Chapters 21-23 : Distributed Databases

Sistemas de Bases de Dados 2020/21

Capítulo refere-se a: Database System Concepts, 7th Ed

Distributed Databases

- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - The goal is to provide a view of a single database, hiding details of distribution
  - Done for improving (local) efficiency, improving availability, …
- Heterogeneous distributed databases
  - Different software/schema on different sites
  - The goal is to integrate existing databases to provide useful functionality
  - The various databases may already exist.
- In distributed databases two types of transactions exist:
  - A **local transaction** accesses data in the *single* site at which the transaction was initiated.
  - A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

Distributed Data Storage

- Data Storage can be distributed by replicating data or be fragmenting data.

- **Replication**
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
- **Fragmentation**
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.

Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- **Full replication** of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (closer to user), governmental regulations
- Latency of replication across geographically distributed data centers is much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Data Replication

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* if replicas exist.
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel.
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*.
- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- Division of relation r into fragments $r_1, r_2, \ldots, r_n$ which contain sufficient information to reconstruct relation r.
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments
  - The original relation is obtained by the **union** of the fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schema must contain a common candidate key (or superkey) to ensure lossless join property
    - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key
  - The original relation is obtained by the **join** of the fragments
- Examples:
  - Horizontal fragmentation of an account relation, by branches
  - Vertical fragmentation of an employer relation, to separate the data for e.g. salaries, functions, etc

Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed
  - Fragments may be successively fragmented to an arbitrary depth
  - An examples is to horizontally fragment an account relation by branches, and vertically fragment it to *hide* balances

Distributed Query Processing

Data Integration From Multiple Sources

- Many database applications require data from multiple databases
- A **federated database system** is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - Creates an illusion of logical database integration without any physical database integration
  - Each database has its **local schema**
  - **Global schema** integrates all the local schema
    - **Schema integration**
  - Queries can be issued against global schema, and translated to queries on local schemas
    - Databases that support common schema and queries, but not updates, are referred to as **mediator** systems

Data Integration From Multiple Sources

- **Data virtualization**
  - Allows data access from multiple databases, but without a common schema
- **External data** approach allows database to treat external data as a database relation (**foreign tables**)
  - Many databases today allow a local table to be defined as a view on external data
  - SQL Management of External Data (SQL MED) standard
- **Wrapper** for a data source is a view that translates data from local to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema

Schema and Data Integration

- **Schema integration**: creating a unified conceptual schema
  - Requires creation of **global schema**, integrating several **local schema**
- **Global-as-view approach**
  - At each site, create a view of local data, mapping it to the global schema
  - Union of local views is the global view
  - Good for queries, but not for updates
    - E.g., which local database should an insert go to?
- **Local-as-view approach**
  - Create a view defining contents of local data as a view of global data
    - Site stores local data as before, the view is for update processing
  - Updates on global schema are mapped to updates to the local views

Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g., data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- Agreement on units of measure

Unified View of Data (Cont.)

- Variations in names
  - E.g., Köln vs Cologne, Mumbai vs Bombay
- One approach: globally unique naming system
  - E.g., GeoNames database (www.geonames.org)
- Another approach: specification of name equivalences
  - E.g., used in the Linked Data project supporting integration of a large number of databases storing data in RDF data

Query Processing Across Data Sources

- Several issues in query processing across multiple sources
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g., web forms, flat file data sources
  - Queries must be broken up and processed partly at the source and partly at a different site
- Removal of duplicate information when sites have overlapping information
  - Decide which sites to execute query
- Global query optimization

Join Locations and Join Ordering

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $r1 \bowtie r2 \bowtie \mathrm{r3}$
- $r1$ is stored at site S_1
- $r2$ at S_2
- $r3$ at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire query locally at site S_I.
  - Ship a copy of the *r1* relation to site S_2 and compute $temp_1 = r1 \bowtie r2$ *at* S_2.
  - Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie r3$ at S_3
  - Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stores at site S_1

  Let r_2 be a relation with schema R_2 stores at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1.
  1. Compute $temp_1 \leftarrow \prod_{R1 \cap R2}$ (r1) at $S1$.
  2. Ship $temp_1$ from S_1 to S_2.
  3. Compute $temp_2 \leftarrow r_2 \bowtie$ temp1 at S_2
  4. Ship $temp_2$ from S_2 to S_1.
  5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$.

