

Database Management Systems

Lecture 11

Distributed Databases

Distributed Transaction Management

- a transaction submitted at a site S could ask for data stored at several other sites
- *subtransaction* - the activity of a transaction at a given site
- context: Strict 2PL with deadlock detection
- problems:
 - distributed concurrency control
 - lock management when objects are stored across several sites
 - deadlock detection
 - distributed recovery
 - transaction atomicity
 - all the effects of a committed transaction (across all the sites it executes at) are permanent
 - none of the actions of an aborted transaction are allowed to persist

Distributed Transaction Management

- distributed concurrency control
 - lock management
 - techniques – synchronous / asynchronous replication
 - which objects will be locked
 - concurrency control protocols
 - when are locks acquired / released
 - lock management
 - *centralized*
 - *primary copy*
 - *fully distributed*

Distributed Transaction Management

- distributed concurrency control
 - lock management
 - centralized:
 - one site does all the locking for all the objects
 - vulnerable to single-site failures
 - primary copy:
 - object O, primary copy PC of O stored at site S with lock manager L
 - all requests to lock / unlock some copy of O are handled by L
 - not vulnerable to single-site failures
 - read some copy C of O stored at site S2:
=> communicate with both S and S2

Distributed Transaction Management

- distributed concurrency control
 - lock management
 - fully distributed:
 - object O, some copy C of O stored at site S with Lock Manager L
 - requests to lock / unlock C are handled by L (the site where the copy is stored)
 - one doesn't need to access 2 sites when reading some copy of O

Distributed Transaction Management

- distributed concurrency control
 - detect and resolve deadlocks
 - each site maintains a local waits-for graph
 - a cycle in such a graph indicates a deadlock
 - but a global deadlock can exist even if none of the local graphs contains a cycle

->

Distributed Transaction Management

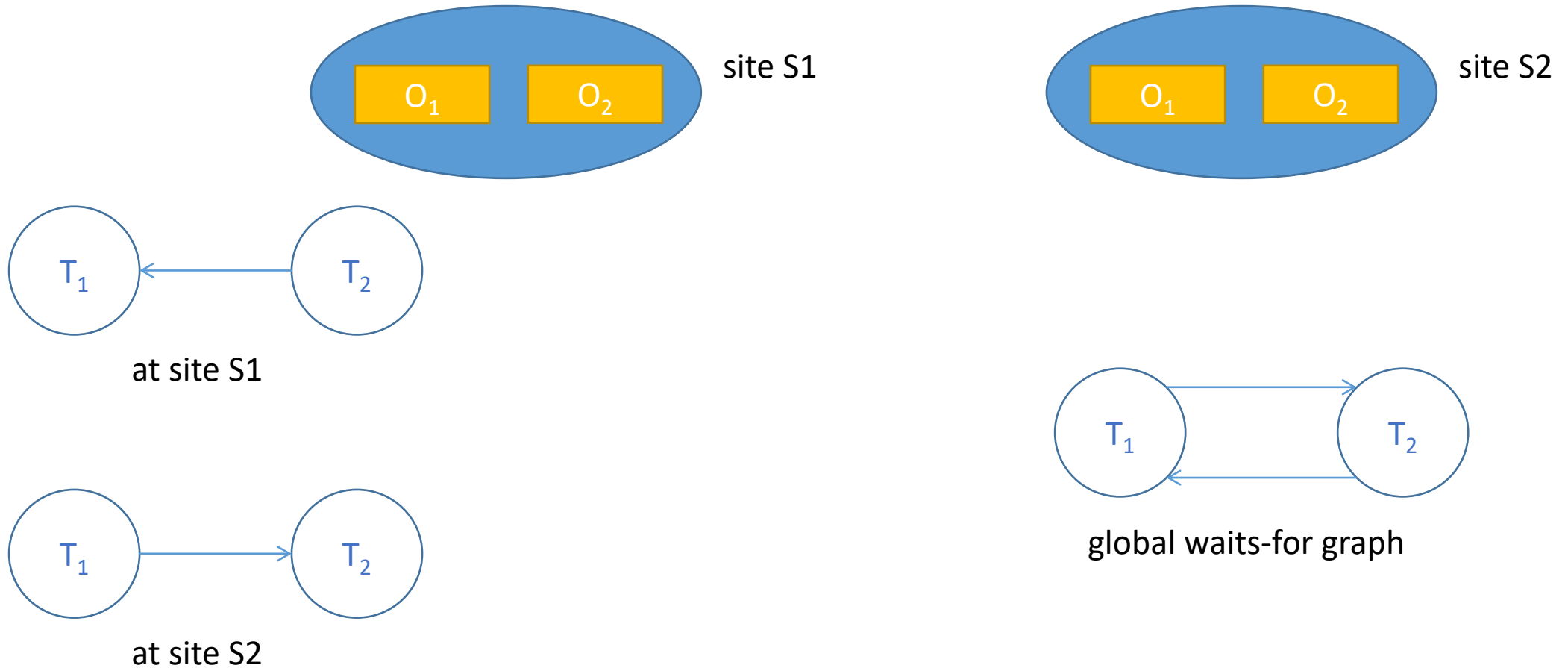
- distributed concurrency control – distributed deadlock
 - e.g., using *read-any write-all*



