Contents

- Review: Layered DBMS Architecture
- Distributed DBMS Architectures
  - DDBMS Taxonomy
- Client/Server Models
- Key Problems of Distributed DBMS
  - Distributed data modeling
  - Distributed query processing & optimization
  - Distributed transaction management
    - Failure recovery

See the Pensum and Anbefalt on the class webpage:
## Functional Layers of a DBMS

<table>
<thead>
<tr>
<th>Interface</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Semantic Integrity Control / Authorization</td>
</tr>
<tr>
<td>Compilation</td>
<td>Query Processing and Optimization</td>
</tr>
<tr>
<td>Execution</td>
<td>Storage Structures</td>
</tr>
<tr>
<td>Data Access</td>
<td>Buffer Management</td>
</tr>
<tr>
<td>Consistency</td>
<td>Concurrency Control / Logging</td>
</tr>
</tbody>
</table>

---

## Dependencies among DBMS components

- Log component (with savepoint mgmt)
- Access Path Mgmt
- Transaction Mgmt
- Sorting component
- Lock component
- System Buffer Mgmt

Indicates a dependency
Centralized DBS
- logically integrated
- physically centralized

Traditionally: one large mainframe DBMS + n “stupid” terminals

Distributed DBS
- Data logically integrated (i.e., access based on one schema)
- Data physically distributed among multiple database nodes
- Processing is distributed among multiple database nodes

Why a Distributed DBS?
Performance via parallel execution
- more users
- quick response
More data volume
- max disks on multiple nodes

Traditionally: m mainframes for the DBMSs + n terminals
### DBMS Implementation Alternatives

- **Distribution**
  - Centralized
  - Distributed
  - Federated
  - Multi-DBMS

- **Autonomy**
  - Homogeneous DBMS
  - Heterogeneous DBMS

- **Heterogeneity**
  - Centralized
  - Distributed

### Common DBMS Architectural Configurations

#### No Autonomy
- Fully integrated nodes
- Complete cooperation on:
  - Data Schema
  - Transaction Mgmt
- Fully aware of all other nodes in the DDB

#### Federated
- Independent DBMSs
- Implement some "cooperation functions":
  - Transaction Mgmt
  - Schema mapping
- Aware of other DBs in the federation

#### Multi
- Fully independent DBMSs
- Unaware of GTM and other DBs in the MultiDB

### Details

- **Complete cooperation on**:
  - Data Schema
  - Transaction Mgmt

- **Aware of other DBs in the federation**
Parallel Database Platforms (A form of Non-autonomous Distributed DBMS)

Shared Everything

Typically a special-purpose hardware configuration, but this can be emulated in software.

Shared Nothing

Can be a special-purpose hardware configuration, or a network of workstations.

Distributed DBMS

- Advantages:
  - Improved performance
  - Efficiency
  - Extensibility (addition of new nodes)
  - Transparency of distribution
    - Storage of data
    - Query execution
  - Autonomy of individual nodes

- Problems:
  - Complexity of design and implementation
  - Data consistency
  - Safety
  - Failure recovery
Client/Server Database Systems

The “Simple” Case of Distributed Database Systems?

Client/Server Environments

- objects stored and administered on server
- objects processed (accessed and modified) on workstations [sometimes on the server too]
- CPU-time intensive applications on workstations
  - GUIs, design tools
- Use client system local storage capabilities
- combination with distributed DBS services
  => distributed server architecture

data (object) server + n smart clients (workstations)
Clients with Centralized Server Architecture

Clients with Distributed Server Architecture
In the Client/Server DBMS Architecture, how are the database services organized? 

*There are several architectural options!*
Client/Server Architectures

Relational
- Application (client process)
- SQL
- DBMS
- Object/Page Cache Management
- Server process

Object-Oriented
- Application (client process)
- Objects, Pages, or Files
- Server process
- Object/Page Cache Management

Object Server Architecture

- Application (client process)
- Object Manager
- Object Cache
- Log/Lock Manager
- Object Manager
- File/Index Manager
- Page Cache Manager
- Storage Allocation and I/O
- Database

©2000 Vera Goebel
Object Server Architecture - summary

- Unit of transfer: object(s)
- Server understands the object concept and can execute methods
- Most DBMS functionality is replicated on client(s) and server(s)

Advantages:
- Server and client can run methods, workload can be balanced
- Simplifies concurrency control design (centralized in server)
- Implementation of object-level locking
- Low cost for enforcing “constraints” on objects