Semijoin Reduction

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

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

- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 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.
- Semijoin can be computed approximately by using a Bloom filter
  - For each tuple of r_2 compute hash value on join attribute; if hash value is i, and set bit i of the bitmap
  - Send bitmap to site containing r_1
  - Fetch only tuples of r_1 whose join attribute value hashes to a bit that is set to 1 in the bitmap
  - Bloom filter is an optimized bitmap filter structure

Distributed Query Optimization

- New physical property for each relation: location of data
- Operators also need to be annotated with the site where they are executed
  - Operators typically operate only on local data
  - Remote data is typically fetched locally before operator is executed
- Optimizer needs to find best plan taking data location and operator execution location into account.

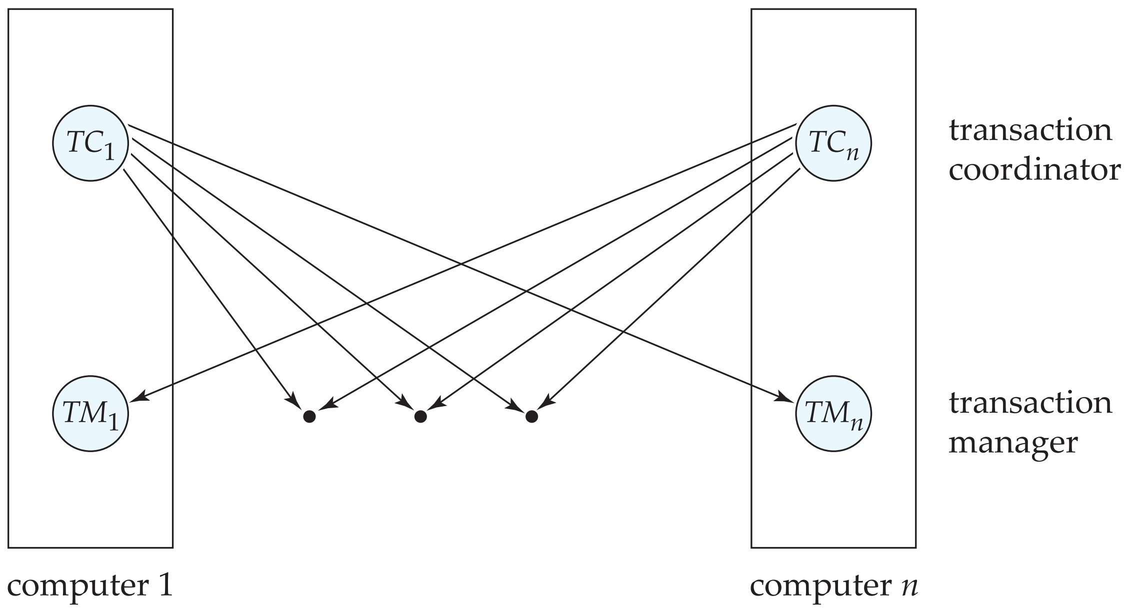
Distributed Transactions

Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

