Chapters 21-23 : Distributed Databases

Sistemas de Bases de Dados 2021/22

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 a 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, …, 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 a 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 the same relation/attribute
  - The 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

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.

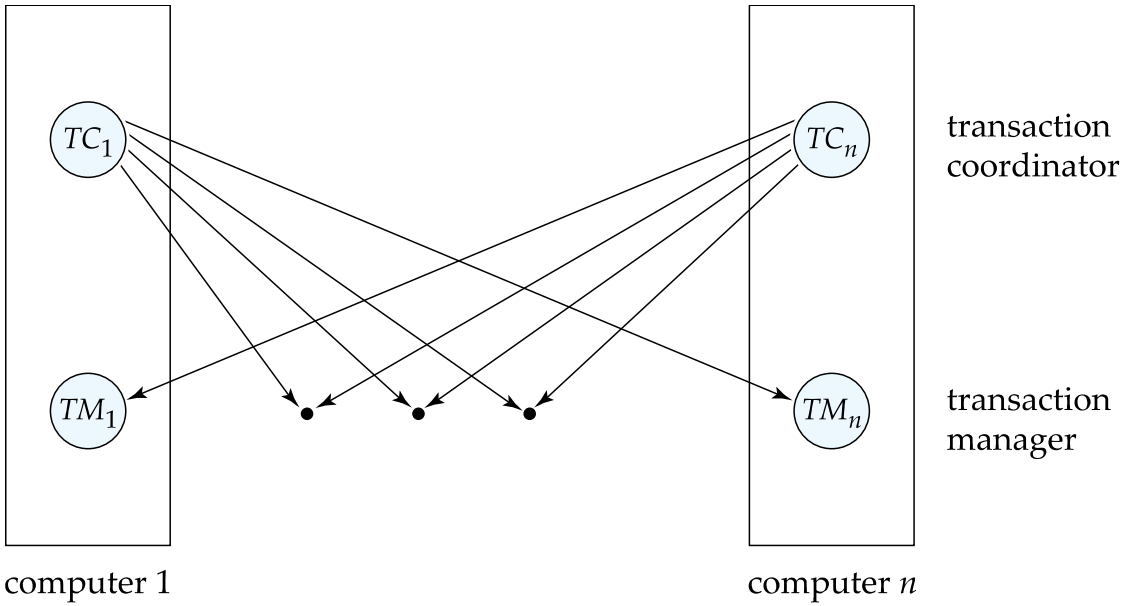
Distributed Transactions

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

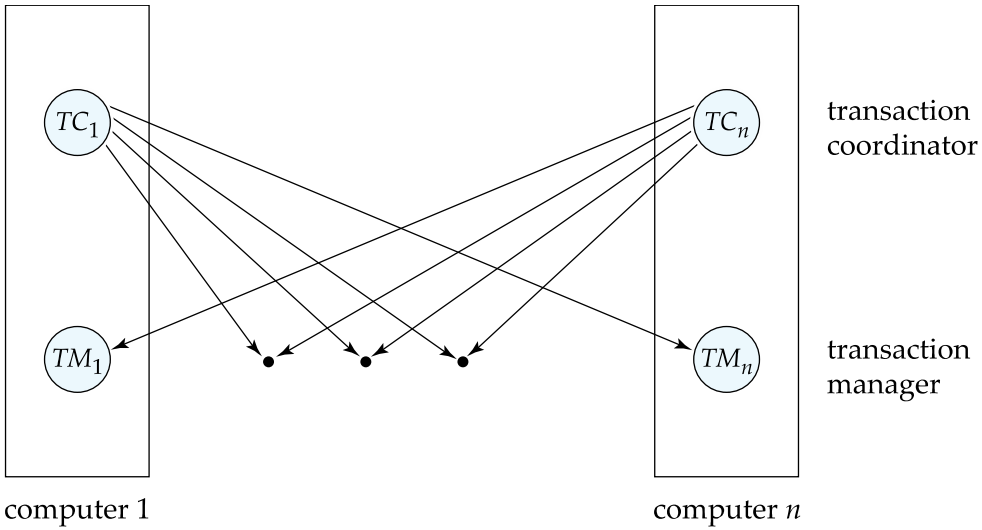
Distributed Transactions

- 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 messages
    - 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

