Distributed DBMS

Immanuel Trummer
itrummer@cornell.edu
www.itrummer.org

[RG, Sec. 21]

Outlook (Next Lectures)

- (**Classical**) distributed database systems
- Eventually consistent systems (**NoSQL**)
- Stronger consistency models (**NewSQL**)

Distributed DBMS

- Systems that **distributing data** over multiple sites
- Systems at different sites can typically run **independently**
- **Heterogeneous** versus **homogeneous** distributed DBMS
- Users do not need to know data **location** when querying
  - Distributed data **independence**

Why Distributed DBMS?

- **Availability**
- **Scalability**
- **Autonomy**

Distributed DBMS Architectures

- **Client-Server**
  - Client interacts with a server storing data
  - Simple to implement but no multi-site queries
- **Collaborating servers**
  - Each server stores and processes data locally
  - Each server may launch initiate distributed processing
- **Middleware systems**
  - Middleware controls servers for distributed processing

How to Distribute Data?

- **Horizontal** fragmentation (partition rows)
- **Vertical** fragmentation (partition columns)
- **Replication** (multiple copies of data)

Distributed DBMS Topics

- Distributed **catalog management**
- Distributed **query processing**
- Distributed **concurrency control**
- Distributed **recovery**

Distributed DBMS Topics

- Distributed **catalog management**
- Distributed **query processing**
- Distributed **concurrency control**
- Distributed **recovery**

Catalog Management

- **Centralized** catalog: catalog stored at one site
  - Vulnerable to failure and performance problems
- **Global** catalog: each site stores a copy
  - Must update all sites in case of changes
- **Local** catalog at each site
  - Each site responsible for objects created there
  - Requires only one update but remote reads

Distributed DBMS Topics

- Distributed **catalog management**
- Distributed **query processing**
- Distributed **concurrency control**
- Distributed **recovery**

Queries without Joins

- SELECT Avg(C.age) FROM Customers C
  WHERE C.ID > 50,000 AND C.ID < 100,000

- ***How to process this query in the following scenarios?***

- **Horizontally partitioned** data (disjunct rows on nodes)

- **Vertically partitioned** data (different columns on nodes)

- **Replicated** data (copies of same data on different nodes)

Execution Cost

- So far: measured cost by number of **pages** read/written
- Additionally, need to consider cost for **sending** data
- May have to send query **result** to site issuing query
- Introduce two cost constants
  - "**Net**" is cost of sending page over the network
  - "**Disk**" is cost for reading/writing page from/to disk

Queries with Joins

- SELECT * FROM Customer NATURAL JOIN Orders

- Assume that **tables** are stored on different nodes

- ***How can we perform the distributed join ... ?***

Naive Join Algorithm

- Load maximum size **blocks** of first table into memory
- For each block, **iterate** over pages from second table
  - Send each page from table 2 over network and join
- (Assume that join result does not need to be sent again)
- **Cost:**
  #pages in table 1 * Disk +
  #blocks in table 1 * #pages in table 2 * (Net + Disk)

