Distributed Transaction Management

Advanced Topics in Database Management (INFSCI 2711)

Some materials are from Database Management Systems,
Ramakrishnan and Gehrke
and
Database System Concepts,
Siberschatz, Korth and Sudarshan
and
Data Management in the Cloud,
Aggrawal, Das, Abbadi

Vladimir Zadorozhny, DINS, University of Pittsburgh

1

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites

2

Distributed Data Storage

- Assume relational data model
- 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.

3

Transactions

- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

T1: R(A); A=A+100; W(A); R(B); B=B-100; W(B); Commit

4

The ACID properties

- **A**tomicity: All actions in the Xact happen, or none happen.
- **C**onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I**solation: Execution of one Xact is isolated from that of other Xacts.
- **D**urability: If a Xact commits, its effects persist.

5

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.

6

Example

- Consider a possible interleaving (*schedule*):

T1:	A=A+100,		B=B-100
T2:		A=1.06*A,	B=1.06*B

- This is OK. But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- The DBMS's view of the second schedule:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

7

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule:* A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

8

Lock-Based Concurrency Control

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

T1: S(A), R(A), unlock(A)
T2: X(A), R(A), W(A), unlock(A)

9

Two-Phase Locking (2PL)

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- A transaction can not request additional locks once it releases any locks.
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

10

Strict 2PL

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- All locks held by a transaction are released when the transaction completes
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

Strict 2PL allows ***only serializable*** schedules

11

Deadlocks and Deadlock Detection

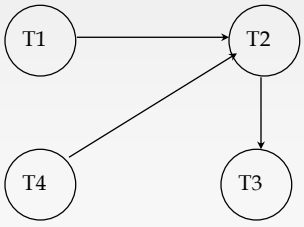
- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

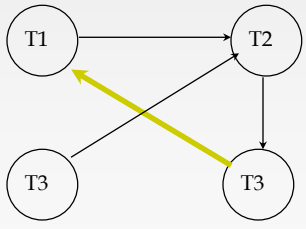
12

Deadlock Detection (Continued)

Example:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)





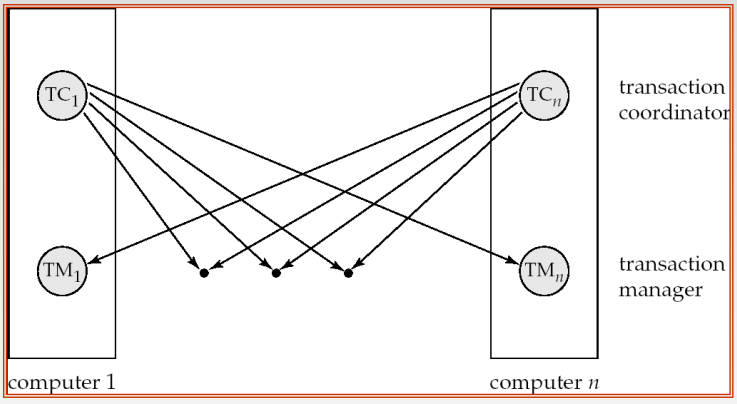
13

Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which 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, which may result in the transaction being committed at all sites or aborted at all sites.

14

Transaction System Architecture



15

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.

16

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.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.

17

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- 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
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

18

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 message, 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

19

Phase 2: Recording the Decision

- *T* can be committed of 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 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.

20

Handling of Failures - Site Failure

When site S_i 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* replies 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_1 must abort *T*
  - S_k must execute **undo** (*T*)

21

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*. Can therefore 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.

22

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 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

Alternative Models of Transaction Processing

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g. transaction crossing an organizational boundary
  - No organization would like to permit an externally initiated transaction to block local transactions for an indeterminate period
- Alternative models carry out transactions by sending messages
  - Code to handle messages must be carefully designed to ensure atomicity and durability properties for updates
    - Isolation cannot be guaranteed, in that intermediate stages are visible, but code must ensure no inconsistent states result due to concurrency
  - Persistent messaging systems are systems that provide transactional properties to messages
    - Messages are guaranteed to be delivered exactly once

Persistent Messaging and Workflows

- **Workflows** provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps
  - E.g. when a bank receives a loan application, it may need to
    - Contact external credit-checking agencies
    - Get approvals of one or more managers

    and then respond to the loan application
  - Persistent messaging forms the underlying infrastructure for workflows in a distributed environment

25

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 automicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

26

Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

27

Single-Lock-Manager Approach (Cont.)

- 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.

28

Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock detection
    - More on this later
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus

29

Primary Copy

- Choose one replica of data item to be the **primary copy**.
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item *Q*, it requests a lock at the primary site of *Q*.
  - Implicitly gets lock on all replicas of the data item
- Benefit
  - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation.
- Drawback
  - If the primary site of *Q* fails, *Q* is inaccessible even though other sites containing a replica may be accessible.

