Unit 1: Introduction to Distributed Database

Distributed Data Processing

- It is a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks
- Distributed data processing refers to the practice of processing and managing data across multiple computers or nodes in a network. This approach offers several benefits such as improved performance, scalability, fault tolerance, and efficient resource utilization.

Advantages of Distributed Data Processing:

1. **Parallelism and Performance Improvement:** Distributing data processing tasks allows multiple nodes to work on different parts of a problem simultaneously, leading to faster execution and improved overall performance.
2. **Scalability**: Distributed processing systems can be easily scaled by adding more nodes to the network. This makes it feasible to handle larger amounts of data and more complex tasks as the system's requirements grow.
3. **Fault Tolerance and Reliability**: With data distributed across multiple nodes, the system becomes more resilient to hardware failures. If one node fails, the processing can continue on other nodes, ensuring high availability.
4. **Reduced Latency**: Data can be processed closer to where it's generated or needed, reducing network latency and improving response times. This is particularly important for real-time applications.
5. **Cost Efficiency**: Distributed processing can make use of commodity hardware, which is often more cost-effective than investing in a single powerful server.
6. **Geographical Distribution:** Distributed data processing is well-suited for applications that need to serve users across different geographical locations, as data can be processed locally.

Disadvantages of Distributed Data Processing:

1. **Complexity:** Managing a distributed system is inherently more complex than a centralized system. It requires expertise in networking, load balancing, data partitioning, and fault tolerance mechanisms.
2. **Data Consistency:** Maintaining data consistency across distributed nodes is challenging. Ensuring that data is up-to-date and accurate can be complex, especially in scenarios involving frequent updates.
3. **Network Communication Overhead:** Data needs to be transmitted between nodes, which introduces network communication overhead. This can impact performance, particularly if nodes are geographically distant.
4. **Security Concerns:** Distributed systems can introduce security challenges, as data needs to traverse the network and nodes might be vulnerable to attacks. Ensuring data security and access control is crucial.
5. **Software Complexity:** Developing applications for distributed processing requires specialized tools and frameworks. Developers need to be familiar with parallel programming, concurrency, and potentially new programming models.
6. **Debugging and Monitoring:** Identifying and resolving issues in a distributed system can be more difficult due to its inherent complexity. Effective monitoring and debugging tools are essential.

Distributed Database Systems

A distributed database system is a collection of interconnected databases, where data is stored on multiple computers or nodes within a network. These databases work together as a single logical database, while physically being distributed across different locations. The goal of a distributed database system is to improve scalability, performance, fault tolerance, and data availability.

Advantages of Distributed Database Systems:

1. **Scalability:** Distributed databases can handle larger amounts of data and growing workloads by adding more nodes to the network. This makes it easier to accommodate increasing data demands.

2. **Improved Performance**: Data can be processed and retrieved from nodes that are geographically closer to the users or applications. This reduces network latency and enhances response times.
3. **Fault Tolerance:** Distributed databases are more resilient to hardware failures or network issues. If one node fails, the data and processing tasks can still be accessed from other nodes, ensuring high availability.
4. **Data Availability:** Since data is replicated across multiple nodes, even if a node goes down, the data remains accessible from other nodes, minimizing downtime.
5. **Load Balancing:** Distributing data across nodes helps balance the load on individual nodes, preventing resource bottlenecks and ensuring even resource utilization.
6. **Geographical Distribution:** Distributed databases are suitable for applications that serve users across different regions. Data can be stored locally to reduce latency and enhance user experience.

Disadvantages of Distributed Database Systems:

1. **Complexity**: Managing a distributed database system is more complex than a centralized one. It requires expertise in data partitioning, replication, consistency, and network management.
2. **Data Consistency:** Ensuring data consistency across distributed nodes can be challenging. Different nodes may have different versions of data, leading to issues unless handled properly.
3. **Network Communication Overhead**: Data needs to be transmitted between nodes, which introduces network overhead. This can impact performance, especially for data-intensive operations.
4. **Security and Privacy:** Distributed systems introduce security concerns as data traverses the network. Ensuring data security and access control across multiple nodes is vital.
5. **Cost:** Setting up and maintaining a distributed database system can involve higher costs compared to a centralized system, due to the need for multiple hardware resources, networking equipment, and specialized expertise.