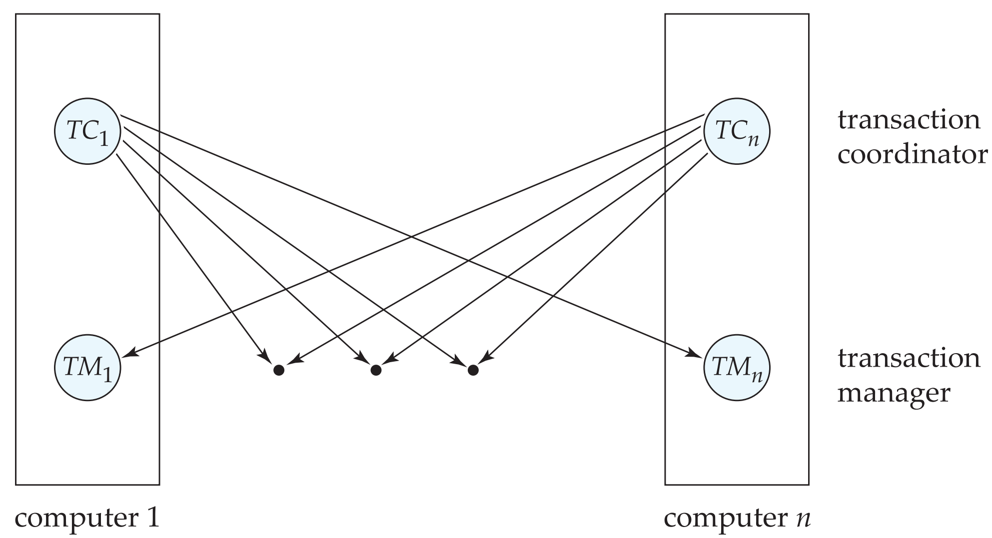
Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local **transaction manager**
  - Each site has a **transaction coordinator**
    - Global transactions submitted to any transaction coordinator



Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - transaction must be committed at all sites or aborted at all sites.
- Each local transaction manager is responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the execution and commit/abort of the transactions executing at that site.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites or aborted at all the sites.
    - cannot have transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- *Three-phase commit* (3PC) protocol avoids some drawbacks of 2PC, but is more complex
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
  - More on these later
- These protocols assume **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

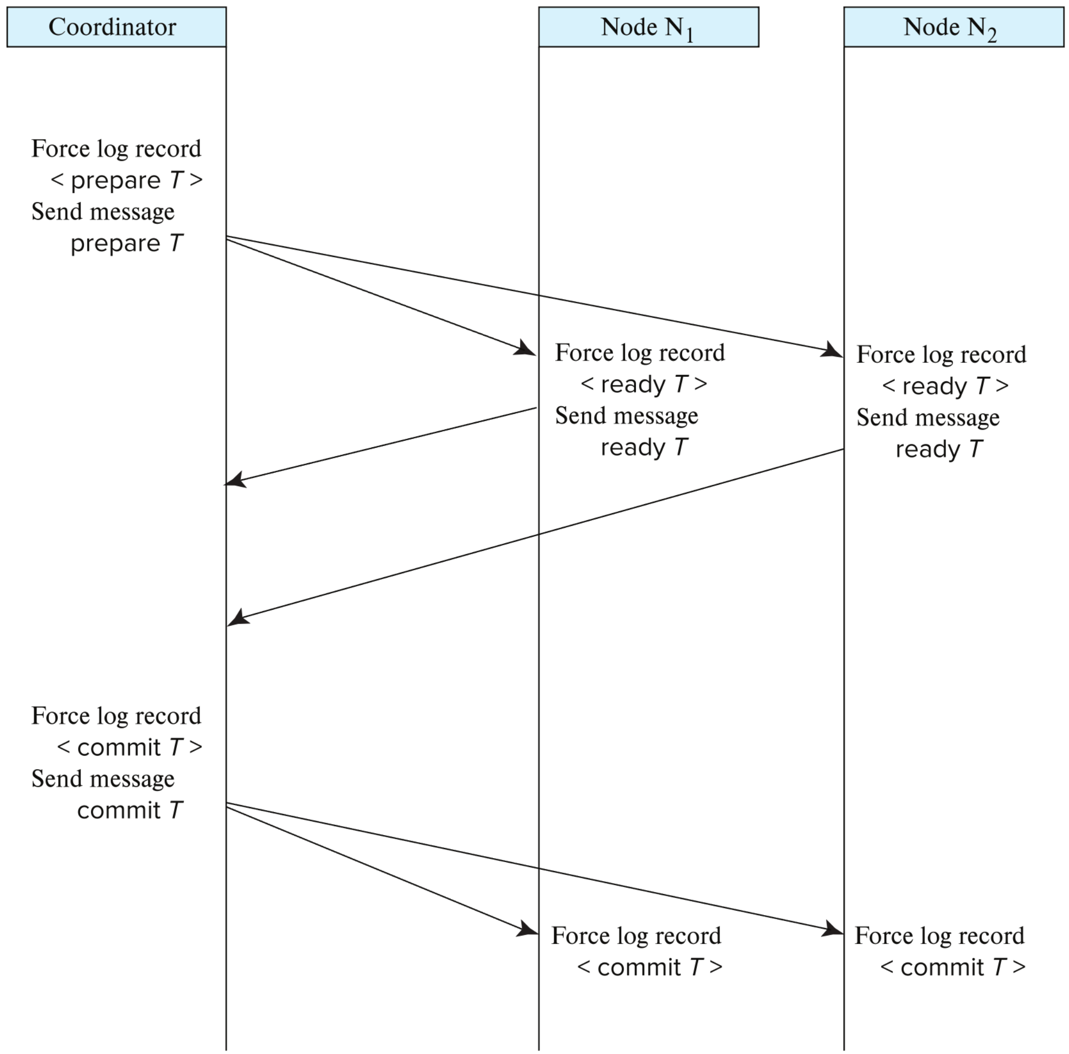
- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving this message, the transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send **ready** *T* message to C_i

