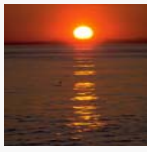
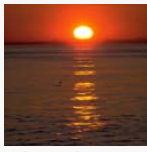
Chapter 22: Distributed Databases

Database System Concepts, 5th Ed.



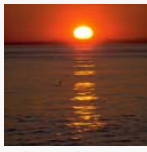
Chapter 22: Distributed Databases

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



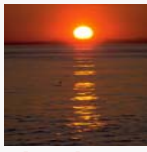
Distributed Database System

- 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



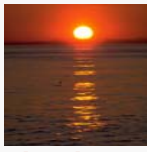
Homogeneous Distributed Databases

- In a 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
- In a 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



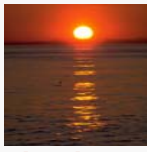
Chapter 22: Distributed Databases

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



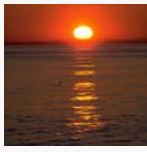
Data Replication

- 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.



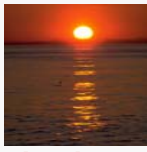
Data Replication (Cont.)

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* is 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
- **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.
- Example : relation account with following schema
- *Account* = (*account_number, branch_name , balance*)



Horizontal Fragmentation of *account* Relation

account_number	*branch_name*	*balance*
A-305	Hillside	500
A-226	Hillside	336
A-155	Hillside	62

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

account_number	*branch_name*	*balance*
A-177	Valleyview	205
A-402	Valleyview	10000
A-408	Valleyview	1123
A-639	Valleyview	750

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

Vertical Fragmentation of *employee_info* Relation

branch_name	*customer_name*	*tuple_id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

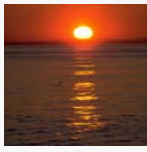
$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}\ (employee\_info\ )$

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
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}\ (employee\_info\ )$

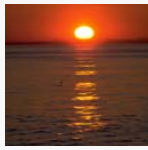
Advantages of Fragmentation

- 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
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.



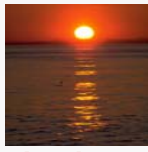
Data Transparency

- **Data transparency**: Degree to which system 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**
- Naming of data items: criteria
  1. Every data item must have a system-wide unique name.
  2. It should be possible to find the location of data items efficiently.
  3. It should be possible to change the location of data items transparently.
  4. Each site should be able to create new data items autonomously.



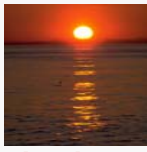
Centralized Scheme - Name Server

- Structure:
  - name server assigns all names
  - each site maintains a record of local data items
  - sites **ask name server** to locate non-local data items
- Advantages:
  - satisfies naming criteria 1-3
- Disadvantages:
  - does not satisfy naming criterion 4
  - name server is a potential performance bottleneck
  - name server is a single point of failure



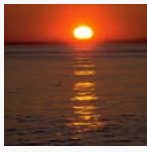
Use of Aliases

- Alternative to centralized scheme: each site prefixes its own site identifier to any name that it generates i.e., *site*17.a*ccount.*
  - Fulfills having a unique identifier, and avoids problems associated with central control.
  - However, fails to achieve network transparency.
- Solution: Create a set of **aliases** for data items; Store the mapping of aliases to the real names at each site.
  - The user can be unaware of the physical location of a data item, and is unaffected if the data item is moved from one site to another.