Disadvantages/problems:
- Remote procedure calls (RPC) for object references
- Complex server design
- Client cache consistency problems
- Page-level locking, objects get copied multiple times, large objects

Page Server Architecture

- Client process
  - Application
  - Object Manager
  - File/Index Manager
  - Page Cache Manager

- Server process
  - Log/Lock Manager
  - Page Cache Manager
  - Storage Allocation and I/O

- Database
  - Pages
  - Page references
  - Locks
  - Log records
Page Server Architecture - summary

- unit of transfer: page(s)
- server deals only with pages (understands no object semantics)
- server functionality: storage/retrieval of page(s), concurrency control, recovery
- advantages:
  - most DBMS functionality at client
  - page transfer -> server overhead minimized
  - more clients can be supported
  - object clustering improves performance considerably
- disadvantages/problems:
  - method execution only on clients
  - object-level locking
  - object clustering
  - large client buffer pool required
File Server Architecture - summary

- unit of transfer: page(s)
- simplification of page server, clients use remote file system (e.g., NFS) to read and write DB page(s) directly
- server functionality: I/O handling, concurrency control, recovery
- advantages:
  - same as for page server
  - NFS: user-level context switches can be avoided
  - NFS widely-used -> stable SW, will be improved
- disadvantages/problems:
  - same as for page server
  - NFS write are slow
  - read operations bypass the server -> no request combination
  - object clustering
  - coordination of disk space allocation

Cache Consistency in Client/Server Architectures

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Asynchronous</th>
<th>Deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Avoidance-Based Algorithms</strong></td>
<td>Client sends 1 msg per lock to server; Client waits; Server replies with ACK or NACK.</td>
<td>Client sends 1 msg per lock to the server; Client continues; Server invalidates cached copies at other clients.</td>
<td>Client sends all write lock requests to the server at commit time; Client waits; Server replies when all cached copies are freed.</td>
</tr>
<tr>
<td><strong>Detection-Based Algorithms</strong></td>
<td>Client sends object status query to server for each access; Client waits; Server replies.</td>
<td>Client sends 1 msg per lock to the server; Client continues; After commit, the server sends updates to all cached copies.</td>
<td>Client sends all write lock requests to the server at commit time; Client waits; Server replies based on W-W conflicts only.</td>
</tr>
</tbody>
</table>

**Best Performing Algorithms**
Comparison of the 3 Client/Server Architectures

Page & File Server
- Simple server design
- Complex client design
- Fine grained concurrency control difficult
- Very sensitive to client buffer pool size and clustering

Object Server
- Complex server design
- “Relatively” simple client design
- Fine-grained concurrency control
- Reduces data movement, relatively insensitive to clustering
- Sensitive to client buffer pool size

Conclusions:
- No clear winner
- Depends on object size and application’s object access pattern
- File server ruled out by poor NFS performance

Problems in Distributed DBMS Services

Distributed Database Design
Distributed Directory/Catalogue Mgmt
Distributed Query Processing and Optimization
Distributed Transaction Mgmt
- Distributed Concurrency Control
- Distributed Deadlock Mgmt
- Distributed Recovery Mgmt
**Distributed Storage in Relational DBMSs**

- **horizontal fragmentation**: distribution of "rows", selection
- **vertical fragmentation**: distribution of "columns", projection
- **hybrid fragmentation**: "projected columns" from "selected rows"
- **allocation**: which fragment is assigned to which node?
- **replication**: multiple copies at different nodes, how many copies?

- Design factors:
  - Most frequent query access patterns
  - Available distributed query processing algorithms
- **Evaluation Criteria**
  - Cost metrics for: network traffic, query processing, transaction mgmt
  - A system-wide goal: Maximize throughput or minimize latency

**Distributed Storage in OODBMSs**

- Must fragment, allocate, and replicate object data among nodes
- **Complicating Factors**:
  - Encapsulation hides object structure
  - Object methods (bound to class not instance)
  - Users dynamically create new classes
  - Complex objects
  - Effects on garbage collection algorithm and performance
- **Approaches**:
  - Store objects in relations and use RDBMS strategies to distribute data
    - All objects are stored in binary relations [OID, attr-value]
    - One relation per class attribute
    - Use nested relations to store complex (multi-class) objects
  - Use object semantics
- **Evaluation Criteria**:
  - Affinity metric (maximize)
  - Cost metric (minimize)
Horizontal Partitioning using Object Semantics

