

CompSci 516
Data Intensive Computing Systems

Lecture 19 - 20
Distributed Databases

Instructor: Sudeepa Roy

Reading Material

- [RG]
 - Chapter 22.6 – 22.14

Acknowledgement:

The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

So far...

- On a single machine:
 - Query Execution and Optimization
 - Transaction CC and Recovery
- With multiple machines:
 - Parallel DBMS
 - Map-Reduce and Spark

Today

- Distributed DBMS
 - Architecture
 - Data Storage
 - Query Execution
 - Transactions – updates
 - Recovery – Two Phase Commit (2PC)

Introduction: Distributed Databases

- Data is stored at several sites, each managed by a DBMS that can run independently
- Desires properties
 - Distributed Data Independence
 - Distributed Transaction Atomicity

Distributed Data Independence

- Users should not have to know where data is located
 - no need to know the locations of references relations, their copies or fragments (later)
 - extends Physical and Logical Data Independence principles
- Queries spanning multiple sites should be optimized in a cost-based manner
 - taking into account **communication costs** and differences in **local computation costs**

Distributed Transaction Atomicity

- Users should be able to write transactions accessing multiple sites just like local transactions
- The effects of a transaction across sites should be atomic
 - all changes persist if transaction commits
 - none persist if transaction aborts

Recent Trends on These Two Properties

- These two properties are in general desirable
- But not always efficiently achievable
 - e.g. when sites are connected by a slow long-distance network
- Even sometimes not desirable for globally distributed sites
 - too much administrative overhead of making location of data transparent
- Therefore not always supported
 - Users have to be aware of where data is located
 - Not much consensus on the design objectives on distributed databases

Recall: Parallel vs. Distributed DBMS

Lecture 11

today

Parallel DBMS

- Parallelization of various operations
 - e.g. loading data, building indexes, evaluating queries
- Data may or may not be distributed initially
- Distribution is governed by **performance consideration**

Distributed DBMS

- Data is physically stored across different sites
 - Each site is typically managed by an independent DBMS
- **Location of data** and autonomy of sites have impact on Query opt., Conc. Control and recovery
- Also governed by other factors:
 - **increased availability** for system crash
 - **local ownership and access**

Types of Distributed Databases

- **Homogeneous:**
 - Every site runs same type of DBMS
- **Heterogeneous:**
 - Different sites run different DBMSs
 - different RDBMSs or even non-relational DBMSs
 - RDBMS = Relational DBMS

More on Heterogeneous Distributed Databases

- Database servers are accessed through well-accepted and standard **Gateway protocols**
 - masks the differences of DBMSs (capability, data format etc.)
 - e.g. ODBC, JDBC
- However, can be expensive and may not be able to hide all differences
 - e.g. when a server is not capable of supporting distributed transaction management



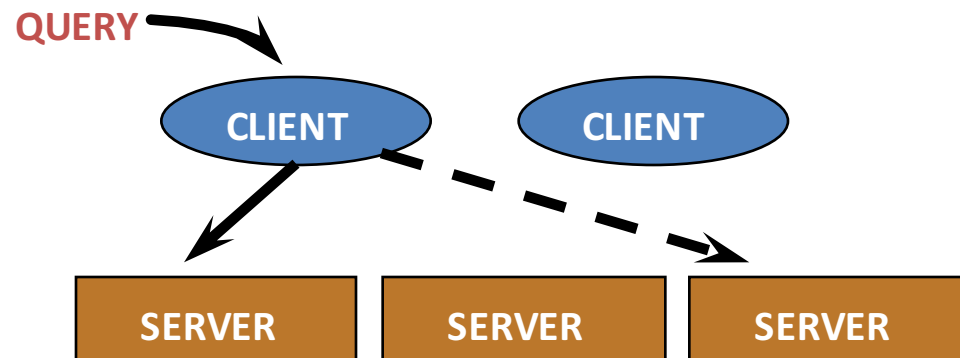
Distributed DBMS Architecture

Distributed DBMS Architectures

- Three alternative approaches
 1. Client-Server
 2. Collaborating Server
 3. Middleware