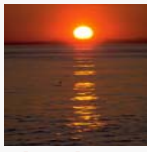
Chapter 22: Distributed Databases

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

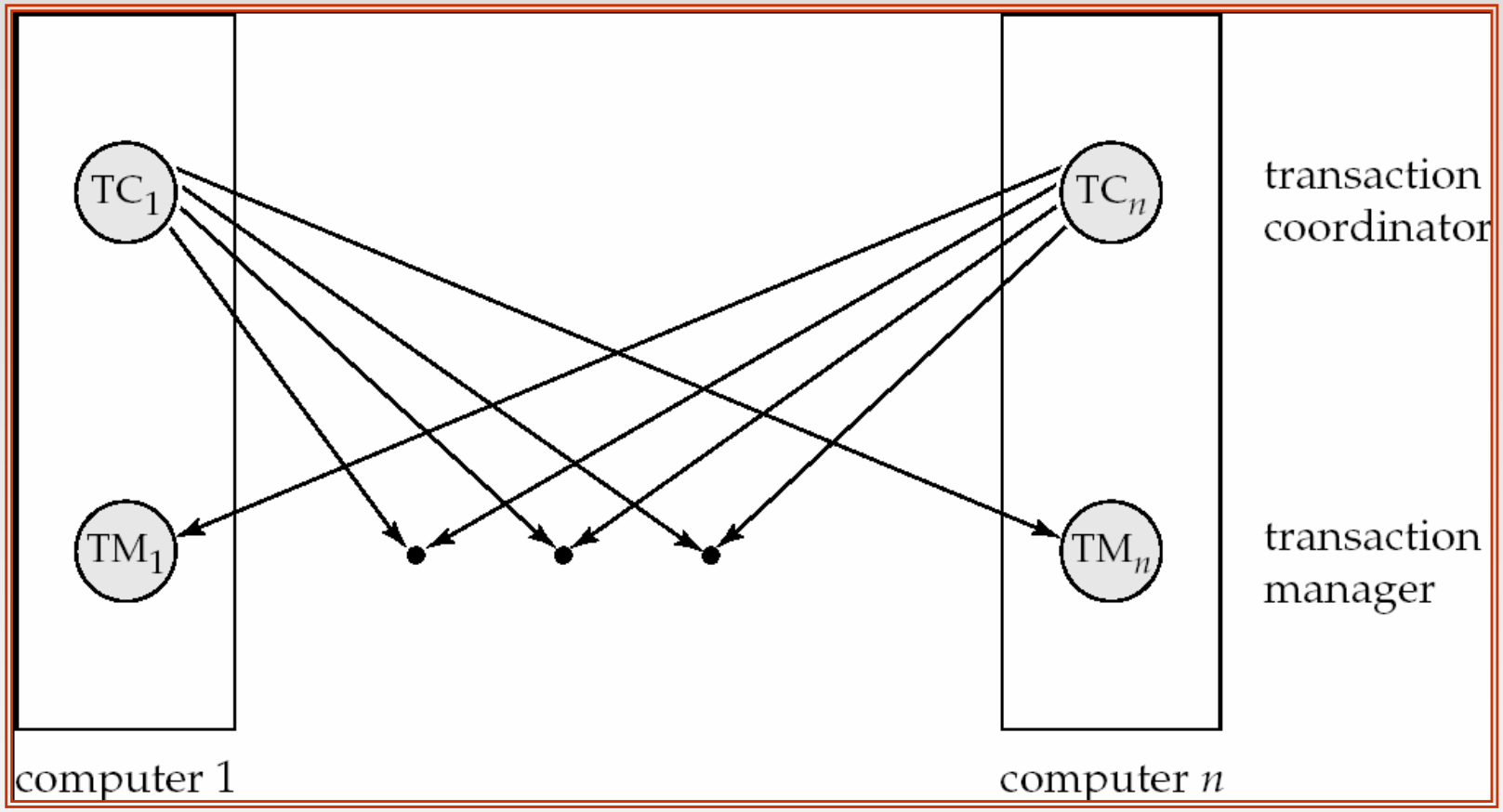


Distributed Transactions

- Transactions 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.



Transaction System Architecture

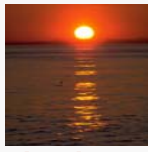


TM manages transactions that access data local to a site.

TC coordinates the execution of transactions (local and global) initiated at a site.

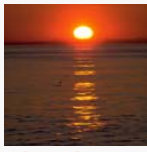
System Failure Modes

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



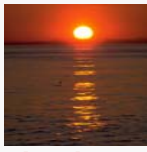
Chapter 22: Distributed Databases

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



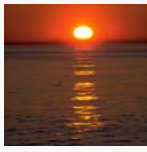
Commit Protocols

- 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.