30

Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item *Q* residing at site S_i, a message is sent to S_i 's lock manager.
  - If *Q* is locked in an incompatible mode, then the request is delayed until it can be granted.
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

31

Majority Protocol (Cont.)

- In case of replicated data
  - If *Q* is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which *Q* is stored.
  - The transaction does not operate on *Q* until it has obtained a lock on a majority of the replicas of *Q*.
  - When writing the data item, transaction performs writes on *all* replicas.
- Benefit
  - Can be used even when some sites are unavailable
    - details on how handle writes in the presence of site failure later
- Drawback
  - Requires 2(*n*/2 + 1) messages for handling lock requests, and (*n*/2 + 1) messages for handling unlock requests.
  - Potential for deadlock even with single item - e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data.

32

Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
- **Shared locks**. When a transaction needs to lock data item *Q*, it simply requests a lock on *Q* from the lock manager at one site containing a replica of *Q*.
- **Exclusive locks**. When transaction needs to lock data item *Q*, it requests a lock on *Q* from the lock manager at all sites containing a replica of *Q*.
- Advantage - imposes less overhead on **read** operations.
- Disadvantage - additional overhead on writes

33

Quorum Consensus Protocol

- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let S be the total of all site weights
- Choose two values read quorum Q_r and write quorum Q_w
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
  - Quorums can be chosen (and S computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is >= Q_r
- Each write must lock enough replicas that the sum of the site weights is >= Q_w
- For now we assume all replicas are written
  - Extensions to allow some sites to be unavailable described later

34

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

35

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

36

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

37

Example Use of the Protocol

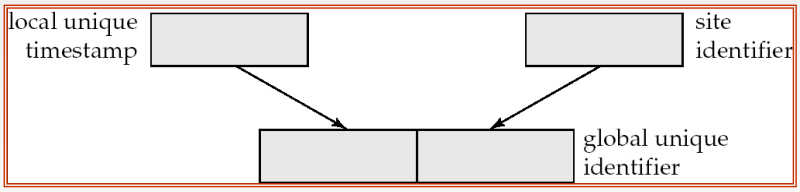
A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
	read(Y)			read(X)
read(Y)		write(Y)		
		write(Z)		read(Z)
	write(X)			
read(X)	abort			
		write(Z)		
		abort		
				write(Y)
				write(Z)

38

Timestamping

- 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.



39

Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: "disadvantages" transactions
- To fix this problem
  - Define within each site S_i a **logical clock** (LC_i), which generates the unique local timestamp
  - Require that S_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 S_i advances its logical clock to the value $x + 1$.

40

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 (Y)
write (X)

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

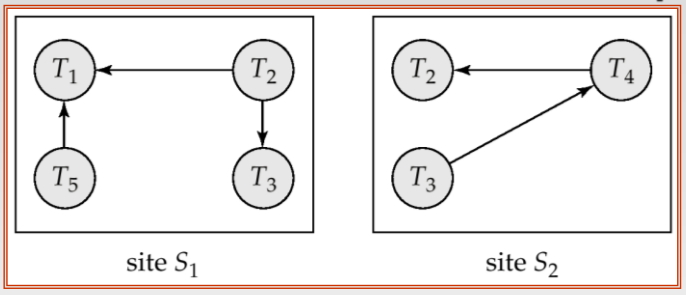
41

Centralized Approach

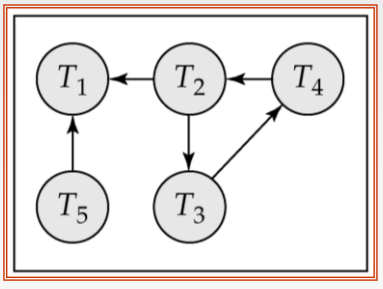
- A global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*:Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph 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.

42

Local and Global Wait-For Graphs



Local

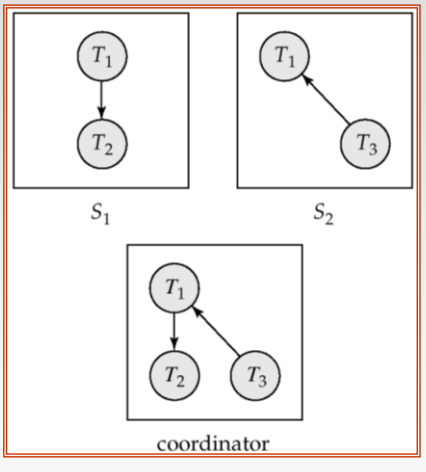


Global

43

Example Wait-For Graph for False Cycles

Initial state:



44

Availability

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- Robustness: ability of system to function spite of failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery/reintegration when a site or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure

45

Reconfiguration

- Reconfiguration:
  - Abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g. name server, concurrency coordinator, global deadlock detector

46

Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to latest version, then release locks
    - Other solutions with better concurrency also available