- Divide instances of a class into multiple groups
  - Based on selected attribute values
    - Attr A < 100
    - Attr A >= 100
  - Based on subclass designation
    - Class C
    - Node1: Subcl X
    - Subcl Y: Node2
- Fragment an object’s complex attributes from its simple attributes
- Fragment an object’s attributes based on typical method invocation sequence
  - Keep sequentially referenced attributes together

Vertical Partitioning using Object Semantics

- Fragment object attributes based on the class hierarchy
  - Instances of Class X
  - Instances of Subclass subX
  - Instances of Subclass subsubX

Breaks object encapsulation?
Path Partitioning using Object Semantics

- Use the object composition graph for complex objects
- A path partition is the set of objects corresponding to instance variables in the subtree rooted at the composite object
  - Typically represented in a "structure index"

<table>
<thead>
<tr>
<th>Composite Object OID</th>
<th>(SubC#1-OID, SubC#2-OID, …SubC#n-OID)</th>
</tr>
</thead>
</table>

More Issues in OODBMS Fragmentation

<table>
<thead>
<tr>
<th>Local Methods</th>
<th>Remote Object Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good performance; Issue: Replicate methods so they are local to all instances?</td>
<td>Fetch the remote data and execute the methods; Issue: Time/cost for data transfer?</td>
</tr>
<tr>
<td>Send data to remote site, execute, and return result OR fetch the method and execute; Issues: Time/cost for two transfers? Ability to execute locally?</td>
<td>Send additional input values via RPC, execute and return result; Issues: RPC time? Execution load on remote node?</td>
</tr>
</tbody>
</table>

- Replication Options
  - Objects
  - Classes (collections of objects)
  - Methods
  - Class/Type specifications
**Distributed Directory and Catalogue Management**

- Directory Information:
  - Description and location of records/objects
    - Size, special data properties (e.g., executable, DB type, user-defined type, etc.)
    - Fragmentation scheme
  - Definitions for views, integrity constraints
- Options for organizing the directory:
  - Centralized — Issues: bottleneck, unreliable
  - Fully replicated — Issues: consistency, storage overhead
  - Partitioned — Issues: complicated access protocol, consistency
  - Combination of partitioned and replicated — e.g., zoned, replicated zoned

**Distributed Query Processing and Optimization**

- Construction and execution of query plans, query optimization
- **Goals**: maximize parallelism (response time optimization) minimize network data transfer (throughput optimization)
- Basic Approaches to distributed query processing:
  - Pipelining — functional decomposition
    
    \[
    \text{Rel A} \rightarrow \text{Node 1} \rightarrow \text{Node 2} \rightarrow \text{Rel B}
    \]
    
    \[\text{Select A.x > 100} \quad \text{Join A and B on y}\]
  - Parallelism — data decomposition
    
    \[
    \text{Frag A.1} \rightarrow \text{Node 1} \rightarrow \text{Node 2} \rightarrow \text{Node 3}
    \]
    
    \[\text{Select A.x > 100} \quad \text{Union}\]
Creating the Distributed Query Processing Plan

- **Factors** to be considered:
  - distribution of data
  - communication costs
  - lack of sufficient locally available information

- **4 processing steps**:
  1. query decomposition
  2. data localization
  3. global optimization
  4. local optimization

Generic Layered Scheme for Planning a Dist. Query

- calculus query on distributed objects
  - query decomposition → algebraic query on distributed objects
  - data localization → fragment query
  - global optimization → optimized fragment query with communication operations
  - local optimization → optimized local queries

This approach was initially designed for distributed relational DBMSs.

It also applies to distributed OODBMSs, Multi-DBMSs, and distributed Multi-DBMSs.
### Distributed Query Processing Plans in OODBMSs

- Two forms of queries
  - Explicit queries written in OQL
  - Object navigation/traversal
- Planning and Optimizing
  - Additional rewriting rules for sets of objects
    - Union, Intersection, and Selection
  - Cost function
    - Should consider object size, structure, location, indexes, etc.
    - Breaks object encapsulation to obtain this info?
    - Objects can “reflect” their access cost estimate
  - Volatile object databases can invalidate optimizations
    - Dynamic Plan Selection (compile N plans; select 1 at runtime)
    - Periodic replan during long-running queries