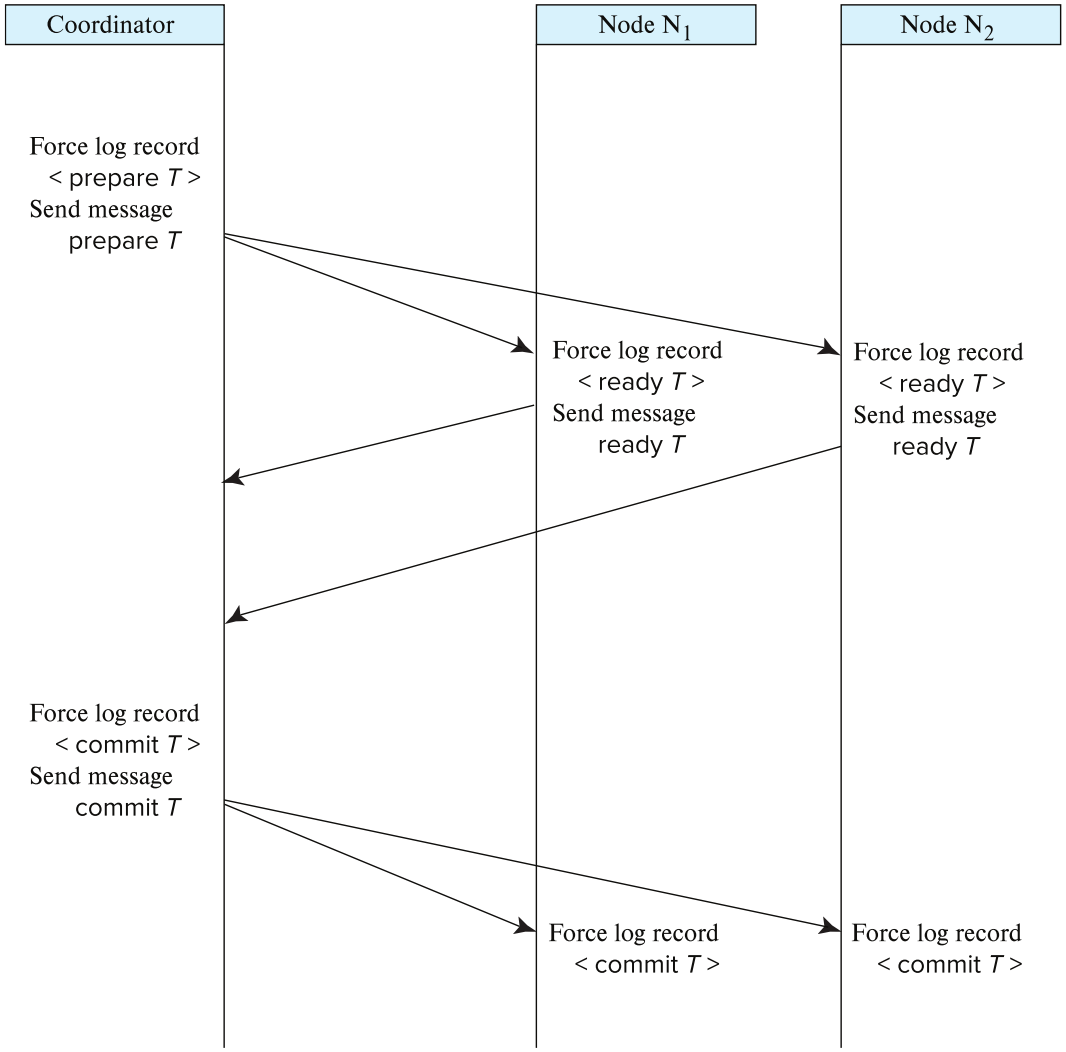
- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving this message, the transaction manager at site determines 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 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

Replication

Consistency of Replicas

- Consistency of replicas
  - Ideally: all replicas should have the same value ➔ updates performed at all replicas
    - But what if a replica is not available (disconnected, or failed)?
  - Suffices if reads get correct value, even if some replica is out of date
  - Above idea formalized by **linearizability**: given a set of read and write operations on a (replicated) data item
    - There must be a linear ordering of operations such that each read sees the value written by the most recent preceding write
- Note that linearizability only addresses a single (replicated) data item; serializability is orthogonal

Consistency of Replicas

- Cannot differentiate node failure from network partition in general
  - Backup coordinator should takeover if primary has failed
  - Use multiple independent links, so single link failure does not result in partition, but it is possible that all links have failed
- Protocols that require all copies to be updated are not robust to failure
- We'll see techniques that can allow continued processing during failures, whether node failure or network partition
  - Key idea: decisions are made based on successfully writing/reading majority
- Alternative: **asynchronous replication:** commit after performing update on a *primary copy* of the data item, and update replicas *asynchronously*
  - Lower overheads, but risk of reading stale data, or lost updates on primary failure

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:

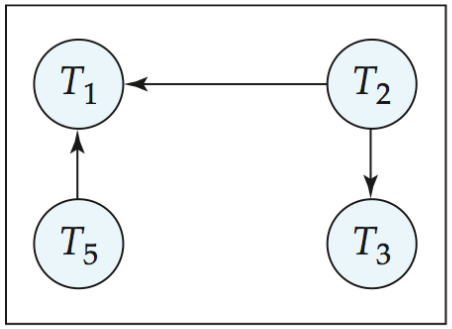
T_1: write (X) write (Y)	T_2: write (X) write (Y)
X-lock on X write (X)	
	X-lock on Y write (Y) wait for X-lock on X
Wait for X-lock on Y	

Result: deadlock which cannot be detected locally at either site

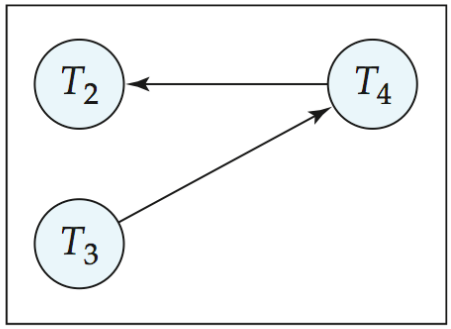
Deadlock Detection

- 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

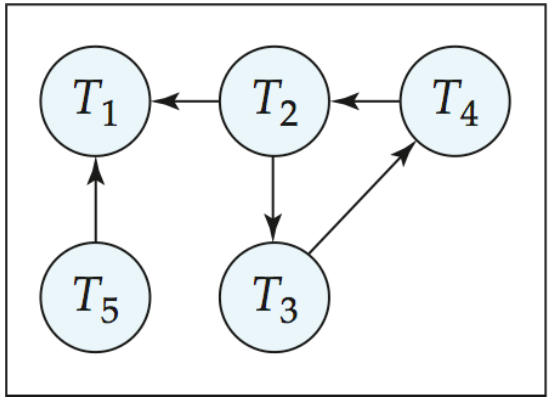


site S_1



site S_2

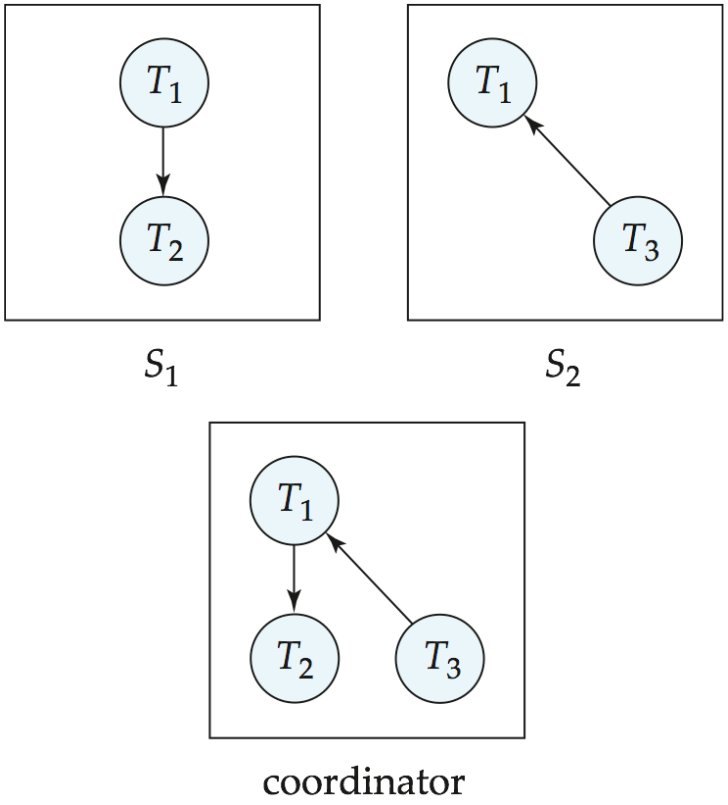
Local



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

Concurrency Control With Replicas

- The focus here is on concurrency control with locking
  - Failures addressed later
  - Ideas described here can be extended to other protocols
- **Primary copy**
  - one replica is chosen as primary copy for each data item
    - Node containing primary replica is called **primary node**
  - concurrency control decisions made at the primary copy only
- Benefit: Low overhead
- Drawback: primary copy failure results in loss of lock information and non-availability of data item, even if other replicas are available
  - Extensions to allow backup server to take over possible, but vulnerable to problems on network partition

Concurrency Control With Replicas (Cont.)

- **Majority protocol**:
  - Transaction requests locks at multiple/all replicas
  - Lock is successfully acquired on the data item only if the lock is obtained at a majority of replicas
- Benefit: resilient to node failures
  - Processing can continue as long as at least a majority of replicas are accessible
- Overheads
  - Higher cost due to multiple messages
  - Possibility of deadlock even when locking single item

Concurrency Control With Replicas (Cont.)