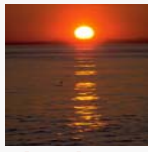
Two Phase Commit Protocol (2PC)

- 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



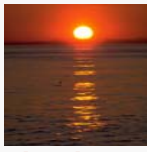
Phase 1: Obtaining a Decision

- 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



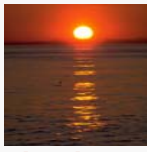
Phase 2: Recording the Decision

- ***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 on 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.



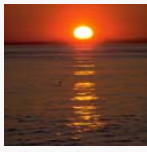
Chapter 22: Distributed Databases

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



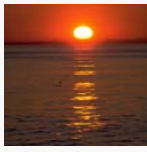
Concurrency Control

- 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



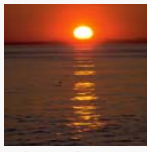
Single-Lock-Manager Approach

- 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



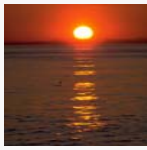
Single-Lock-Manager Approach (Cont.)

- 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.



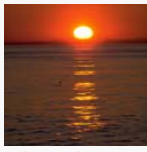
Distributed Lock Manager

- 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
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus



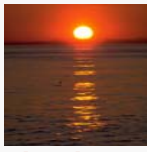
Primary Copy

- 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.**



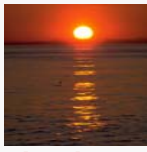
Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- 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.



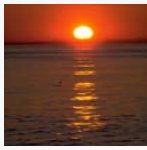
Majority Protocol (Cont.)

- 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
- 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.



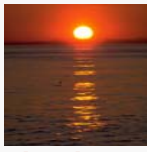
Biased Protocol

- 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**. 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



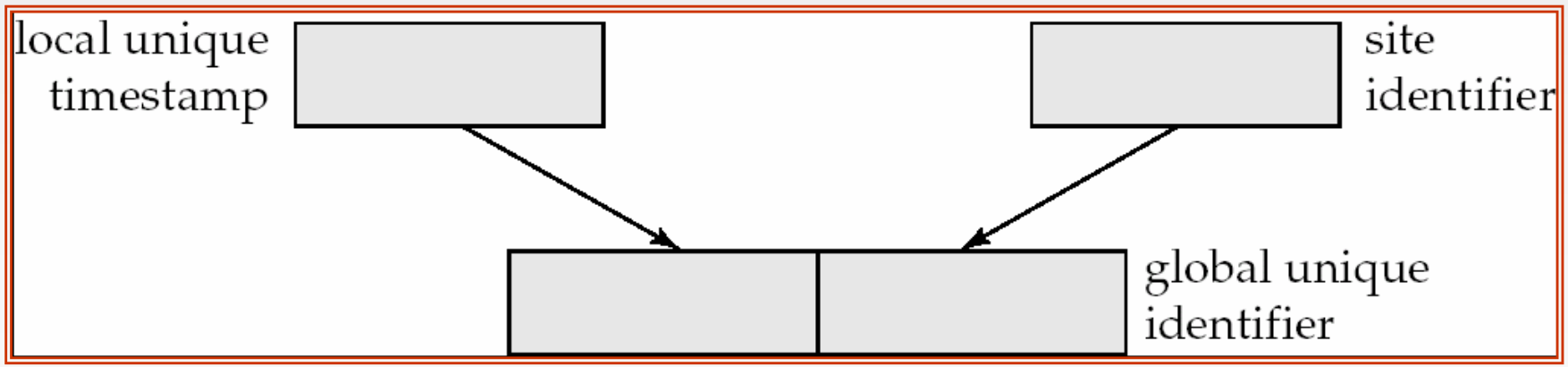
Quorum Consensus Protocol

- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let S be the total of all site weights
- For each item we 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



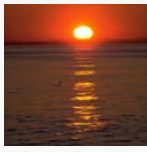
Timestamping

- 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.



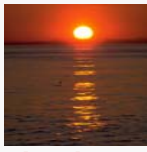
Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps
- To fix this problem
  - Define within each site S_i a **logical clock** (LC_i), which generates the unique local timestamp
  - Synchronization
    - Require that S_i advance its logical clock whenever a request is received from a transaction T_i 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.