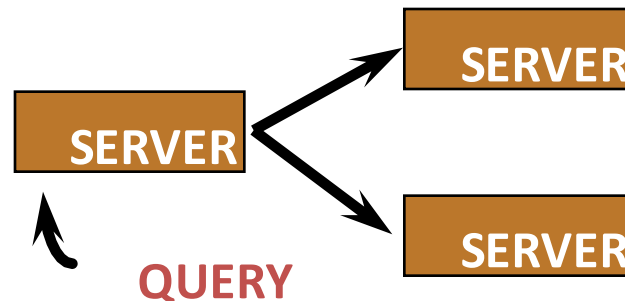
Client-Server Systems

- One or more client (e.g. personal computer) and one or more server processes (e.g. a mainframe)
 - A client process can ship a query to any server process
 - Clients are responsible for user interfaces
 - Server manages data and executes queries
- Advantages
 - clean separation and centralized server
 - expensive server machines are not underutilized
 - users can run GUI on clients that they are familiar with
- Challenge
 - need to carefully handle communication costs
 - e.g. fetching tuples one at a time might be bad – need to do caching on client side



Collaborating Server Systems

- Queries can span multiple sites
 - not allowed in client-servers as the clients would have had to break queries and combine the results
- When a server receives a query that requires access to data at other servers
 - it generates appropriate subqueries
 - puts the result together



Middleware Systems

- Allows a single query to span multiple servers
- But does not require all db servers to be capable of handling multi-site execution strategies
 - need just one db server capable of managing queries and transactions spanning multiple servers (called **middleware**)
 - the remaining servers can handle only the local queries and transactions
- The middleware layer is capable of executing joins and other operations on data obtained from other servers, but typically does not maintain any data
- Useful when trying to integrate several “legacy systems”
 - whose basic capabilities cannot be extended

Storing Data in Distributed DBMS

Storing Data in a Distributed DBMS

- Relations are stored across several sites
- Accessing data at a remote site incurs message-passing costs
- To reduce this overhead, a single relation may be **partitioned** or **fragmented** across several sites
 - typically at sites where they are most often accessed
- The data can be **replicated** as well
 - when the relation is in high demand

Fragmentation

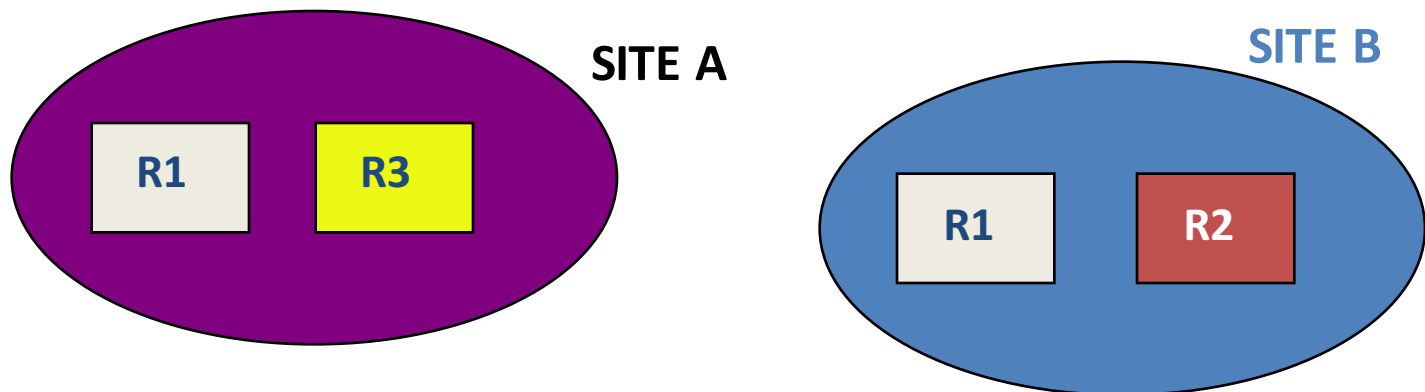
- Break a relation into smaller relations or fragments
 - store them in different sites as needed

