Advanced DB

CHAPTER 22
DISTRIBUTED DATABASES

Chapter 19: Distributed Databases

- Homogeneous and Heterogeneous Databases
- Distributed Data Storage
- Distributed Transactions
- Commit Protocols
- Concurrency Control in Distributed Databases
- Availability
- Distributed Query Processing
- Heterogeneous Distributed Databases
- Directory Systems

Distributed Database System

- Distributed DB 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
- Homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- Heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

Distributed Data Storage

- Assume relational data model
- Replication
  - 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.
- Advantages
  - ***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
  - ***Cost of updates***: each replica of relation *r* must be updated.
  - ***Concurrency control complexity***: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.

Data Fragmentation

- Division of relation r into fragments $r_1, r_2, \cdots, r_n$ which contain sufficient information to reconstruct relation *r*.
- *Horizontal fragmentation*: each tuple of *r* is assigned to one or more fragments
- *Vertical fragmentation*: the schema for relation *r* is split into several smaller schemas
  - All schemas 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.
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.

Original Slides:
© Silberschatz, Korth and Sudarshan

Copyright © 2006 - 2008 by S.-g. Lee

Horizontal Fragmentation

Account-schema = (*branch-name*, *account-number*, *balance*)

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1 = \sigma_{branch\text{-}name = \text{“Hillside”}}(account)$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2 = \sigma_{branch\text{-}name = \text{“Valleyview”}}(account)$

Vertical Fragmentation

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5

$$account_v_1 = \Pi_{branch\text{-}name,\ customer\text{-}name,\ tuple\text{-}id}(employee\text{-}info)$$

account number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5

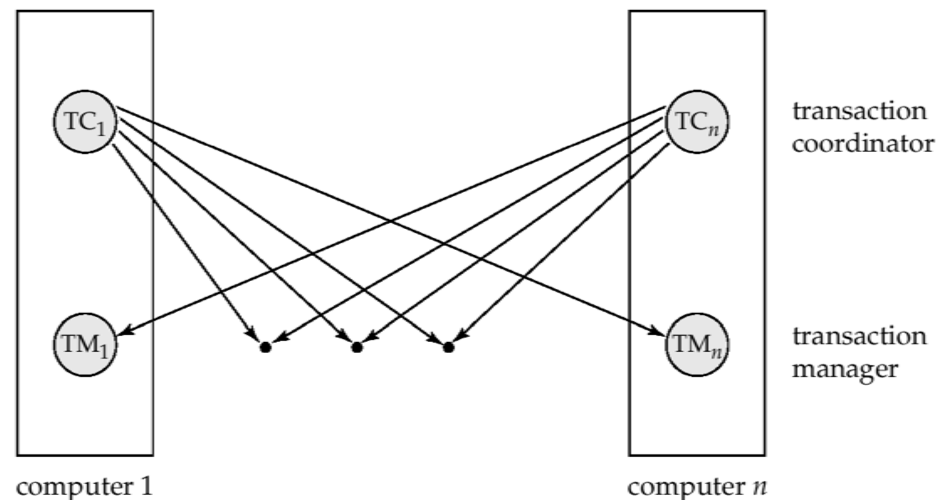
$$account_v_2 = \Pi_{account\text{-}number,\ balance,\ tuple\text{-}id}(employee\text{-}info)$$

Data Transparency

- Degree to which user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency
- Ways to achieve transparency
  - Naming – centralized vs aliases
  - Views

Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local *transaction manager* responsible for:
  1. Maintaining a log for recovery purposes
  2. Coordinating the concurrent execution of the transactions executing at that site.
- Each site has a *transaction coordinator*, which is responsible for:
  1. Starting the execution of transactions that originate at the site.
  2. Distributing subtransactions at appropriate sites for execution.
  3. 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.