6. **Consistency Trade-offs:** Achieving strong data consistency across all nodes can impact performance, while relaxing consistency might lead to potential conflicts or data anomalies.
7. **Complex Query Optimization:** Optimizing queries that involve data spread across multiple nodes requires sophisticated query optimization techniques.

Promises of DDBS

1. Transparency:

a. Data Transparency: Users and applications should be able to access and manipulate the data without needing to know its physical location or distribution. This simplifies the development process and shields users from the complexities of data distribution.
b. Fragmentation Transparency: Data fragmentation (splitting data into smaller parts for distribution) should be hidden from users and applications. They should interact with the data as if it were a single, cohesive unit.
c. Network Transparency: Users and applications should be able to access distributed data as if it were stored on a single system, regardless of the actual network structure and communication pathways.
d. Replication Transparency: The presence of data replicas should be transparent to users. They shouldn't need to know whether the data they're accessing is a replica or the original.

2. Reliability:

a. Fault Tolerance: Distributed databases should continue to function even if individual nodes or components fail. Replication of data across nodes ensures that data remains available even when certain nodes are inaccessible.
b. Redundancy: Data replication provides redundancy, allowing the system to continue working even when some nodes experience failures.

3. Performance:

a. Parallel Processing: By distributing data and processing tasks across multiple nodes, distributed databases can achieve parallelism, leading to faster query execution and improved performance.
b. Reduced Latency: Data can be accessed from nodes that are closer to the user or application, reducing network latency and improving respon se times.
c. Load Balancing: Distributing data and processing across nodes helps balance the workload, preventing resource bottlenecks and ensuring efficient resource utilization.

4. Scalability:

a. Horizontal Scalability: Additional nodes can be added to the distributed database system to handle increased workloads and growing data demands. This ability to scale out ensures that the system remains responsive as requirements change.
b. Vertical Scalability: In some cases, individual nodes can be upgraded to handle larger workloads, providing vertical scalability as well. **Complicating Factors**

1. Data Distribution and Fragmentation: Dividing and distributing data across multiple nodes while ensuring efficient query processing and load balancing is a complex task. Deciding on the appropriate fragmentation strategy and mapping data to nodes requires careful planning.
2. Data Consistency and Replication: Maintaining consistent and synchronized data across distributed replicas introduces challenges. Balancing the benefits of data availability with ensuring data consistency among replicas is a complex trade-off.
3. Concurrency Control: Coordinating concurrent transactions to prevent conflicts and maintain data integrity is intricate. Ensuring isolation and

consistency across distributed nodes without causing performance bottlenecks is a critical concern.

4. Query Optimization and Processing: Optimizing queries in a distributed environment involves considering factors like data distribution, network latency, and communication overhead. Developing efficient query plans that utilize distributed resources effectively is a challenge.
5. Network Communication and Latency: Network delays and limited bandwidth can impact response times and overall system performance. Designing strategies to minimize the effects of network latency on query execution is essential.
6. Fault Tolerance and Recovery: Dealing with node failures, network disruptions, and system crashes requires effective fault detection, recovery mechanisms, and backup strategies. Ensuring data availability and system reliability under such circumstances is a complex task.

Design Issues of DDBMS

1. Distributed Database Design:

Designing the structure of the distributed database involves decisions about data fragmentation, allocation, and distribution. This includes determining how to divide tables into fragments, which nodes will store which fragments, and how to handle relationships between distributed fragments.

2. Distributed Data Control:

Managing access to distributed data is a challenge. Distributed data control involves enforcing data access permissions, ensuring data consistency, and maintaining data integrity across multiple nodes while preserving the desired levels of security.

3. Distributed Query Processing:

Distributed query processing involves optimizing and executing queries that involve data stored on multiple nodes. Deciding how to distribute queries, retrieve data efficiently, and minimize data transfer across the network are key concerns.