TID					
t1					
t2					
t3					
t4					

- **Horizontal:**
 - Usually disjoint
 - Can often be identified by a **selection query** (employees in a city)
 - To retrieve the full relation, need a union
- **Vertical:**
 - Identified by **projection queries**
 - Typically unique TIDs added to each tuple
 - TIDs replicated in each fragments
 - Ensures that we have a **Lossless Join**

Replication

- When we store several copies of a relation or relation fragments
 - can be replicated at one or more sites
 - e.g. R is fragmented into R1, R2, R3; one copy of R2, R3; but two copies at R1 at two sites
- Advantages
 - Gives increased availability – e.g. when a site goes down
 - Faster query evaluation – e.g. using a local copy
- Synchronous and Asynchronous (later)
 - Vary in how current different copies are when a relation is modified



Distributed Catalog Management

- Must keep track of how data is fragmented and replicated across sites
 - in addition to usual schema, authorization, and statistical information
- Must be able to uniquely identify each replica of each fragment
 - Globally unique name may compromise autonomy of servers
 - To preserve local autonomy: **Global relation name = <local-name, birth-site>**
 - To identify a replica, add a **replica-id** field (now called **global replica name**)
- **Site Catalog:** Describes all objects (fragments, replicas) at a site + Keeps track of replicas of relations created at this site
 - To find a relation, look up its birth-site catalog
 - Birth-site never changes, even if relation is moved