- **Biased protocol**
  - Shared lock can be obtained on any replica
    - Reduces overhead on reads
  - Exclusive lock must be obtained on *all* replicas
    - Blocking if any replica is unavailable

Quorum Consensus Protocol

Quorum consensus protocol for locking

- 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 $2 * Q_w > S$
- Each read must lock enough replicas that the sum of the site weights is $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is $\geq Q_w$
- Can choose Q_r and Q_w to tune relative overheads on reads and writes
  - Suitable choices result in majority and biased protocols.
    - What are they?

Dealing with Failures

- Read one write all copies protocol assumes all copies are available
  - Will block if any site is not available
- *Read one write all available* (ignoring failed sites) is attractive, but incorrect
  - Failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an incorrect value
  - With network partitioning, sites in each partition may update same item concurrently
    - believing sites in other partitions have all failed

Handling Failures with Majority Protocol

- The majority protocol with version numbers
  - Each replica of each item has a **version number**
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
- Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task

Handling Failures with Majority Protocol

- Atomic commit of updated replicas must be ensured using either
  - *2 phase commit on all locked replicas*, or
  - distributed consensus protocol such as Paxos (studied in other courses)
- Failure of nodes during 2PC can be ignored as long as majority of sites enter prepared state
- Failure of coordinator can cause blocking
  - Consensus protocols can avoid blocking

Handling Failures with Majority Protocol

- Benefits of majority protocol
  - Failures (network and site) do not affect consistency
    - Reads are guaranteed to see latest successfully written version of a data item
  - Protocol can proceed as long as
    - Sites available at commit time contain a majority of replicas of any updated data items
    - During reads a majority of replicas are available to find version numbers
  - No need for any special reintegration protocol: nothing needs to be done if nodes fail and subsequently recover
- Drawback of majority protocol
  - Higher overhead, especially for reads

Distributed Database in Oracle

- Some fragmentation can be achieved *expanding* a local database with another:
- **create database link *linkname***
  - The relations from *linkname* are known by relation@linkname
  - It is also possible to create aliases (cf. above in these slides) with
- **create synonym *alias* for *relation@linkname***
- This can be coupled with **materialized view**s which further enable vertical fragmentation
  - It is possible to establish how and when a materialised view is updated
    - **Fast refresh** uses materialised view logs to update only the rows that have changed since the last refresh.
    - **Complete refresh** always updates the entire materialised view.
    - **Force refresh** performs a fast refresh when possible. When a fast refresh is not possible, force refresh performs a complete refresh
- Queries can be distributed over the various sites

Replication in Oracle

- Oracle has support for homogeneous distributed replicated databases
  - It supports a multimaster replication with two-phase commit protocol
  - It supports master-slave replication by creating snapshots
- To create a replica
- **create snapshot *name* as *select query* with *type***
  - Replicas can be **read only** or **updatable** (*type* above)
- Groups of replicas, and their refreshing mechanisms can be define via special API procedures (Advanced Replication Management API)
  - In the labs you'll test DBMS_REFRESH
    - To create a refresh group and establish the refresh policy
- DBMS_REFRESH.MAKE(…)
    - To force a refresh
- DBMS_REFRESH. REFRESH(…)

Wrap-up

Sistemas de Bases de Dados 2021/22

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

Syllabus revisited

- Storing and file structure
  - Basis on how data is stored and (low-level) accessed in database systems
  - Understand how that is related to the OS storing, and how it affects performance of databases
- Indexing and hashing
  - Data structures for fast access, and how their performance depends on the specific data
- Query processing and optimisation
  - Get to know how a database system processes queries
  - Algorithms for query processing
  - Understanding how the performance of algorithms depends on the specific data
  - Learn about optimisation methods for (automatically) tailoring the queries to the specific data

Syllabus revisited

- Transactions, concurrency and recovery
  - Understand the concept of ACID transaction
  - Protocols for isolation, and for recovery
  - Understanding the need for various (weaker) isolation levels in transactional database systems, and tailor their use
- Distributed Databases
  - Basis for distributed databases and their practical use
  - Adaptation of transaction protocols and of query processing

Goals revisited

- *Pretende-se dotar os alunos das bases necessárias à compreensão dos problemas envolvidos na construção e funcionamento de sistemas de gestão de bases de dados, dando ênfase à utilização eficiente de sistemas de bases de dados.*

- The most important components, and underlying concepts, of a database system have been exposed in the lectures
- They have been tested, and used in practice, in the labs
  - Not in big examples, but enough for understanding the differences between the various concepts/approaches
- They have been witnessed in Oracle, and in the database systems used for the project assignment
- This provides a systemic view of information systems, and a basis for more advanced courses in this area