Reduce to logical algebraic expressions

### Distributed Query Optimization

- Information needed for optimization (fragment statistics):
  - size of objects, image sizes of attributes
  - transfer costs
  - workload among nodes
  - physical data layout
  - access path, indexes, clustering information
  - properties of the result (objects) - formulas for estimating the cardinalities of operation results
- **Execution cost** is expressed as a weighted combination of I/O, CPU, and communication costs (mostly dominant).

\[
\text{total-cost} = C_{\text{CPU}} \times \#\text{insts} + C_{\text{IO}} \times \#\text{I/Os} + C_{\text{MSG}} \times \#\text{msgs} + C_{\text{TR}} \times \#\text{bytes}
\]

Optimization Goals: response time of single transaction or system throughput
**Distributed Transaction Management**

- Transaction Management (TM) in centralized DBS
  - Achieves transaction ACID properties by using:
    - concurrency control (CC)
    - recovery (logging)
- TM in DDBS
  - Achieves transaction ACID properties by using:

**Classification of Concurrency Control Approaches**

- CC Approaches
  - Pessimistic
    - Locking
    - Timestamp Ordering
  - Optimistic
    - Locking
    - Timestamp Ordering

- Phases of pessimistic transaction execution:
  - validate
  - read
  - compute
  - write

- Phases of optimistic transaction execution:
  - read
  - compute
  - validate
  - write
Two-Phase-Locking Approach (2PL)

Communication Structure of Centralized 2PL

- **Obtain Lock**
- **Release Lock**
- **Transaction Duration**
- **Number of locks**
- **Period of data item use**
- **Lock Granted**
- **Operation**
- **End of Operation**
- **Release Locks**
Communication Structure of Distributed 2PL

- Data Processors at participating sites
- Participating LMs
- Coordinating TM

1. Lock Request
2. Operation
3. End of Operation
4. Release Locks

Distributed Deadlock Management

Example

Site X
- T1 x holds lock Lc
- T2 x holds lock Lb
- T1 x needs a lock for T1 on site y to complete
- T2 y waits for T1 on site y to complete
- T1 y waits for T2 on site x to release Ld
- T2 y waits for T1 on site x to complete

Site Y
- T1 y holds lock La
- T2 y holds lock Ld

Distributed Waits-For Graph
- Requires many messages to update lock status
- Combine the “wait-for” tracing message with lock status update messages
- Still an expensive algorithm

Isolation
### Communication Structure of Centralized 2P Commit Protocol

#### Coordinating TM | Participating Sites
---|---
1. Prepare | Make local commit/abort decision
2. Vote-Commit or Vote-Abort | Write log entry
3. Global-Abort or Global-Commit | Count the votes
   - If missing any votes or any Vote-Abort
   - Then Global-Abort
   - Else Global-Commit
4. ACK | Write log entry

- **Who participates?** Depends on the CC alg.
- Other communication structures are possible:
  - Linear
  - Distributed

### State Transitions for 2P Commit Protocol

#### Advantages:
- Preserves atomicity
- All processes are “synchronous within one state transition”

#### Disadvantages:
- Many, many messages
- If failure occurs, the 2PC protocol blocks!

At tempted solutions for the blocking problem:
1. Termination Protocol
2. 3-Phase Commit
3. Quorum 3P Commit

---

©2000 Vera Goebel
Failures in a Distributed System

Types of Failure:
- Transaction failure
- Node failure
- Media failure
- Network failure
  - Partitions each containing 1 or more sites

Who addresses the problem?
- Termination Protocols
- Modified Concurrency Control & Commit/Abort Protocols
- Recovery Protocols, Termination Protocols, & Replica Control Protocols

Issues to be addressed:
- How to continue service
- How to maintain ACID properties while providing continued service
- How to ensure ACID properties after recovery from the failure(s)

Termination Protocols

Use timeouts to detect potential failures that could block protocol progress

Timeout states:
- Coordinator: wait, commit, abort
- Participant: initial, ready

Coordinator Termination Protocol:
- Wait – Send global-abort
- Commit or Abort – BLOCKED!