Distributed Query Processing

No joins
Join

Non-Join Distributed Queries

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
      AND S.rating < 7
```

tid	sid	sname	rating	age
T1			4	
T2			5	
T3			9	

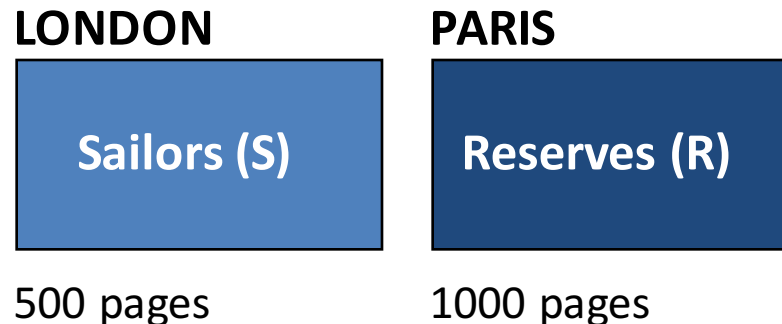
stored at **Shanghai**

stored at **Tokyo**

- **Horizontally Fragmented:** Tuples with rating < 5 at Shanghai, >= 5 at Tokyo.
 - Must compute SUM(age), COUNT(age) at both sites.
 - If WHERE contained just S.rating > 6, just one site
- **Vertically Fragmented:** sid and rating at Shanghai, sname and age at Tokyo, tid at both.
 - Must reconstruct relation by join on tid, then evaluate the query
 - if no tid, decomposition would be lossy
- **Replicated:** Sailors copies at both sites.
 - Choice of site based on local costs (e.g. index), shipping costs

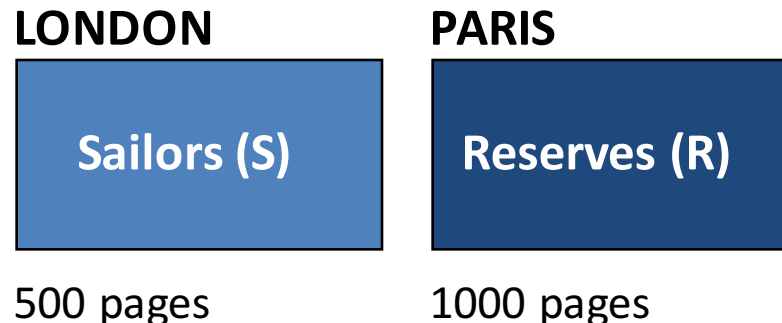
Joins in a Distributed DBMS

- Can be very expensive if relations are stored at different sites
1. Fetch as needed
 2. Ship to one site
 3. Semi-join
 4. Bloom join



1. Fetch As Needed

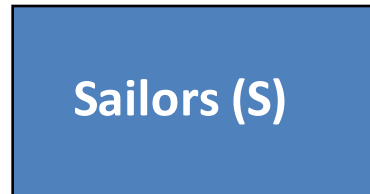
- Page-oriented Nested Loop Join
 - Sailors as outer – for each S page, fetch all R pages from Paris
 - if cached at London, each R page fetched once
 - Otherwise, **Cost: $500d + 500 * 1000(d+s)$**
 - **d** is cost to read/write page; **s** is cost to ship page
 - If query was not submitted at London, must add cost of shipping result to query site
 - Can also do Index NL at London, fetching matching Reserves tuples to London as needed



2. Ship To One Site

- Ship Sailors (S) to Paris
 - **Cost:** $500(2d + s) + 4500d$
 - For relation S: reading in London, shipping to Paris, and saving it in Paris: $500(2d + s)$
 - Assume Sort-Merge Join with cost $3(M+N)$, i.e. enough memory
 - Then join cost = $3*(500+1000)d$
 - If result size is very large, may be better to ship both relations to result site and then join them
- Not all tuples in S join with a tuple in R
 - unnecessary shipping
 - solution: Semi-join

LONDON



500 pages

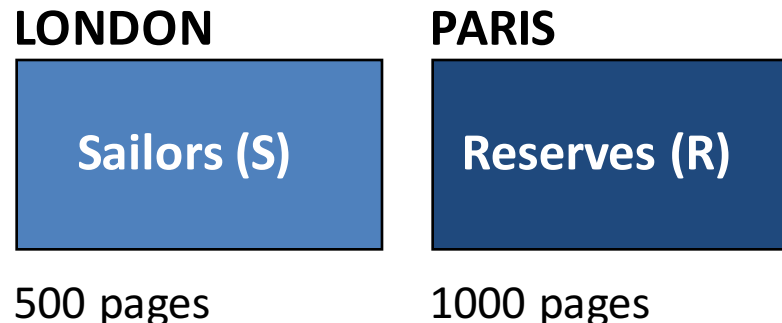
PARIS



1000 pages

3. Semijoin

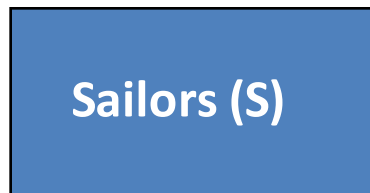
- Suppose want to ship R to London and then do join with S at London. Instead,
 1. **At London**, project S onto join columns and ship this to Paris
 - Here foreign keys, but could be arbitrary join
 2. **At Paris**, join S-projection with R
 - Result is called **reduction** of Reserves w.r.t. Sailors (only these tuples are needed)
 3. Ship reduction of R to back to London
 4. **At London**, join S with reduction of R
- Tradeoff the cost of computing and shipping projection for cost of shipping full R relation
- Especially useful if there is a selection on Sailors, and answer desired at London



4. Bloomjoin

- Similar idea like semi-join
- Suppose want to ship R to London and then do join with S at London. Instead,
 1. **At London**, compute a bit-vector of some size k:
 - Hash join column values into range 0 to k-1
 - If some tuple hashes to p, set bit p to 1 (p from 0 to k-1)
 - Ship bit-vector to Paris
 2. **At Paris**, hash each tuple of R similarly
 - discard tuples that hash to 0 in S's bit-vector
 - Result is called **reduction** of R w.r.t S
 3. Ship bit-vector reduced R to London
 4. **At London**, join S with reduced R
- Bit-vector cheaper to ship, almost as effective
 - the size of the reduction of R shipped back can be larger. Why?

LONDON



500 pages

PARIS



1000 pages

Distributed Query Optimization

- Cost-based approach
 - consider all plans
 - pick cheapest
- Similar to centralized optimization, but have differences
 1. Communication costs must be considered
 2. Local site autonomy must be respected
 3. New distributed join methods
- Query site constructs global plan, with suggested local plans describing processing at each site
 - If a site can improve suggested local plan, free to do so

Distributed transactions

Updating Distributed Data

Synchronous
Asynchronous

Updating Distributed Data

