WHERE <condition>
- $T_2 \leftarrow$ B JOIN T_1



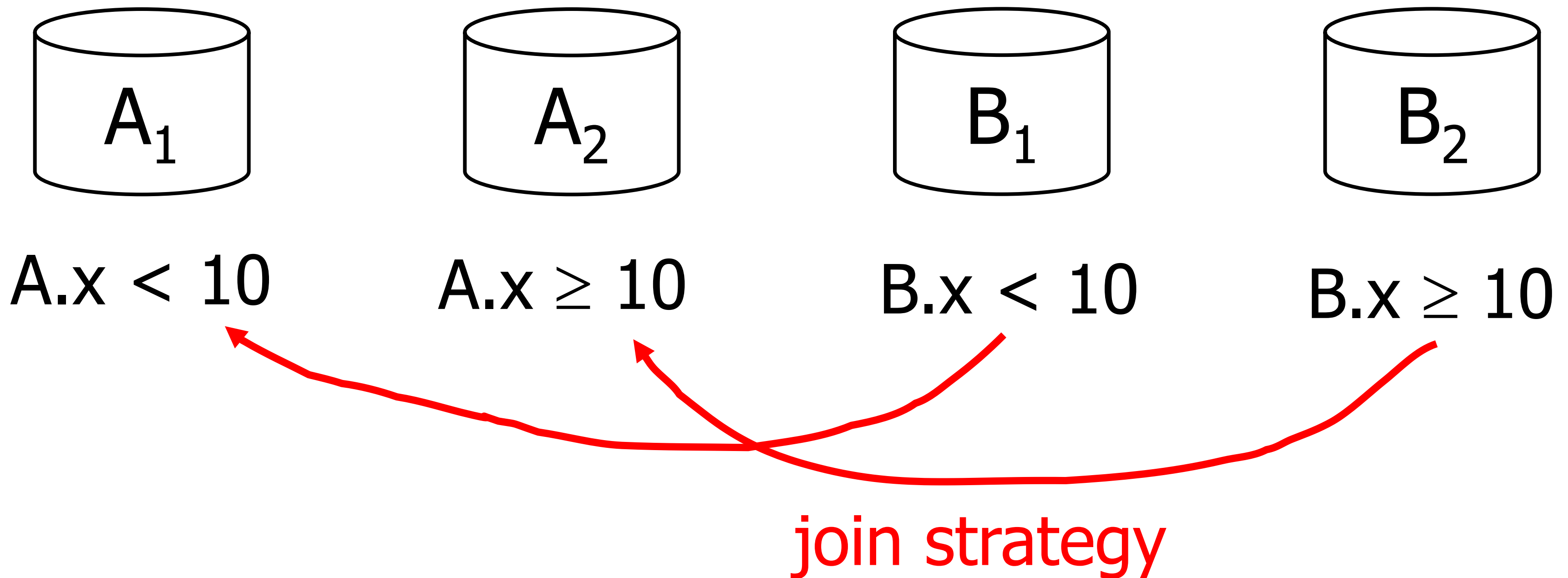
Parallelism II: Concurrent Operations

- SELECT * FROM A WHERE <condition>



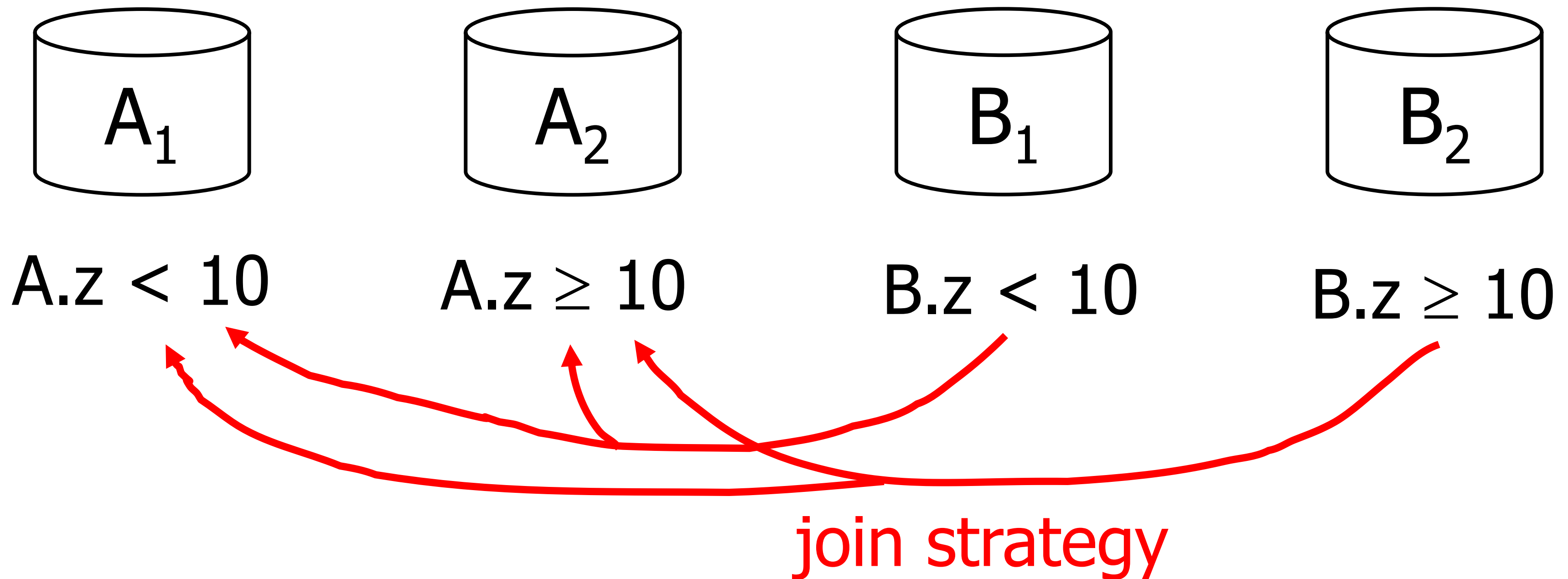
Parallel Join Processing : I

A NATURAL-JOIN B [over attribute X]



Parallel Join Processing : II

A NATURAL-JOIN B [over attribute X]



Parallel Join Processing: Example

- Consider the following relational algebra expression

  account $\bowtie$ *depositor* $\bowtie$ *branch*
- *account* is stored at site S_1
- *depositor* at S_2
- *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I

Example: Possible Strategies

1. Ship all relations → S_I. Local optimization at $S_{I.}$
2. Another Strategy:
   - Ship *account* relation to site S_2
   - Compute $temp_1 = account \bowtie depositor$ *at* S_2.
   - Ship $temp_1$ to S_3
   - Compute $temp_2 = temp_1 \bowtie branch$ at S_3.
   - Ship the result $temp_2$ to S_I.
3. Devise similar strategies, exchanging the roles S_1, S_2, S_3

Example: Semijoin Strategy

Let r_1 be a relation stored at site S_1

Let r_2 be a relation stored at site S_2

Evaluate $r_1 \bowtie r_2$ and obtain the result at S_1.

a. Compute $temp_1 \leftarrow \Pi_{\text{Attrs(r1)} \cap \text{Attrs(r2)}} (r_1)$ at $S1$.

b. Ship $temp_1$ from S_1 to S_2.

c. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at S_2

d. Ship $temp_2$ from S_2 to S_1.

e. Compute $r_1 \bowtie temp_2$ at S_1. This is $r_1 \bowtie r_2$.

Semijoin Definition

The **semijoin** of r_1 with r_2, is denoted by:

$$r_1 \ltimes r_2$$

and defined as:

$$\Pi_{R1} (r_1 \bowtie r_2)$$

Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contribute to $r_1 \bowtie r_2$.

In step (c) above, $temp_2 = r_2 \ltimes r_1$.

For joins of several relations, the above strategy can be extended to a series of semijoin steps.

Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation r_i is stored at site S_i. The result must be presented at site S_1.
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2.
- Simultaneously, r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they produced;
  S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1 $(r_1 \bowtie r_2)$ $(r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.

Outline

- Distributed Commit (2 Phase Commit)
- Distributed Locking
- Distributed Querying
- Distributed Hashing (peer-to-peer systems)

Peer-to-Peer Networks

Setting:

- Autonomous, homogeneous, and loosely-coupled peers that share resources.
- E.g., Napster, peer-to-peer Youtube.

Distributed Hashing Problem:

- (K,V), i.e., (key,value), pairs to be distributed across the peers, so that, given K, its V can be located in small number of messages.
- Desire a truly distributed solution [the index should be distributed, with no replication of entries].

Distributed Hashing Solutions

Simple solutions:

- Centralized look-up database that gives location of V for any K. Use hash function to map K and hence (K,V)'s to locations. E.g., Google.

Chord Circles:

- Use a hash function to map K's and peer-IDs onto a "circle" of hash values.
- Storage, Finger-tables, Look-Up, Dynamicity.