4. Distributed Concurrency Control:

Ensuring data consistency in the presence of concurrent transactions is complex in a distributed environment. Distributed concurrency control involves managing locks, timestamps, and other mechanisms to avoid conflicts and maintain data integrity.

5. Reliability of Distributed DBMS:

Reliability focuses on ensuring the system's availability and data integrity, even in the face of hardware failures, network disruptions, and other issues. Techniques like data replication, backup strategies, and fault detection are used to enhance reliability.

6. Replication:

Replication involves creating and managing multiple copies of data across distributed nodes. It improves data availability and fault tolerance but introduces challenges related to data consistency, update propagation, and replication control.

7. Parallel DBMSs:

Parallel DBMSs leverage multiple processors and nodes to process queries and transactions concurrently, leading to improved performance. Designing effective parallel query execution plans, load balancing, and synchronization are central concerns.

8. Database Integration:

Integrating databases from different locations or systems requires handling heterogeneous data formats, resolving schema mismatches, and ensuring consistent data access across the integrated databases.

9. Big Data Processing and nosql:

Handling massive volumes of data in a distributed environment requires scalable architectures and efficient processing techniques. Technologies like MapReduce and Hadoop are used for distributed processing of big data. NoSQL databases offer flexible data models suitable for various types of unstructured or semi-structured data. Designing data models for NoSQL databases requires considering factors like schema flexibility, data partitioning, and eventual consistency.

Architectural Models for DDBMSs:

Architectural models for DDBMSs can be classified based on three key dimensions:

1. Autonomy:

Autonomy refers to the level of control individual DBMSs have and their ability to operate independently within a distributed environment.

- **Tight Integration:**

A single-image database is accessible to all users, enabling them to access information from multiple databases. One data manager controls user requests.

- **Semiautonomous Systems**:

Individual DBMSs can function independently but choose to share some local data within a federation.

- **Total Isolation:**

Each DBMS operates in isolation, unaware of other DBMSs, with no global control.

Autonomy Dimensions:

- Design Autonomy: Each DBMS can choose its preferred data models and transaction management techniques.
- Communication Autonomy: Each DBMS decides what data to share with other DBMSs.
- Execution Autonomy: Each DBMS can execute transactions as it sees fit.

2. Distribution:

Distribution involves the physical dispersion of data across multiple sites.

- **No Distribution:** No data distribution; data remains in one location.

- **Client/Server Distribution:**

Data is concentrated on a server, while clients provide application environments and user interfaces.

- **Peer-to-Peer Distribution:**

Machines have equal DBMS functionality; there's no distinction between client and server. Each machine has a complete DBMS.

3. Heterogeneity:

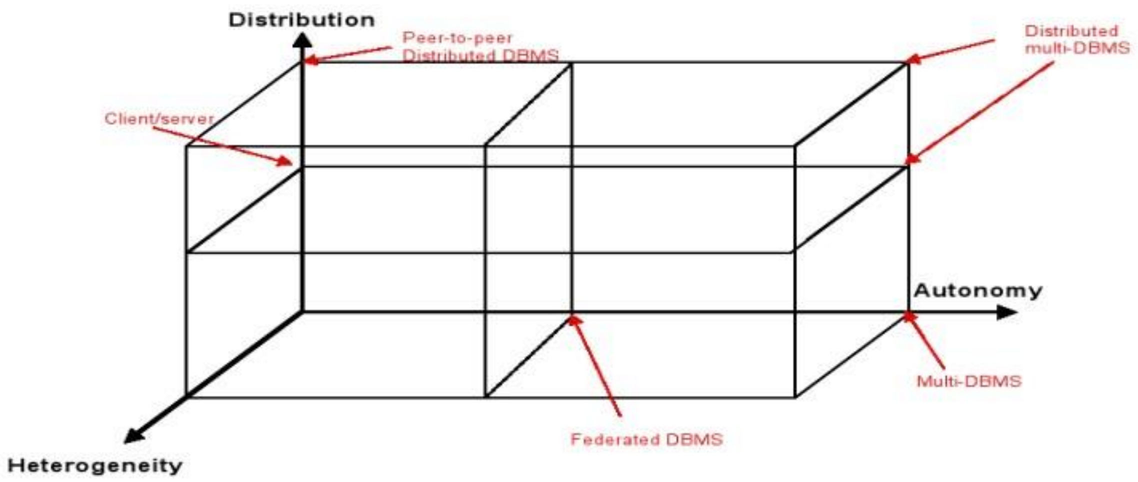
Heterogeneity refers to differences in components at various levels, including hardware, communications, operating systems, and database components.