Participant Termination Protocol:
- Ready – Query the coordinator
- If timeout then query other participants;
  - If global-abort or global-commit then proceed and terminate else BLOCKED!
Replica Control Protocols

- Update propagation of committed write operations

Number of locks

BEGIN Obtain Locks Period of data use 2-Phase Commit Release Locks END

STRICT UPDATE LAZY UPDATE

2-Phase Commit

Consistency

Strict Replica Control Protocol

- Read-One-Write-All (ROWA)
- Part of the Concurrency Control Protocol and the 2-Phase Commit Protocol
  - CC locks all copies
  - 2PC propagates the updated values with 2PC messages (or an update propagation phase is inserted between the wait and commit states for those nodes holding an updateable value).
Lazy Replica Control Protocol

- Propagates updates from a primary node.
- Concurrency Control algorithm locks the primary copy node (same node as the primary lock node).
- To preserve single copy semantics, must ensure that a transaction reads a current copy.
  - Changes the CC algorithm for read-locks
  - Adds an extra communication cost for reading data
- Extended transaction models may not require single copy semantics.

Recovery in Distributed Systems

Select COMMIT or ABORT (or blocked) for each interrupted subtransaction

Commit Approaches:
- Redo – use the undo/redo log to perform all the write operations again
- Retry – use the transaction log to redo the entire subtransaction (R + W)

Abort Approaches:
- Undo – use the undo/redo log to backout all the writes that were actually performed
- Compensation – use the transaction log to select and execute “reverse” subtransactions that semantically undo the write operations.

Implementation requires knowledge of:
- Buffer manager algorithms for writing updated data from volatile storage buffers to persistent storage
- Concurrency Control Algorithm
- Commit/Abort Protocols
- Replica Control Protocol
Network Partitions in Distributed Systems

Issues:
- Termination of interrupted transactions
- Partition integration upon recovery from a network failure
- Data availability while failure is ongoing

Data Availability in Partitioned Networks

- Concurrency Control model impacts data availability.
- ROWA – data replicated in multiple partitions is not available for reading or writing.
- Primary Copy Node CC – can execute transactions if the primary copy node for all of the read-set and all of the write-set are in the client’s partition.

[Availability is still very limited . . . We need a new idea!]
Quorums

- Quorum – a special type of majority
- Use quorums in the Concurrency Control, Commit/Abort, Termination, and Recovery Protocols
  - CC uses read-quorum & write-quorum
  - C/A, Term, & Recov use commit-quorum & abort-quorum
- Advantages:
  - More transactions can be executed during site failure and network failure (and still retain ACID properties)
- Disadvantages:
  - Many messages are required to establish a quorum
  - Necessity for a read-quorum slows down read operations
  - Not quite sufficient (failures are not “clean”)

Read-Quorums and Write-Quorums

- The Concurrency Control Algorithm serializes valid transactions in a partition. It must obtain
  - A read-quorum for each read operation
  - A write-quorum for each write operation
- Let N=total number of nodes in the system
- Define the size of the read-quorum Nr and the size of the write-quorum Nw as follows:
  - Nr + Nw > N
  - Nw > (N/2)
- When failures occur, it is possible to have a valid read quorum and no valid write quorum

ACID

Isolation

Simple Example:

N = 8
Nr = 4
Nw = 5
Commit-Quorums and Abort-Quorums

- The Commit/Abort Protocol requires votes from all participants to commit or abort a transaction.
  - Commit a transaction if the accessible nodes can form a commit-quorum
  - Abort a transaction if the accessible nodes can form an abort-quorum
  - Elect a new coordinator node (if necessary)
  - Try to form a commit-quorum before attempting to form an abort-quorum
- Let $N=$ total number of nodes in the system
- Define the size of the commit-quorum $N_c$ and the size of abort-quorum $N_a$ as follows:
  - $N_a + N_c > N$; $0 \leq N_a, N_c \leq N$

<table>
<thead>
<tr>
<th>Simple Examples:</th>
<th>$N = 7$</th>
<th>$N_c = 4$</th>
<th>$N_a = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N = 7$</td>
<td>$N_c = 5$</td>
<td>$N_a = 3$</td>
</tr>
</tbody>
</table>

Conclusions

- Nearly all commercial relational database systems offer some form of distribution
  - Client/server at a minimum
- Only a few commercial object-oriented database systems support distribution beyond $N$ clients and 1 server
- Future research directions:
  - Distributed data architecture and placement schemes (app-influenced)
  - Distributed databases as part of Internet applications
  - Continued service during disconnection or failure
    - Mobile systems accessing databases
    - Relaxed transaction models for semantically-correct continued operation