Send All Data to One Site

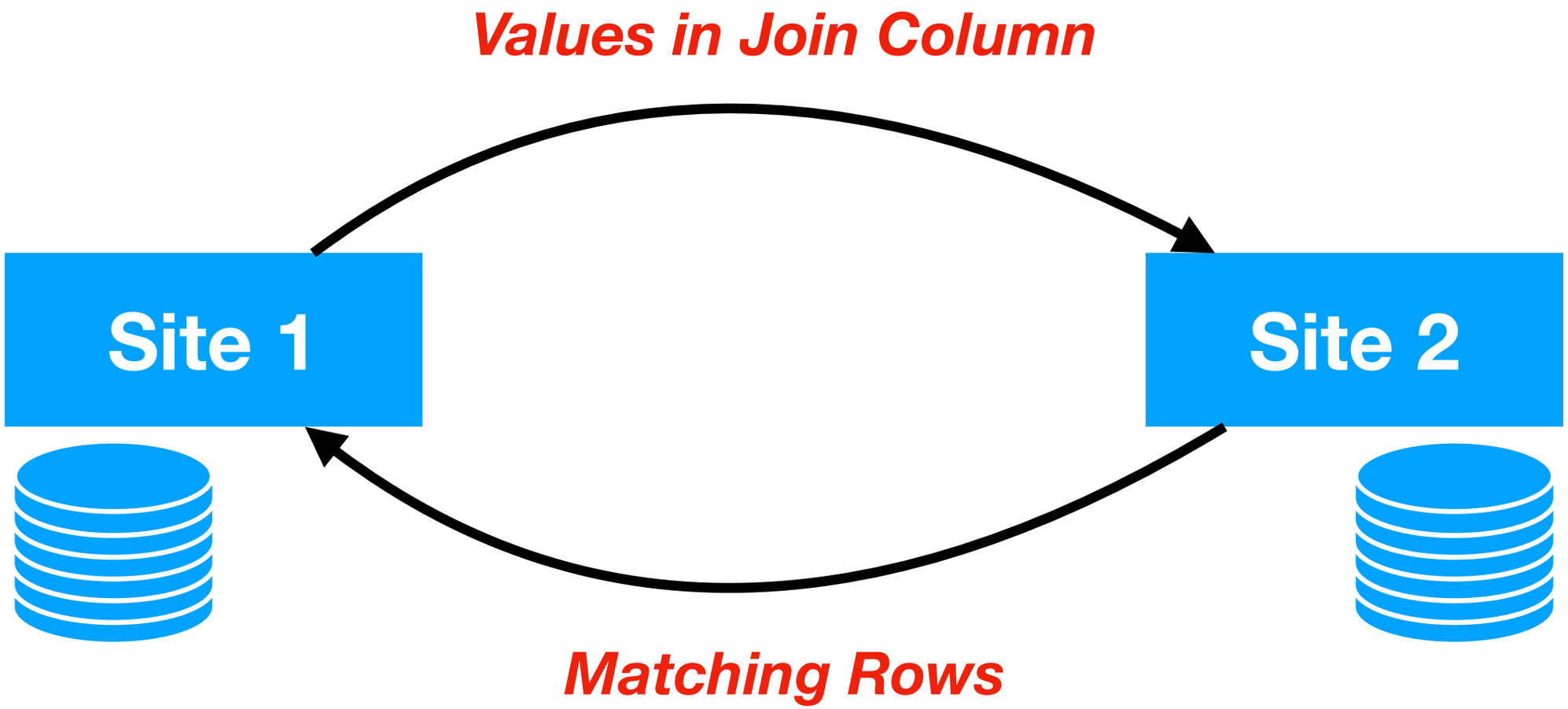
- Idea: send **all data to same node** and perform join there

- **Cost:**
  #pages in table 2 * Net +
  #pages in table 1 * Disk +
  #blocks in table 1 * #pages in table 2 * Disk

Semijoin

- Idea: **avoid sending rows** that do not join with anything
- Here we assume a binary equality join condition
- Step 1: **collect** distinct values in join column of table 1
- Step 2: **send** those values to site storing table 2
- Step 3: collect rows **matching** values on site 2
- Step 4: send **back** matching rows only
- Step 5: **join** table 1 rows with matching rows