- **Hardware Heterogeneity:** Differences in the physical hardware among machines.

- **Communications Heterogeneity:** Differences in communication mechanisms among machines.

- **Operating System Heterogeneity:** Variation in operating systems running on different machines.

- **DB Components Heterogeneity:** Differences in data models, query languages, and transaction management algorithms among DBMSs.



DDBMS Architecture

1. Client/Server Architecture:

In a Client/Server DDBMS architecture, the distribution of data and tasks is organized between clients and servers. Clients are responsible for user interfaces and application logic, while servers handle data storage and processing. This architecture enhances scalability, as more clients can be added without affecting data storage on the servers.

General Idea: This architecture divides the functionality of a distributed database system into two main classes: server functions and client functions.

1. Server Functions: Involve data management, query processing, optimization, and transaction management.
2. Client Functions: Include user interface and might also involve some data management tasks.

Two-Level Architecture: This architecture creates a clear distinction between servers responsible for data management and clients for user interactions.

Types of Client/Server:

1. Multiple Client/Single Server: Multiple clients communicate with a single server.
2. Multiple Client/Multiple Server: Multiple clients interact with multiple servers.

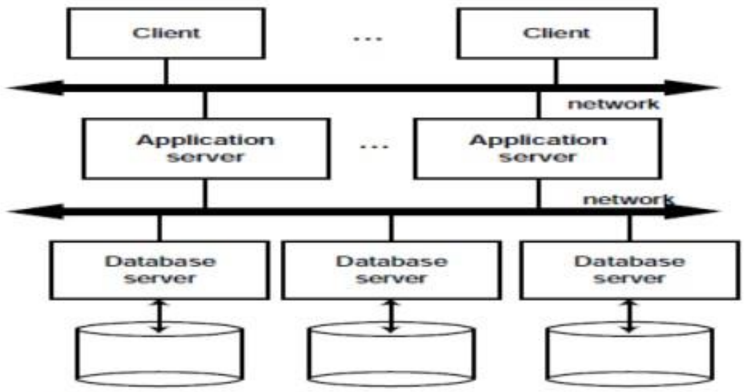


Fig. 1.13 Distributed Database Servers

- **Characteristics:**

- Clients initiate requests for data and operations.

- Servers manage data storage and execute queries.

- Clients and servers communicate over a network.
- Centralized data storage on servers allows for better management and data consistency.

- **Advantages:**
- Efficient utilization of server resources.
- Centralized data management simplifies administration.
- Clients can be lightweight and focused on user interactions.

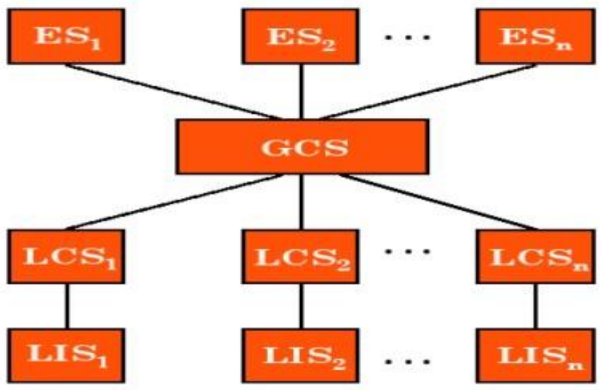
- **Disadvantages:**
- Load on servers can become a bottleneck.
- Network communication overhead between clients and servers.