- **Synchronous Replication:** All copies of a modified relation (or fragment) must be updated before the modifying transaction commits
 - Data distribution is made “transparent” (not visible!) to users
- **Asynchronous Replication:** Copies of a modified relation are only periodically updated; different copies may get out of sync in the meantime
 - Users must be aware of data distribution
 - More efficient – many current products follow this approach

Synchronous Replication

- **Voting:** transaction must write **a majority of copies** to modify an object; must read enough copies to be sure of seeing at least one most recent copy
 - E.g., 10 copies; 7 written for update; 4 copies read (why?)
 - Each copy has version number – copy with the highest version number is current
 - Not attractive usually because reads are common
- **Read-any Write-all:** **Read any copy, Write all copies**
 - Writes are slower and reads are faster, relative to Voting
 - Most common approach to synchronous replication
- **Choice of technique determines which locks to set**

Cost of Synchronous Replication

- Before an update transaction can commit, it must obtain locks on all modified copies
 - Sends lock requests to remote sites, and while waiting for the response, holds on to other locks
 - If sites or links fail, transaction cannot commit until they are back up
 - Even if there is no failure, committing must follow an expensive **commit protocol** with many messages
- So the alternative of asynchronous replication is becoming widely used

Asynchronous Replication

- Allows modifying transaction to commit before all copies have been changed
 - readers nonetheless look at just one copy
 - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time
- Two approaches: Primary Site and Peer-to-Peer replication
 - Difference lies in how many copies are “updatable” or “master copies”

Peer-to-Peer Replication

- More than one of the copies of an object can be a master
- Changes to a master copy must be propagated to other copies somehow
- If two master copies are changed in a conflicting manner, **conflict resolution** needed
 - e.g., Site 1: Joe's age changed to 35; Site 2: to 36
- Best used when conflicts do not arise:
 - E.g., Each master site owns a disjoint fragment
 - E.g., Updating rights held by one master at a time – then propagated to other sites

Primary Site Replication

- Exactly one copy of a relation is designated the **primary** or **master copy**
 - Replicas at other sites cannot be directly updated
 - The primary copy is published
 - Other sites subscribe to this relation (or its fragments)
 - These are **secondary copies**
- Main issue: How are changes to the primary copy propagated to the secondary copies?
 - Done in two steps
 - First, “capture” changes made by committed transactions
 - Then, “apply” these changes
 - more details in the [RG] book (optional reading)

Data Warehousing and Replication

- **A hot trend:** Building giant “warehouses” of data from many sites
 - Enables complex **decision support queries** over data from across an organization
- **Warehouses can be seen as an instance of asynchronous replication**
 - Source data typically controlled by different DBMSs
 - regularly clean data and remove mismatches (USD vs. EURO) while creating replicas
 - **OLAP and data warehousing will be covered later**

Distributed Transactions

Distributed CC
Distributed Recovery

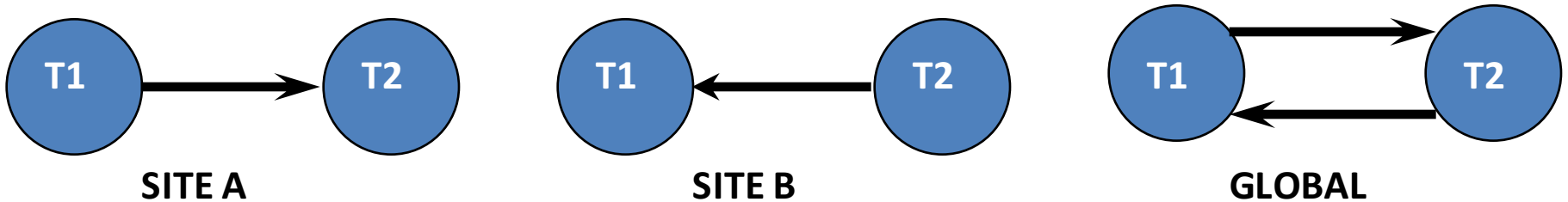
Distributed Transactions

- **Distributed CC**
 - How can locks for objects stored across several sites be managed?
 - How can deadlocks be detected in a distributed database?
- **Distributed Recovery**
 - When a transaction commits, all its actions, across all the sites at which it executes must persist
 - When a transaction aborts, none of its actions must be allowed to persist

Distributed Locking

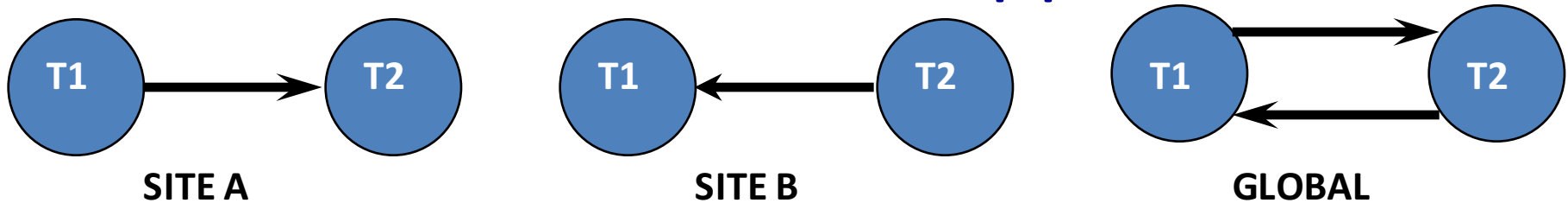
- How do we manage locks for objects across many sites?