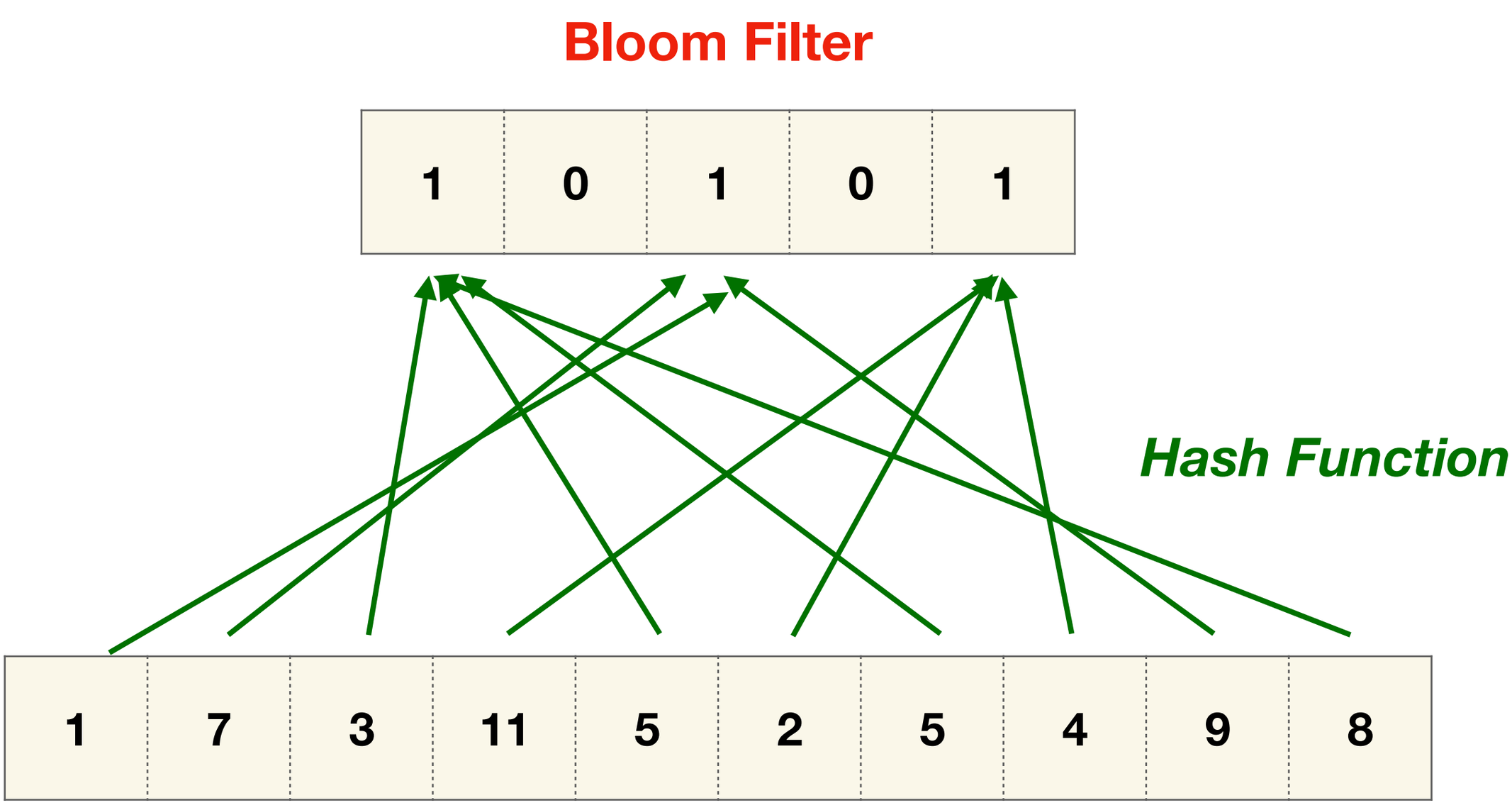
Semijoin Illustration



Bloom Filter

- A data structure for **testing** whether an element is in a set
  - Very **space** efficient but produces **false positives**
- Contains one bit for each **hash bucket**
  - Bit **set to one** if at least one element falls into bucket
  - Bit set to zero otherwise
- Element **cannot** be in set if its hash bucket bit is zero