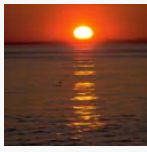
Replication with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency
- **Master-slave replication**: updates are performed at a single "master" site, and propagated to "slave" sites.
  - Propagation is not part of the update transaction: it is decoupled
    - May be immediately after transaction commits
    - May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g. from central office to branch-office
  - Also useful for running read-only queries offline from the main database



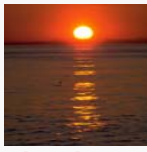
Replication with Weak Consistency (Cont.)

- Replicas should see a **transaction-consistent snapshot** of the database
  - That is, a state of the database reflecting all effects of all transactions **up to some point** in the serialization order, and no effects of any later transactions.
- E.g. Oracle provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by recomputation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh



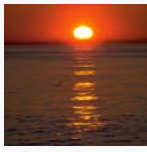
Multimaster and Lazy Replication

- With **multimaster replication** (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all replicas
  - Basic model in distributed databases, where transactions are unaware of the details of replication, and database system **propagates updates as part of the same transaction**
- Many systems support **lazy propagation** where updates are transmitted after transaction commits
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency



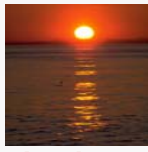
Chapter 22: Distributed Databases

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



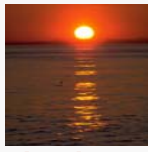
Availability

- 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 despite 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



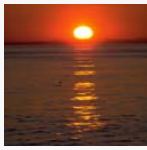
Reconfiguration

- 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. We must ensure to propagate updates once the failed site recovers.
  - 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



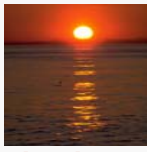
Reconfiguration (Cont.)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two ore more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Solution: majority based approach



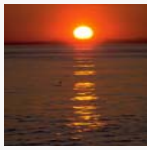
Majority-Based Approach

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable
  - Each replica of each item has a **version number** which is updated when the replica is updated, as outlined below
  - A lock request is sent to **at least ½ the sites** at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites
  - Read operations look at all replicas locked, and read the value from the replica with largest version number
    - May write this value and version number back to replicas with lower version numbers (no need to obtain locks on all replicas for this task)



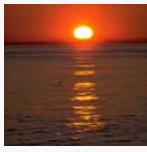
Majority-Based Approach

- Majority protocol (Cont.)
  - 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



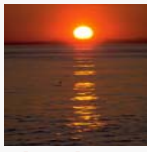
Read One Write All Available

- Biased protocol is a special case of quorum consensus
  - Allows reads to read any one replica but updates require all replicas to be available at commit time (called **read one write all**)
- Read one write all available (ignoring failed sites) is attractive, but incorrect
  - A 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**
    - If site was aware of failure, reintegration could have been performed, but no way to guarantee this



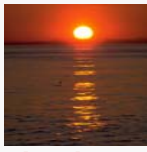
Site Reintegration

- 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



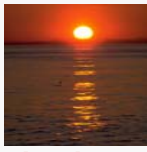
Comparison with Remote Backup

- Remote backup (hot spare) systems are also designed to provide high availability
- Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and **onlydata and log records shipped**
  - No need for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can **provide higher availability** by having multiple (> 2) replicas and using the majority protocol



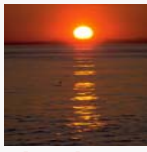
Coordinator Selection

- Backup coordinators
  - site which maintains enough information locally to assume the role of coordinator if the actual coordinator fails
  - executes the same algorithms and maintains the same internal state information as the actual coordinator;
  - allows fast recovery from coordinator failure
    - but involves overhead during normal processing because the coordinator and its backup need to communicate regularly
- Election algorithms
  - used to elect a new coordinator in case of failures
  - Example: Bully Algorithm - applicable to systems where every site can send a message to every other site.



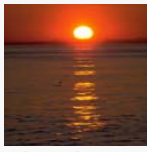
Bully Algorithm

- If site S_i sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed S_i tries to elect itself as the new coordinator.
- S_i sends an election message to every site with a higher identification number, S_i then waits for any of these processes to answer within *T*.
- If no response within *T*, assume that all sites with number greater than *i* have failed, S_i elects itself the new coordinator.
- If answer is received S_i begins time interval *T'*, waiting to receive a message that a site with a higher identification number has been elected.