Original Slides:
© Silberschatz, Korth and Sudarshan

System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site
  - Loss of messages
    - Handled by network transmission control protocols; e.g. 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.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2 *PC*) protocol is widely used
- The *three-phase commit* (3 *PC*) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

Two Phase Commit Protocol (2PC)

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

© Silberschatz, Korth and Sudarshan

Copyright © 2006 - 2008 by S.-g. Lee

Phase 1: Obtaining a Decision

1. Coordinator asks all participants to *prepare* to commit transaction *T*.
   - 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
2. Upon receiving message, transaction manager at site determines if it can commit the transaction
   - if not
     - add a record <**no** *T*> to the log
     - send "**abort** *T*" message to C_i
   - if the transaction can be committed, then:
     - add the record <**ready** *T*> to the log
     - force *all log records* for *T* to stable storage
     - send "**ready** *T*" message to C_i

Phase 2: Recording the Decision

1. Decision
   - *T* can be committed if C_i received a "**ready** *T*" message from *all* the participating sites
   - otherwise *T* must be aborted
2. Log Decision
   - Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log
   - Forces record onto stable storage
   - Once the record is written on stable storage it is irrevocable (even if failures occur)
3. Coordinator sends a message to each participant informing it of the decision (commit or abort)
4. Participants take appropriate action locally
   - and write <**commit** *T*> or <**abort** *T*> to their log

Handling of Failures - Site Failure

- When (noncoordinator) site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure
  - Log contains <**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*. Can therefore **abort** *T*.
  4. If none of the above cases holds
     - all active sites have a <**ready** *T*> record but no additional control records (such as <**abort** *T*> of <**commit** *T*>)
     - In this case active sites must wait for C_i to recover
- Blocking problem
  - Situation where active sites 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 in the same partition 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: the site is unusable!
- 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, new transactions can be executed
  - the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions

Alternative Models of Transaction

- 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
- Persistent messaging systems
  - Systems that provide transactional properties to messages
  - Messages are guaranteed to be delivered exactly once

Persistent Messaging

- Motivating example: *funds transfer between two banks*
  - Two phase commit has the potential to block operation
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue
  - Once transaction sending a message is committed, message is guaranteed to be delivered
    - What if the receiving site is down?
    - Code to handle undeliverable messages must also be available
      - e.g. credit money back to source account.
  - There are many situations where extra effort of error handling is worth the benefit of absence of blocking
    - e.g. most transactions across organizations
  - If sending transaction aborts, message must not be sent

Persistent Messaging (cont.)

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - E.g. if destination account does not exist, failure message must be sent back to source site
  - When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed => get humans to take care of problem
- User code executing transaction processing using 2PC does not have to deal with such failures

Persistent Messaging and Workflows

- Workflows
  - 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
  - Workflow Management Systems (WFMS)

Implementation of Persistent Messaging

- Sending site protocol

1. Sending transaction writes message to a special relation *messages-to-send*. The message is also given a unique identifier.
   - Writing to this relation is treated as any other update, and is undone if the transaction aborts.
   - The message remains locked until the sending transaction commits
2. A message delivery process monitors the *messages-to-send* relation
   - When a new message is found, the message is sent to its destination
   - When an acknowledgment is received from a destination, the message is deleted from *messages-to-send*
   - If no acknowledgment is received after a timeout period, the message is resent
     - This is repeated until the message gets deleted on receipt of acknowledgement, or the system decides the message is undeliverable

Implementation of Persistent Messaging

- Receiving site protocol

  When a message is received

  1. It is written to a *received-messages* relation
     - if it is not already present (the message id is used for this check).
  2. The transaction performing the write is committed
  3. An acknowledgement is then sent to the sending site
- There may be very long delays in message delivery coupled with repeated messages
    - Could result in processing of duplicate messages if we are not careful
  - Option 1: messages are never deleted from *received-messages*
  - Option 2: messages are given timestamps
    - Messages older than some cut-off are deleted from *received-messages*
    - Received messages are rejected if older than the cut-off

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

- 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
  - 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

- 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 must cooperate for deadlock detection
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus

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.

Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- To access data item Q
  - 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.
  - (What if Q is not replicated?)
- Benefit
  - Can be used even when some sites are unavailable
- 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.

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
  - need to obtain a lock at *one* site containing a replica of Q.
- Exclusive locks
  - need to obtain locks at *all* sites containing a replica of Q.
- Conflicting lock requests are always detected.
- Advantage / Disadvantage
  - imposes less overhead on read operations.
  - additional overhead on writes

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 $>= Q_w$
- Conflicting lock requests are always detected.
- By configuring Q_r, Q_w, and weights of sites, we can emulate majority and biased protocols

Distributed Query Processing

- For centralized systems
  - the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system
  - Other issues must be taken into account:
  - The cost of a data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

© Silberschatz, Korth and Sudarshan

Copyright © 2006 - 2008 by S.-g. Lee

Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Consider the horizontal fragmentation of the *account* relation into