  Transaction is now in ready state at the site

Phase 2: Recording the Decision

- *T* can be committed if C_i received a **ready** *T* message from all the participating sites: otherwise, *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record is in stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Two-Phase Commit Protocol



Handling of Failures - Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing, then participating sites must decide on *T*' s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. So, it can abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites that are in the same partition as the coordinator think that the sites in the other partition have failed and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*> *L* = list of locks held by *T* when the log is written (read locks can be omitted).
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value for the even if some nodes fail during the execution of the protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a majority of the nodes participating remain alive and can communicate with each other

Three-Phase Commit

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
- Phase 2 of 2PC is split into 2 phases, Phase 2 and Phase 3 of 3PC
  - In phase 2 coordinator makes a decision as in 2PC (called the **pre-commit** decision) and records it in multiple (at least K) sites
  - In phase 3, coordinator sends commit/abort message to all participating sites,
- Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure
  - Avoids blocking problem as long as < K sites fail
- Drawbacks:
  - higher overheads
  - assumptions may not be satisfied in practice

Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

Single-Lock-Manager Approach

- In the **single lock-manager** approach, lock manager runs on a *single* chosen site, say S_i
  - All lock requests sent to central lock manager
- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure.

Distributed Lock Manager

- In the **distributed lock-manager** approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
  - Locking is performed separately on each site accessed by transaction
    - Every replica must be locked and updated
    - But special protocols may be used for replicas (more on this later)
- Advantage: work is distributed and can be made robust to failures
- Disadvantage:
  - Possibility of a global deadlock without local deadlock at any single site
  - Lock managers must cooperate for deadlock detection

Deadlock Handling

Consider the following two transactions and history, with item X and transaction T_1 at site 1, and item Y and transaction T_2 at site 2:

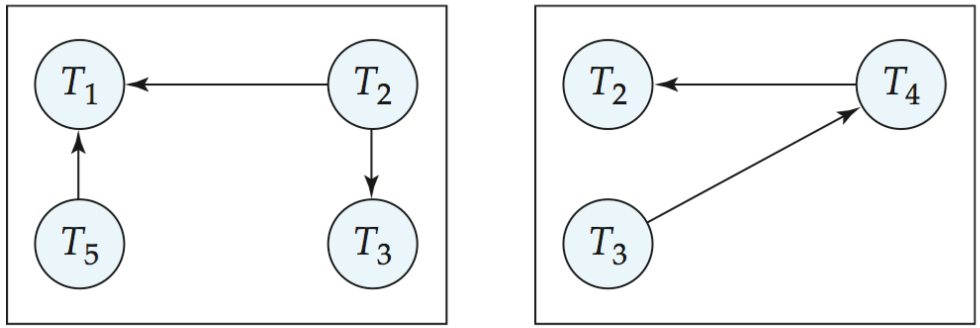
T_1: write (X) write (Y)	T_2: write (X) write (Y)
X-lock on X write (X)	
	X-lock on Y write (Y) wait for X-lock on X
Wait for X-lock on Y	