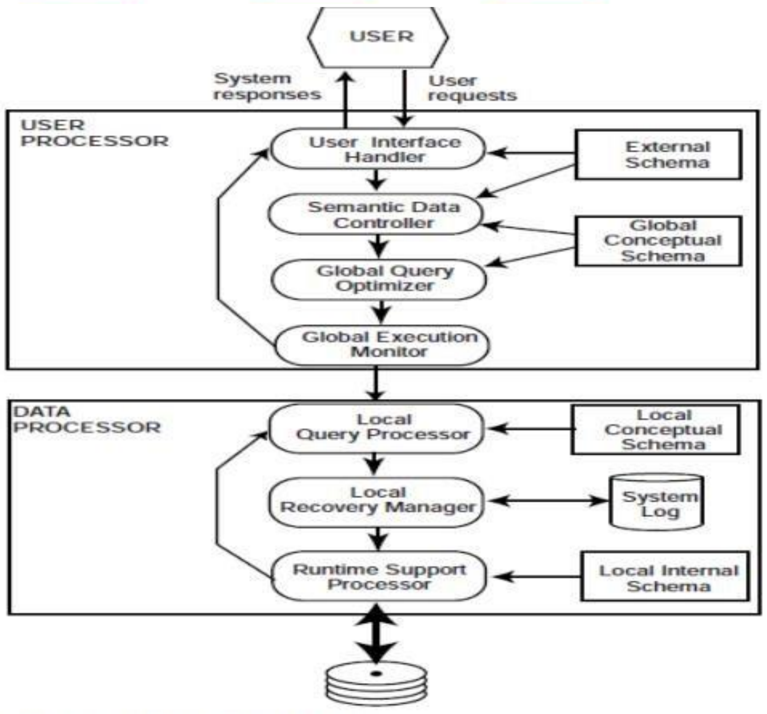
2. Peer-to-Peer Architecture:

In a Peer-to-Peer (P2P) DDBMS architecture, there is no clear distinction between clients and servers. Each node in the network acts both as a client and a server. All nodes have equal functionality, and data is distributed across the network in a decentralized manner. This architecture is suitable for environments with dynamic and frequently changing nodes.

This architecture involves multiple nodes (peers), where each peer acts as both a client and a server.

1. Local Internal Schema (LIS): Describes the local physical data organization, which can vary across nodes.
2. Local Conceptual Schema (LCS): Describes the logical data organization within each site. Necessary due to data fragmentation and replication.
3. Global Conceptual Schema (GCS): This represents the combined logical view of the data, formed by uniting LCSs of all sites.
4. External Schema (ES): Defines the user/application view of the data.





Components of a Distributed DBMS

- Characteristics:

- Each node can initiate queries and provide data.

- Nodes communicate directly with each other.

- Data distribution is more evenly spread across nodes.

- Advantages:

- Scalability due to the absence of central bottlenecks.

- Redundancy and fault tolerance through data replication.

- Decentralized nature supports dynamic and ad hoc networks.

- Disadvantages:

- Data consistency and integrity can be challenging to maintain.

- Network management complexity increases with the number of nodes.

3. Multi-Database System (MultiDBS) Architecture:

MultiDBS architecture involves multiple independent databases that are interconnected through a middleware. Each database can be managed by a different DBMS and can have its own schema and data model. Middleware provides mechanisms for data sharing and communication between the databases.