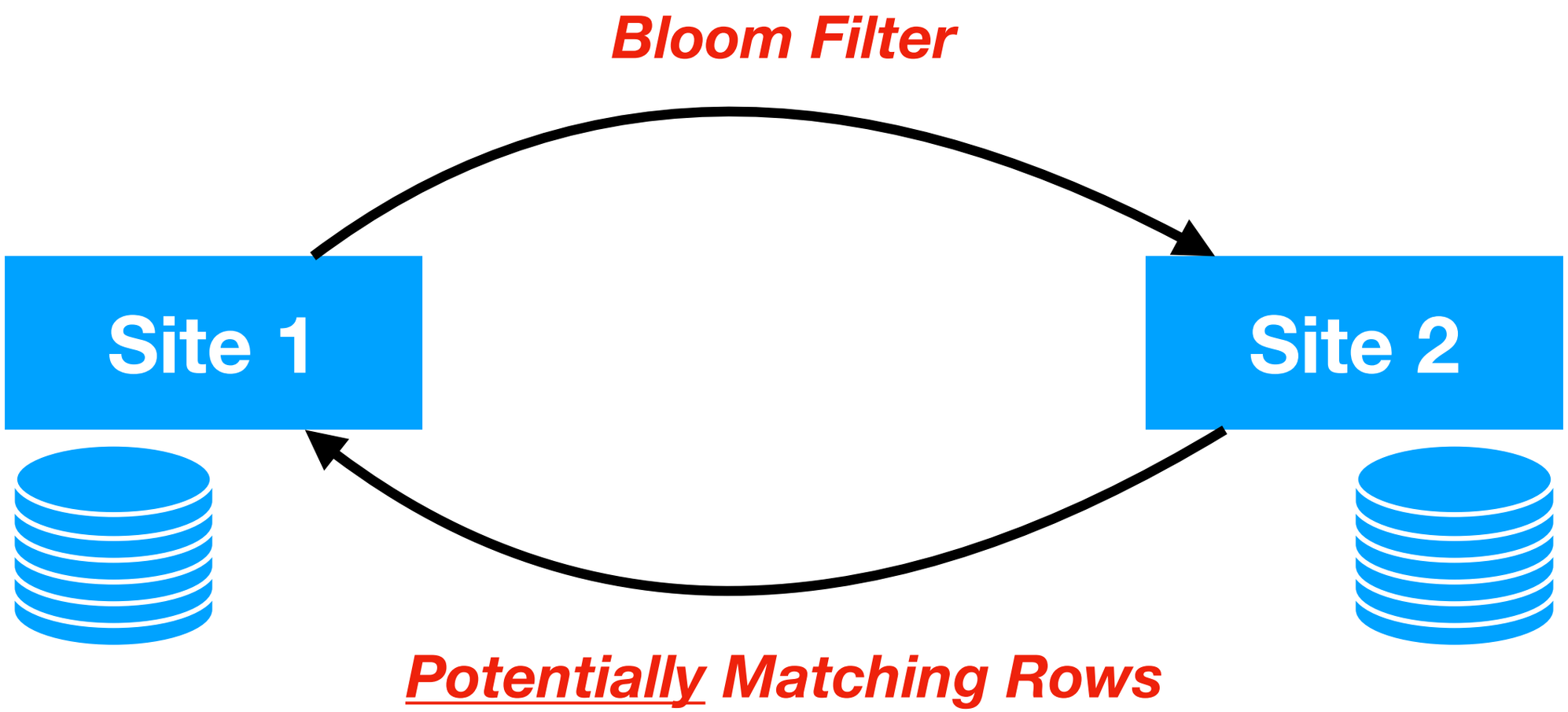
Bloom Filter Illustration



Bloom Join

- Step 1: **collect** distinct values in join column of table 1
- Step 2: **approximate** that value set via a Bloom filter
- Step 3: **send** Bloom filter to site storing table 2
- Step 4: collect rows **potentially** matching values on site 2
- Step 5: send back **potentially** matching rows only
- Step 6: join table 1 rows with **potentially** matching rows

Bloom Join Illustration



Semi vs. Bloom Join

- Bloom join reduces data sent to identify **candidate rows**
- Semi-join reduces rows that are sent back **in response**
- **Scenario** determines which join works best!

Distributed Query Optimization

- As before: we select **cheapest plan** based on cost model
- Difference 1: cost model integrates **communication** cost
- Difference 2: new choices for distributed **operators**
- Difference 3: may have to respect **autonomy** of local sites

Distributed DBMS Topics

- Distributed **catalog management**
- Distributed **query processing**
- Distributed **concurrency control**
- Distributed **recovery**

Distributed Concurrency Control

- Transactions may access objects on **multiple nodes**
  - **Local** concurrency control is insufficient
- Can use **distributed variant** of two-phase locking
  - Q1: where should we store the **locks**?
  - Q2: how to handle distributed **deadlocks**?

Storing Locks

- **Centralized**: all locks managed by one single site
  - Vulnerable to failure of single site for locking
- **Primary copy**: primary copy site manages its locks
  - Locking requires communication with primary copy
- **Fully distributed**: each site manages locks of its objects
  - Requires communication between sites for writes

Distributed Deadlocks

- **Centralized**: send waits-for graphs to one single site
  - Central site may become the **bottleneck**
- **Hierarchical**: pass on graphs in a site hierarchy
  - May take **time** to detect deadlocks across certain sites
- **Timeouts**: assume deadlock after a timeout
  - May lead to unnecessary **aborts** if timeout too low

Distributed DBMS Topics

- Distributed **catalog management**
- Distributed **query processing**
- Distributed **concurrency control**
- Distributed **recovery**

Distributed Recovery

- Recovery is **complex** for single node already
  - Here, system is either running or not
- Only **some** nodes may fail in distributed system
- **Communication** between nodes may fail