Result: deadlock which cannot be detected locally at either site

Deadlock Detection

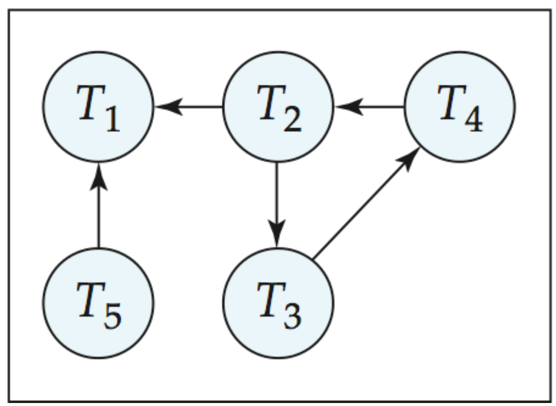
- In the **centralized deadlock-detection** approach, a global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.
  - a number of changes have occurred in a local wait-for graph.
  - the coordinator needs to invoke cycle-detection.
- If the coordinator finds a cycle, it selects a victim and notifies all sites. The sites roll back the victim transaction.

Local and Global Wait-For Graphs



site S_1 site S_2

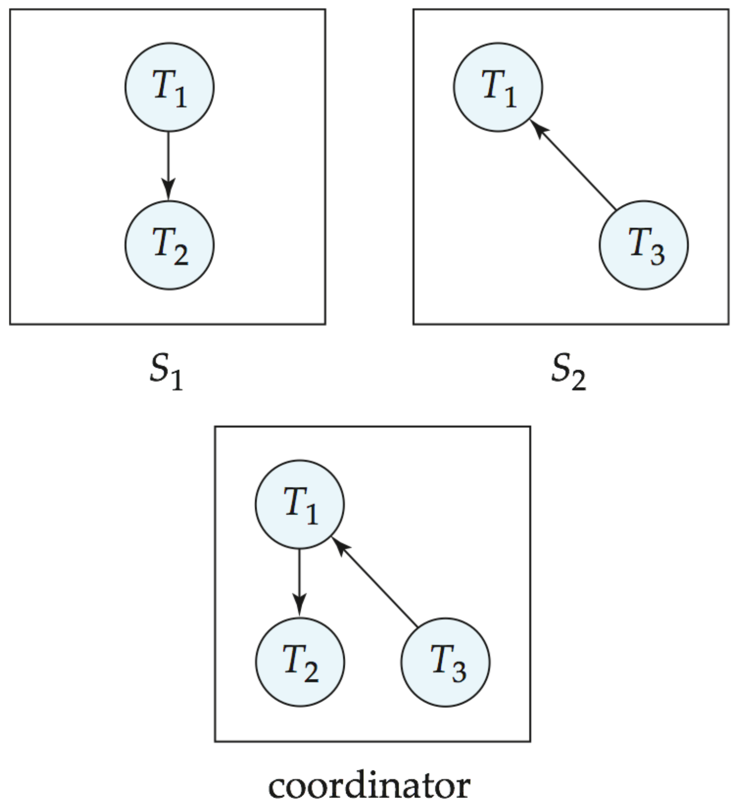
Local



Global

Example Wait-For Graph for False Cycles

Initial state:



False Cycles (Cont.)

- Suppose that starting from the state shown in figure,
  1. T_2 releases resources at S_1
     - resulting in a message remove $T_1 \rightarrow T_2$ message from the Transaction Manager at site S_1 to the coordinator)
  2. And then T_2 requests a resource held by T_3 at site S_2
     - resulting in a message insert $T_2 \rightarrow T_3$ from S_2 to the coordinator
- Suppose further that the insert message reaches before the **delete** message
  - this can happen due to network delays
- The coordinator would then find a false cycle

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

- The false cycle above never existed in reality.
- False cycles cannot occur if two-phase locking is used.