  $account_1 = \sigma_{branch\text{-}name = \text{"Hillside"}}(account)$; $account_2 = \sigma_{branch\text{-}name = \text{"Valleyview"}}(account)$
- The query

$$\begin{aligned} & \sigma_{branch\text{-}name = \text{"Hillside"}}(account) \\ = \; & \sigma_{\text{branch-name} = \text{"Hillside"}}(account_1 \cup account_2) \\ = \; & \sigma_{branch\text{-}name = \text{"Hillside"}}(account_1) \cup \sigma_{branch\text{-}name = \text{"Hillside"}}(account_2) \end{aligned}$$

- For $account_1$
  - Has only tuples pertaining to the Hillside branch => eliminate the selection operation
- For $account_2$
  - Using the definition, $\sigma_{branch\text{-}name = \text{"Hillside"}}(\sigma_{branch\text{-}name = \text{"Valleyview"}}(account))$
  - This expression is the empty set regardless of the contents of the *account* relation
- Final strategy
  - Hillside site is to return $account_1$

Simple Join Processing

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

$$account \bowtie depositor \bowtie branch$$

  - *account* is stored at site S_1
  - *depositor* at S_2
  - *branch* 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

- Strategy 1
  - Ship copies of all three relations to site S_I
  - Process the entire query locally at site S_I.
- Strategy 2
  - Ship a copy of the *account* relation to site S_2 and compute

    $temp_1 = account \bowtie depositor$ at S_2.
  - Ship $temp_1$ from S_2 to S_3, and compute

    $temp_2 = temp_1 \bowtie branch$ at S_3.
  - Ship the result $temp_2$ to S_I.
- Others
  - 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 $r_1(R_1)$ be at site S_1, and $r_2(R_2)$ at site S_2
- Query: $r_1 \bowtie r_2$ - produce result at S_1

  1. Compute $temp_1 \leftarrow \Pi_{R1 \cap R2} (r_1)$ at S_1
  2. Ship $temp_1$ from S_1 to S_2
  3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ 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 $r_1 \bowtie r_2$.

- Formally, the *semijoin* of r_1 with r_2,

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

  - i.e., selects those tuples of r_1 that contributes to $r_1 \bowtie r_2$

  - In step 3 above, $temp_2 = r_2 \ltimes r_1$

Semijoin Strategy (cont.)

- Can be extended for joins of several relations

Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

- where relation r_i is stored at site S_i
- the result must be presented at site S_1

1. Simultaneously,
   - $r_1 \ltimes r_2$ shipped to S_2 and $r_1 \bowtie r_2$ computed at S_2
   - $r_3 \ltimes r_4$ shipped to S_4 and $r_3 \bowtie r_4$ computed at S_4
2. Simultaneously,
   - S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they are produced
   - S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1 as they are produced
3. $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed at S_1
   - as soon as tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive
   - may be computed in parallel with the computations of $(r_1 \bowtie r_2)$ at S_2 and $(r_3 \bowtie r_4)$ at S_4.

Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms
  - Data models may differ (hierarchical, relational , etc.)
  - Transaction commit protocols may be incompatible
  - Concurrency control may be based on different techniques (locking, timestamping, etc.)
  - System-level details almost certainly are totally incompatible.
- Reasons for supporting Hetero. Distr. DB (multidatabase system)
  - Preservation of investment in existing hardware, system software, applications
  - Local autonomy and administrative control
  - Allows use of special-purpose DBMSs
  - Full integration into a homogeneous DBMS faces
    - technical difficulties and cost of conversion
    - organizational/political difficulties

Unified View of Data

- Need a software layer on top of existing database systems
  - which is designed to manipulate information in heterogeneous databases
  - which creates an illusion of logical database integration without any physical database integration
- To provide a unified view of data
  - Agreement on a common data model
    - Typically the relational model
  - Agreement on a common conceptual schema
  - 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
  - Variations in names
    - E.g. Köln vs Cologne, Mumbai vs Bombay

Mediator Systems

- Systems that integrate multiple heterogeneous data sources by providing
  - an integrated global view and
  - query facilities on global view
- Mediators generally do not bother about transaction processing
  - cf) multidatabase system
  - the terms mediator and multidatabase are sometimes used interchangeably
  - The term *virtual database* is also used to refer to mediator/multidatabase systems

END OF CHAPTER 22