- T_1 wants to read O_1 and write O_2
- T_2 wants to read O_2 and write O_1
- T_1 acquires a S lock on O_1 and an X lock on O_2 at site S1
- T_2 obtains a S lock on O_2 and an X lock on O_1 at site S2
- T_1 asks for an X lock on O_2 at site S2
- T_2 asks for an X lock on O_1 at site S1

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - e.g., using *read-any write-all*



Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - distributed deadlock detection algorithms
 - *centralized*
 - *hierarchical*
 - based on a *timeout* mechanism

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - distributed deadlock detection algorithms
 - centralized:
 - all the local waits-for graphs are periodically sent to a single site S
 - S - responsible for global deadlock detection
 - the global waits-for graph is generated at site S
 - nodes
 - the union of the nodes in the local graphs
 - edges
 - there is an edge from node N1 to node N2 if such an edge exists in one of the local graphs

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - distributed deadlock detection algorithms
 - hierarchical:
 - sites are organized into a hierarchy, e.g., grouped by city, county, country, etc.
 - each site periodically sends its local waits-for graph to its parent site
 - assumption: more deadlocks are likely across related sites
 - all the deadlocks are detected in the end

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - distributed deadlock detection algorithms

- hierarchical:

- example:

RO (CJ (Cluj-Napoca, Dej, Turda), BN (Bistrita, Beclean))

Cluj-Napoca : T1 -> T2

Dej: T2 -> T3

Turda: T3 -> T4 <- T7

Bistrita: T5 -> T6

Beclean: T4 -> T7 -> T6 -> T5

CJ: T1 -> T2 -> T3 -> T4 <- T7

BN: T5 <-> T6 <- T7 <- T4 (*)

RO: T1 -> T2 -> T3 -> T4 <-> T7 -> T6 <-> T5

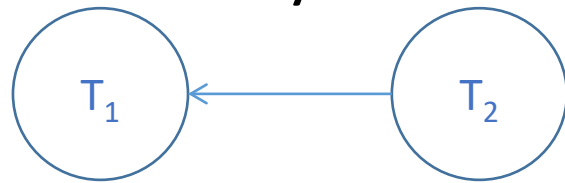
Obs RO: T5 or T6 was aborted at (*)

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
 - distributed deadlock detection algorithms
 - based on a timeout mechanism:
 - a transaction is aborted if it lasts longer than a specified interval
 - can lead to unnecessary restarts
 - however, the deadlock detection overhead is low
 - could be the only available option in a heterogeneous system (if the participating sites cannot cooperate, i.e., they cannot share their local waits-for graphs)

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
- phantom deadlocks
 - "deadlocks" that don't exist, but are detected due to delays in propagating local information
 - lead to unnecessary aborts
 - example:



at site S1



global waits-for graph



at site S2

- generate local waits-for graphs, send them to the site responsible for global deadlock detection
- T₂ aborts (not because of the deadlock) => local waits-for graphs are changed, there is no cycle in the "real" global waits-for graph
- but the built waits-for graph does have a cycle, T₁ could be chosen as a victim

Distributed Transaction Management

- distributed recovery
 - more complex than in a centralized DBMS
 - new types of failure
 - network failure
 - site failure
 - commit protocol
 - either all the subtransactions of a transaction commit, or none of them does
- normal execution
 - ensure all the necessary information is provided to recover from failures
 - a log is maintained at each site; it contains:
 - data logged in a centralized DBMS
 - actions carried out as part of the commit protocol

Distributed Transaction Management

- distributed recovery
 - transaction T
 - coordinator
 - the Transaction Manager at the site where T originated
 - subordinates
 - the Transaction Managers at the sites where T's subtransactions execute

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol (2PC)
 - exchanged messages + records written in the log
 - 2 rounds of messages, both initiated by the coordinator
 - *voting phase*
 - *termination phase*
 - any Transaction Manager can abort a transaction
 - however, for a transaction to commit, all Transaction Managers must decide to commit

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol
 - the user decides to commit transaction T
- => the commit command is sent to T's coordinator, 2PC is initiated
1. the coordinator sends a *prepare* message to each subordinate
 2. upon receiving a *prepare* message, a subordinate decides whether to commit / abort its subtransaction
- the subordinate force-writes an *abort* or a *prepare** log record
 - then it sends a *no* or *yes* message to the coordinator

* *prepare* log records are specific to the commit protocol, they are not used in centralized DBMSs