Distributed Deadlocks

- Unnecessary rollbacks may result
  - When deadlock has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.
  - Due to false cycles in the global wait-for graph; however, likelihood of false cycles is low.
- In the **distributed deadlock-detection** approach, sites exchange wait-for information and check for deadlocks
  - Expensive and not used in practice

Leases

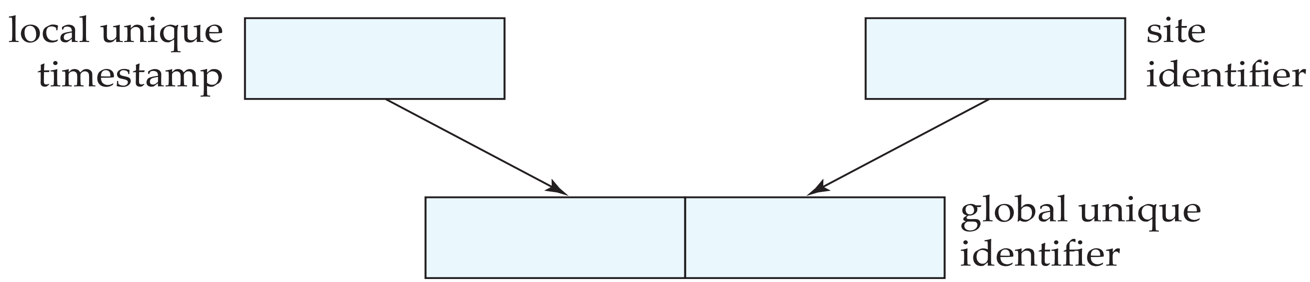
- A **lease** is a lock that is granted for a specific period of time
- If a process needs a lock even after expiry of lease, process can **renew** the lease
- But if renewal is not done before end time of lease, the lease **expires,** and lock is released
- Leases can be used if there is only one coordinator for a protocol at any given time
  - Coordinator gets a lease and renews it periodically before expire
  - If coordinator dies, lease will not be renewed and can be acquired by backup coordinator

Leases (Cont.)

- Coordinator must check that it still has lease when performing action
  - Due to delay between check and action, must check that expiry is at least some time t' into the future
    - t' includes delay in processing and maximum network delay
    - Old messages must be ignored
- Leases depend on clock synchronization

Distributed Timestamp-Based Protocols

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a *unique* timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - Each site generates a unique local timestamp using either a logical counter or the local clock.
  - Global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier.



Distributed Timestamps

- A node with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: "disadvantages" transactions
- To fix this problem
  - Keep clocks synchronized using network time protocol
  - Or, define within each node N_i a **logical clock** (LC_i), which generates the unique local timestamp
    - Require that N_i advance its logical clock whenever a request is received from a transaction Ti with timestamp $< x,y >$ and x is greater that the current value of LC_i.
    - In this case, site N_i advances its logical clock to the value $x + 1$

Distributed Timestamp Ordering

- Centralized TSO and multiversion TSO easily extended to distributed setting
  - Transactions use a globally unique timestamp
  - Each site that performs a read or write performs same checks as in centralized case
- Clocks at sites should be synchronized
  - Otherwise a transaction initiated at a site with a slower clock may get restarted repeatedly.

Distributed Validation

- The validation protocol used in centralized systems can be extended to distributed systems
- Start/validation/finish timestamp for a transaction T_i may be issued by any of the participating nodes
  - Must ensure StartTS(T_i) < TS(T_i) < FinishTS(T_i)
- Validation for T_i is done at each node that performed read/write
  - Validation checks for transaction T_i are same as in centralized case
    - Ensure that no transaction that committed after T_i started has updated any data item read by T_i.
  - A key difference from centralized case is that may T_i start validation after a transaction with a higher validation timestamp has already finished validation
    - In that case T_i is rolled back

Distributed Validation (Cont.)

- Two-phase commit (2PC) needed to ensure atomic commit across sites
  - Transaction is validated, then enters prepared state
  - Writes can be performed (and transaction finishes) only after 2PC makes a commit decision
  - If transaction T_i is in prepared state, and another transaction T_k reads old value of data item written by T_i, T_k will fail if T_i commits
    - Can make the read by T_k wait, or create a commit dependency for T_k on T_i.

Distributed Validation (Cont.)

- Distributed validation is not widely used, but optimistic concurrency control without read-validation is widely used in distributed settings
  - Version numbers are stored with data items
  - Writes performed at commit time ensure that the version number of a data item is same as when data item was read
  - Hbase supports atomic checkAndPut() as well as checkAndMutate() operations; see book for details