 1. **Centralized:** One site does all locking
 - Vulnerable to single site failure
 2. **Primary Copy:** All locking for an object done at the primary copy site for this object
 - Reading requires access to locking site as well as site where the object copy is stored
 - i.e. reading requires communication between sites (slow read is bad)
 3. **Fully Distributed:** Locking for a copy done at site where the copy is stored
 - Locks at all sites while writing an object (unlike previous two)

Distributed Deadlock Detection



- Each site maintains a **local waits-for graph**
- A global deadlock might exist even if the local graphs contain no cycles
- Further, **phantom deadlocks** may be created while communicating
 - due to delay in propagating local information
 - might lead to unnecessary aborts

Three Distributed Deadlock Detection Approaches



1. Centralized

- send all local graphs to one site periodically
- A global waits-for graph is generated

2. Hierarchical

- organize sites into a hierarchy and send local graphs to parent in the hierarchy
- e.g. sites (every 10 sec) → sites in a state (every min) → sites in a country (every 10 min) → global waits for graph
- intuition: more deadlocks are likely across closely related sites

3. Timeout

- abort transaction if it waits too long (low overhead)

Distributed Recovery

- Two new issues:
 - New kinds of failure, e.g., links and remote sites
 - If “sub-transactions” of a transaction execute at different sites, all or none must commit
 - Need a **commit protocol** to achieve this
 - Most widely used: **Two Phase Commit (2PC)**
- A log is maintained at each site
 - as in a centralized DBMS
 - commit protocol actions are additionally logged

Two Phase Commit (2PC)

Two-Phase Commit (2PC)

- Site at which transaction originates is **coordinator**
- Other sites at which it executes are **subordinates**
 - w.r.t. coordination of this transaction

Example on whiteboard

When a transaction wants to commit – 1/5

1. Coordinator sends **prepare** message to each subordinate

When a transaction wants to commit – 2/5

2. Subordinate receives the prepare message

- a) decides whether to abort or commit its subtransaction
- b) force-writes an **abort** or **prepare** log record
- c) then sends a **no** or **yes** message to coordinator

When a transaction wants to commit – 3/5

3. If coordinator gets unanimous **yes** votes from all subordinates

- a) it force-writes a **commit** log record
- b) then sends **commit** message to all subs

Else (if receives a no message or no response from some subordinate),

- a) it force-writes **abort** log record
- b) then sends **abort** messages

When a transaction wants to commit – 4/5

4. Subordinates force-write **abort/commit** log record based on message they get
 - a) then send **ack** message to coordinator
 - b) If commit received, commit the subtransaction
 - c) write an **end** record

When a transaction wants to commit – 5/5

5. After the coordinator receives ack from all subordinates,
 - writes **end** log record

Transaction is officially committed when the coordinator's commit log record reaches the disk

- subsequent failures cannot affect the outcomes

Comments on 2PC

end of Lecture 19

- Two rounds of communication
 - first, **voting**
 - then, **termination**
 - Both initiated by coordinator
- Any site (coordinator or subordinate) can unilaterally decide to abort a transaction
 - but unanimity/consensus needed to commit
- Every message reflects a decision by the sender
 - to ensure that this decision survives failures, it is first recorded in the local log and is force-written to disk
- All commit protocol log records for a transaction contain tid and Coordinator-id
 - The coordinator's abort/commit record also includes ids of all subordinates.

Restart After a Failure at a Site – 1/4

continue in Lecture 20

- Recovery process is invoked after a sites comes back up after a crash