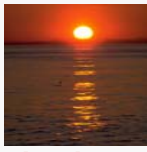
Bully Algorithm (Cont.)

- If no message is sent within *T*', assume the site with a higher number has failed; S_i restarts the algorithm.
- After a failed site recovers, it immediately begins execution of the same algorithm.
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.



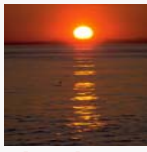
Chapter 22: Distributed Databases

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



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.



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
- Query: find all tuples of *account* relation
- Consider the horizontal fragmentation of the *account* relation into

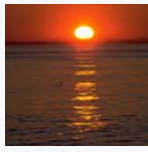
  $account_1 = \sigma_{branch\_name = \text{“Hillside”}}(account)$

  $account_2 = \sigma_{branch\_name = \text{“Valleyview”}}(account)$
- The query $\sigma_{branch\_name = \text{“Hillside”}}(account)$ becomes

  $\sigma_{\text{branch\_name} = \text{“Hillside”}}(account_1 \cup account_2)$

  which is optimized into

  $\sigma_{branch\_name = \text{“Hillside”}}(account_1) \cup \sigma_{branch\_name = \text{“Hillside”}}(account_2)$

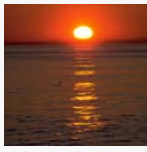


Example Query (Cont.)

- Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.

- Apply the definition of $account_2$ to obtain

  $\sigma_{branch\_name = \text{“Hillside”}} (\sigma_{branch\_name = \text{“Valleyview”}} (account))$

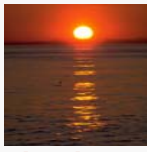


Simple Join Processing

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

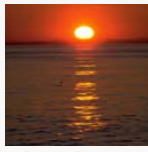
  $account \bowtie depositor \bowtie branch$

- *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

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I.
- 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.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

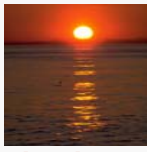


Semijoin Strategy

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

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

- This strategy is particularly useful when few tuples in r_2 contribute to the join.
  - The advantage in terms of cost savings results from having to ship only $temp_2$ rather than r_2 to S_1.



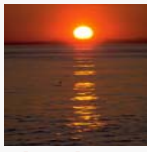
Formal Definition

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

$$r_1 \ltimes r_2$$

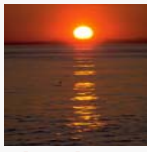
- it is defined by:
- $\Pi_{R1} (r_1 \bowtie r_2)$
- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 above, $temp_2 = r_2 \ltimes r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

- A lot of theoretical work has been developed regarding the use of semijoins for query optimization.



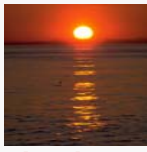
Join Strategies that Exploit Parallelism

- 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.
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2: simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they are produced without waiting for the entire join to be computed;
  S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1 in the same way
- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1, $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.



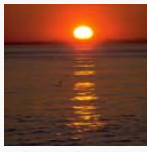
Chapter 22: Distributed Databases

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



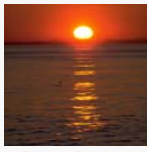
Heterogeneous Distributed Databases

- 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.
- A **multidatabase 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



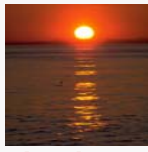
Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - Technical difficulties and cost of conversion
    - Organizational/political difficulties
      - Organizations do not want to give up control on their data
      - Local databases wish to retain a great deal of **autonomy**



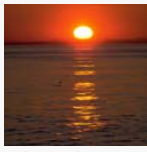
Unified View of Data

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



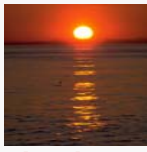
Query Processing

- Several issues in query processing in a heterogeneous database
- **Schema translation**
  - Write a **wrapper** for each data source to translate data to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema
- **Limited query capabilities**
  - Some data sources allow only restricted forms of selections
    - E.g. web forms, flat file data sources
  - Queries have to 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**



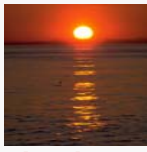
Mediator Systems

- **Mediator** systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing
  - But the terms mediator and multidatabase are sometimes used interchangeably
  - The term **virtual database** is also used to refer to mediator/multidatabase systems