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol
- 3.
 - if the coordinator receives *yes* messages from all subordinates:
 - it force-writes a *commit* log record
 - it then sends *commit* messages to all subordinates
 - otherwise (i.e., if it receives at least one *no* message or if it doesn't receive any message from a subordinate for a predetermined timeout interval)
 - it force-writes an *abort* log record
 - it then sends an *abort* message to each subordinate

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol
- 4.
 - upon receiving an *abort* message, a subordinate:
 - force-writes an *abort* log record
 - sends an *ack* message to the coordinator
 - aborts the subtransaction
 - upon receiving a *commit* message, a subordinate:
 - force-writes a *commit* log record
 - sends an *ack* message to the coordinator
 - commits the subtransaction

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol
 5. after it receives *ack* messages from all subordinates, the coordinator writes an *end* log record for the transaction
- * obs. sending a message – the sender has made a decision
 - the message is sent only after the corresponding log record has been forced to stable storage (to ensure the corresponding decision can survive a crash)
- * obs. T is a committed transaction if the commit log record of T's coordinator has been forced to stable storage

Distributed Transaction Management

- distributed recovery
 - two-phase commit protocol
 - log records for the commit protocol
 - record type
 - transaction id
 - coordinator's identity
 - the commit / abort log record for the coordinator also contains the identities of the subordinates

Distributed Transaction Management

- distributed recovery
 - restart after a failure – site S comes back up after a crash
 - if there is a *commit* or an *abort* log record for transaction T:
 - must redo / undo T
 - if S is T's coordinator:
 - periodically send *commit* / *abort* messages to subordinates until *ack* messages are received
 - write an *end* log record after receiving all *ack* messages
 - if there is a *prepare* log record for transaction T, but no *commit* / *abort*, S is one of T's subordinates
 - contact T's coordinator repeatedly until T's status is obtained
 - write a *commit* / an *abort* log record
 - redo / undo T

Distributed Transaction Management

- distributed recovery
 - restart after a failure – site S
 - if there are no *commit* / *abort* / *prepare* log records for T:
 - abort T, undo T
 - if S is T's coordinator, T's subordinates may subsequently contact S

Distributed Transaction Management

- distributed recovery
 - link and remote site failures
 - current site S, remote site R, transaction T
 - if R doesn't respond during the commit protocol for T, either because R failed or the link failed:
 - if S is T's coordinator:
 - S should abort T
 - if S is one of T's subordinates, and has not voted yet:
 - S should abort T
 - if S is one of T's subordinates and has voted yes:
 - S is blocked until T's coordinator responds

Distributed Transaction Management

- distributed recovery
 - 2PC – observations
 - *ack* messages
 - used to determine when can a coordinator C “forget” about a transaction T
 - C must keep T in the transaction table until it receives all *ack* messages
 - C fails after sending *prepare* messages, but before writing a *commit* / an *abort* log record
 - when C comes back up it aborts T
 - i.e., absence of information => T is presumed to have aborted
 - if a subtransaction doesn't change any data, its commit / abort status is irrelevant

Distributed Transaction Management

- distributed recovery
 - 2PC with Presumed Abort
 - coordinator C, transaction T, some subordinate S, some subtransaction t
 - C aborts T
 - T is undone
 - C immediately removes T from the Transaction Table, i.e., it doesn't wait for *ack* messages
 - subordinates' names need not be recorded in C's abort log record
 - S doesn't send an *ack* message when it receives an *abort* message

Distributed Transaction Management

- distributed recovery
 - 2PC with Presumed Abort
 - coordinator C, transaction T, some subordinate S, some subtransaction t
 - t doesn't change any data
 - t responds to *prepare* messages with a *reader* message, instead of *yes / no*
 - C subsequently ignores readers
 - if all subtransactions are readers, the 2nd phase of the protocol is not needed

Distributed Catalog Management - optional

- keeping track of data distribution across sites
- one should be able to identify each replica of each fragment for a relation that is fragmented and replicated
- local autonomy should not be compromised
 - solution - names containing several fields:
 - global relation name:
 - <local-name, birth-site>
 - global replica name:
 - <local-name, birth-site, replica_id>

Distributed Catalog Management - optional

- centralized system catalog
 - stored at a single site
 - contains data about all the relations, fragments, replicas
 - vulnerable to single-site failures
 - can overload the server
- global system catalog maintained at each site
 - every copy of the catalog describes all the data
 - not vulnerable to single-site failures (the data can be obtained from a different site)
 - local autonomy is compromised:
 - changes to a local catalog must be propagated to all the other sites

Distributed Catalog Management - optional

- local catalog maintained at each site
 - each site keeps a catalog that describes local data, i.e., copies of data stored at the site
 - the catalog at the birth-site for a relation keeps track of all the fragments / replicas of the relation
 - create a new replica / move a replica to another site:
 - must update the catalog at the birth-site
 - not vulnerable to single-site failures & doesn't compromise local autonomy

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition,
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,
<http://infolab.stanford.edu/~ullman/fcdb.html>