 - reads the log and executes the commit protocol
 - the coordinator or a subordinate may have a crash
 - one site can be the coordinator some transaction and subordinates for others

Restart After a Failure at a Site – 2/4

- If we have a **commit** or **abort** log record for transaction T, but not an end record, must redo/undo T respectively
 - If this site is the coordinator for T (from the log record), keep sending **commit/abort** messages to subs until **acks** received
 - then write an **end** log record for T

Restart After a Failure at a Site – 3/4

- If we have a **prepare** log record for transaction T, but not **commit/abort**
 - This site is a subordinate for T
 - Repeatedly contact the coordinator to find status of T
 - Then write **commit/abort** log record
 - Redo/undo T
 - and write **end** log record

Restart After a Failure at a Site – 4/4

- If we don't have even a **prepare** log record for T
 - T was not voted to commit before crash
 - unilaterally abort and undo T
 - write an end record
- No way to determine if this site is the coordinator or subordinate
 - If this site is the coordinator, it might have sent prepare messages
 - then, subs may send yes/no message – coordinator is detected – ask subordinates to abort

Blocking

- If coordinator for transaction T fails, subordinates who have voted **yes** cannot decide whether to commit or abort T until coordinator recovers.
 - T is **blocked**
 - Even if all subordinates know each other (extra overhead in **prepare** message) they are blocked unless one of them voted **no**
- Note: even if all subs vote yes, the coordinator then can give a no vote, and decide later to abort!

Link and Remote Site Failures

- If a remote site does not respond during the commit protocol for transaction T, either because the site failed or the link failed:
 - If the current site is the coordinator for T, should abort T
 - If the current site is a subordinate, and has not yet voted **yes**, it should abort T
 - If the current site is a subordinate and has voted **yes**, it is blocked until the coordinator responds
 - needs to periodically contact the coordinator until receives a reply

Observations on 2PC

- Ack messages used to let coordinator know when it can “forget” a transaction; until it receives all acks, it must keep T in the transaction Table
- If coordinator fails after sending **prepare** messages but before writing **commit/abort** log records, when it recovers, it aborts the transaction
- If a subtransaction does no updates, its commit or abort status is irrelevant

2PC with Presumed Abort

- When coordinator aborts T, it undoes T and removes it from the transaction Table immediately
 - Doesn't wait for **acks**
 - “presumes abort” if transaction not in transaction Table.
- Subordinates do not send acks on **abort**
- If subtransaction **does not do updates**, it responds to **prepare** message with a **reader message** instead of yes/no.
- Coordinator treats reader as yes
 - subsequently ignores readers
- If all subtransactions are readers, 2nd phase not needed

Three-Phase Commit

- Can avoid blocking even if the coordinator fails during recovery
- **prepare -> precommit -> commit**
- Coordinator sends out prepare and receives yes votes from all subs
 - as before
- Then it sends a **precommit** message (instead of commit)
 - new step – subs now send back acks
- when a sufficient no. of acks received (more than the max no. of failures that must be handled)
 - coordinator force-writes commit log records + sends a commit message to all subs
- Coordinator postpones the decision to commit
 - until it is sure that enough subs are aware of the decision
- **If the coordinator fails**
 - these sites can communicate with each other and detect that the transaction must be committed, otherwise aborts the transaction
 - no need to wait for the coordinator to recover
- **But additional cost during normal operation**
 - not often used in practice