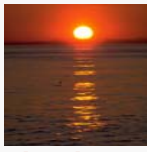
Chapter 22: Distributed Databases

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



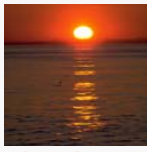
Directory Systems

- Typical kinds of directory information
  - Employee information such as name, id, email, phone, office addr, ..
  - Even personal information to be accessed from multiple places
    - e.g. Web browser bookmarks
- White pages
  - Entries organized by name or identifier
    - Meant for forward lookup to find more about an entry
- Yellow pages
  - Entries organized by properties
  - For reverse lookup to find entries matching specific requirements
- When directories are to be accessed across an organization
  - Alternative 1: Web interface. Not great for programs
  - Alternative 2: Specialized **directory access protocols**
    - Coupled with specialized user interfaces



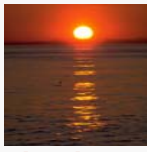
Directory Access Protocols

- Most commonly used directory access protocol:
  - LDAP (Lightweight Directory Access Protocol)
  - Simplified from earlier X.500 protocol
- Question: Why not use database protocols like ODBC/JDBC?
- Answer:
  - Simplified protocols for a **limited type** of data access, evolved parallel to ODBC/JDBC
  - Provide a nice **hierarchical naming** mechanism similar to file system directories
    - Data can be partitioned amongst multiple servers for different parts of the hierarchy, yet give a single view to user
      - E.g. different servers for Bell Labs Murray Hill and Bell Labs Bangalore
  - Directories may use databases as storage mechanism



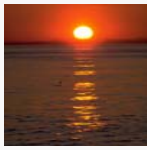
LDAP: Lightweight Directory Access Protocol

- LDAP Data Model
- Data Manipulation
- Distributed Directory Trees



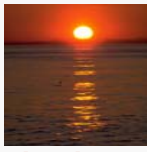
LDAP Data Model

- LDAP directories store **entries**
  - Entries are similar to objects
- Each entry must have unique **distinguished name (DN)**
- DN made up of a sequence of **relative distinguished names (RDNs)**
- E.g. of a DN
  - cn=Silberschatz, ou-Bell Labs, o=Lucent, c=USA
  - Standard RDNs (can be specified as part of schema)
    - cn: common name ou: organizational unit
    - o: organization c: country
  - Similar to paths in a file system but written in reverse direction



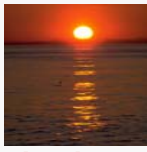
LDAP Data Model (Cont.)

- Entries can have attributes
  - Attributes are multi-valued by default
  - LDAP has several built-in types
    - Binary, string, time types
    - Tel: telephone number PostalAddress: postal address
- LDAP allows definition of **object classes**
  - Object classes specify attribute names and types
  - Can use inheritance to define object classes
  - Entry can be specified to be of one or more object classes
    - No need to have single most-specific type



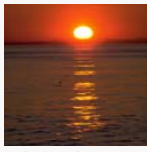
LDAP Data Model (cont.)

- Entries organized into a **directory information tree** according to their DNs
  - **Leaf level** usually represent specific objects
  - **Internal node** entries represent objects such as organizational units, organizations or countries
  - Children of a node **inherit** the DN of the parent, and add on RDNs
    - E.g. internal node with DN c=USA
      - Children nodes have DN starting with c=USA and further RDNs such as o or ou
    - DN of an entry can be generated by traversing path from root
  - Leaf level can be an **alias** pointing to another entry
    - Entries can thus have more than one DN
      - E.g. person in more than one organizational unit



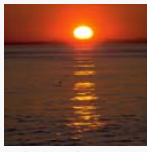
LDAP Data Manipulation

- Unlike SQL, LDAP does not define DDL or DML
- Instead, it defines a network protocol for DDL and DML
  - Users use an API or vendor specific front ends
  - LDAP also defines a file format
    - LDAP Data Interchange Format (LDIF)
- Querying mechanism is very simple: only selection & projection