Distributed Recovery

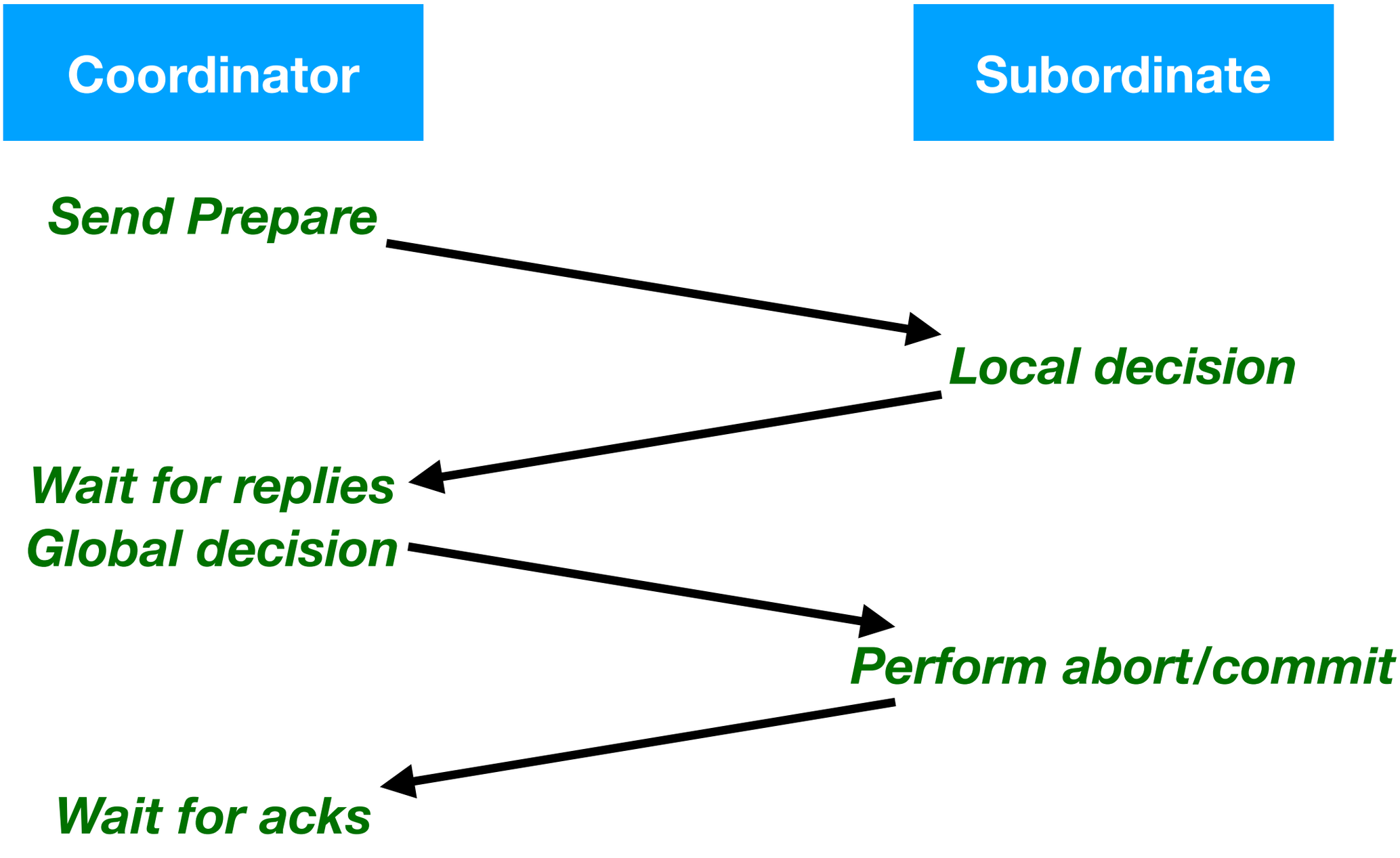
- Recovery is **complex** for single node already
  - Here, system is either running or not
- Only **some** nodes may fail in distributed system
- **Communication** between nodes may fail

Need to coordinate between nodes for consistent data!

Two-Phase Commit

- Guarantees distributed transaction **atomicity**
  - I.e., distributed transaction affects **all sites or none**
- Protocol distinguishes **coordinator** and **subordinate**
- Involves **two rounds** of communication between them
- Each site maintains persistent **local log** for recovery

Two-Phase Commit



Failure Detection

- Participants write **log entries** before sending messages
  - Allows us to recognize **at which stage** we failed
- Participants assume failure if no reply until a **timeout**
  - Allows us to recognize **other failed participants**

Recovery by Phase and Role

	Subordinate	Coordinator
Phase 1	Failure before reply: abort	Assume abort by default
Phase 2	Failure after sending no: abort Failure after sending yes: query coordinator	Read global decision from log, inform subordinates