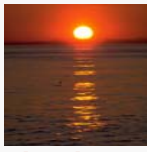
LDAP Queries

- LDAP query must specify
  - **Base**: a node in the DIT from where search is to start
  - **A search condition**
    - Boolean combination of conditions on attributes of entries
      - Equality, wild-cards and approximate equality supported
  - **A scope**
    - Just the base, the base and its children, or the entire subtree from the base
  - Attributes to be returned
  - Limits on number of results and on resource consumption
  - May also specify whether to automatically dereference aliases
- LDAP URLs are one way of specifying query
- LDAP API is another alternative



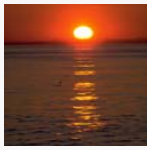
LDAP URLs

- First part of URL specifis server and DN of base
  - ldap::/aura.research.bell-labs.com/o=Lucent,c=USA
- Optional further parts separated by ? symbol
  - ldap::/aura.research.bell-labs.com/o=Lucent,c=USA??sub?cn=Korth
  - Optional parts specify
    1. attributes to return (empty means all)
    2. Scope (sub indicates entire subtree)
    3. Search condition (cn=Korth)



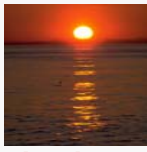
C Code using LDAP API

```
#include <stdio.h>
#include <ldap.h>
main( ) {
        LDAP *ld;
        LDAPMessage *res, *entry;
        char *dn, *attr, *attrList [ ] = {“telephoneNumber”, NULL};
        BerElement *ptr;
        int vals, i;
            // Open a connection to server
        ld = ldap_open(“aura.research.bell-labs.com”, LDAP_PORT);
        ldap_simple_bind(ld, “avi”, “avi-passwd”);
         … actual query (next slide) …
      ldap_unbind(ld);
}
```



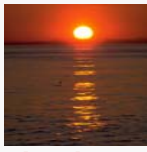
C Code using LDAP API (Cont.)

```
ldap_search_s(ld, “o=Lucent, c=USA”, LDAP_SCOPE_SUBTREE,
                    “cn=Korth”, attrList, /* attrsonly*/ 0, &res);
          /*attrsonly = 1 => return only schema not actual results*/
printf(“found%d entries”, ldap_count_entries(ld, res));
for (entry=ldap_first_entry(ld, res); entry != NULL;
                    entry=ldap_next_entry(id, entry)) {
       dn = ldap_get_dn(ld, entry);
       printf(“dn: %s”, dn);   /* dn: DN of matching entry */
       ldap_memfree(dn);
       for(attr = ldap_first_attribute(ld, entry, &ptr); attr != NULL;
            attr = ldap_next_attribute(ld, entry, ptr))
     {                                  // for each attribute
          printf(“%s:”, attr);              // print name of attribute
          vals = ldap_get_values(ld, entry, attr);
          for (i = 0; vals[i] != NULL; i ++)
                    printf(“%s”, vals[i]);  // since attrs can be multivalued
          ldap_value_free(vals);
     }
}
ldap_msgfree(res);
```



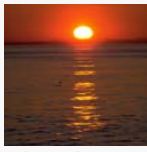
LDAP API (Cont.)

- LDAP API also has functions to create, update and delete entries
- Each function call behaves as a separate transaction
  - LDAP does not support atomicity of updates



Distributed Directory Trees

- Organizational information may be split into multiple directory information trees
  - Suffix of a DIT gives RDN to be tagged onto all entries to get an overall DN
    - E.g. two DITs, one with suffix o=Lucent, c=USA
      and another with suffix o=Lucent, c=India
  - Organizations often split up DITs based on **geographical location** or by organizational structure
  - Many LDAP implementations **support replication** (master-slave or multi-master replication) of DITs (not part of LDAP 3 standard)
- A node in a DIT may be a **referral** to a node in another DIT
  - E.g. Ou= Bell Labs may have a separate DIT, and DIT for o=Lucent may have a leaf with ou=Bell Labs containing a referral to the Bell Labs DIT
  - Referrals are the **key to integrating** a distributed collection of directories
  - When a server gets a query reaching a referral node, it may either
    - **Forward query** to referred DIT and return answer to client, or
    - **Give referral back to client**, which transparently sends query to referred DIT (without user intervention)



End of Chapter