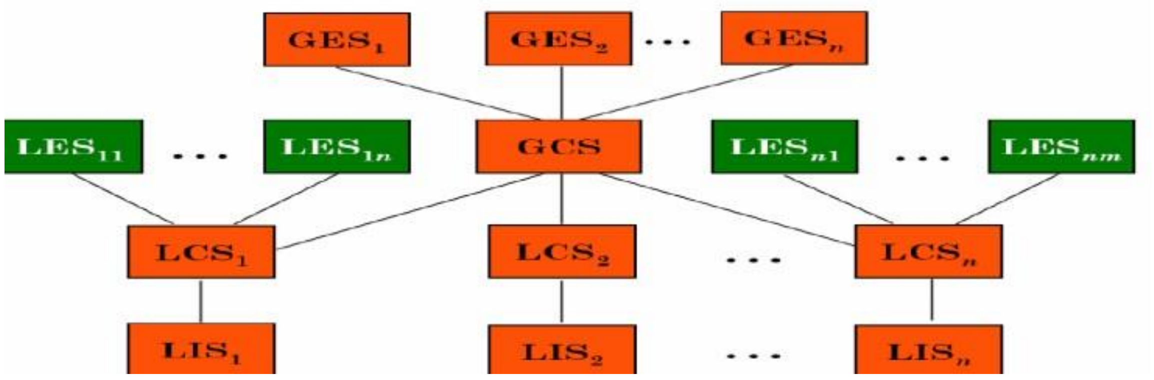
In this architecture, multiple autonomous database systems are integrated.

Global Conceptual Schema (GCS):

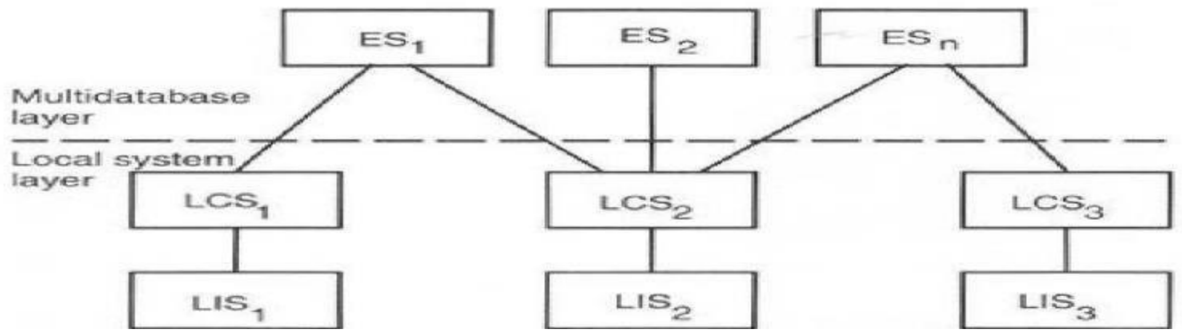
1. With GCS: GCS represents parts of the LCSs, forming a unified view of shared data.
2. Without GCS: Each local DBMS defines what part of its local DB it's willing to share.

Different Architecture Models:

1. Models with a GCS: GCS is a union of LCSs from various local databases.



2. Models without a GCS: Multi-database layer interacts with local DBMSs to create external views on top of LCSs.



Six Levels of Schemas in Multi-DBMS:

1. Multi-Database View Level: Depicts multiple user views, comprising subsets of the integrated distributed database.
2. Multi-Database Conceptual Level: Depicts the integrated multi-database, including global logical multi-database structure definitions.
3. Multi-Database Internal Level: Illustrates data distribution across different sites and mapping from multi-database to local data.
4. Local Database View Level: Displays the public view of local data at each site.
5. Local Database Conceptual Level: Depicts local data organization at each site.
6. Local Database Internal Level: Describes the physical data organization at each site.

- Characteristics:

  a. Databases can be of varying types (relational, object-oriented, etc.).
  b. Middleware handles data exchange and communication.
  c. Each database can operate autonomously.

- Advantages:

a. Support for different data models and schemas in separate databases.
b. Flexibility to choose DBMSs suitable for specific data requirements.
c. Modular approach allows for independent database management.

- Disadvantages:

a. Complexity in managing data sharing and integration.
b. Middleware can introduce a performance overhead.
c. Maintaining data consistency and integrity across databases can be challenging.

Unit 2: Distributed Database Design and Access Control

Top-Down Design Process

In the top-down approach to designing distributed database systems:

- Designing Systems from Scratch: This strategy involves starting with a clean slate. It means creating a new distributed database system from the ground up, considering the specific requirements and objectives of the distributed environment.
- Homogeneous Systems: The top-down approach often results in the creation of a unified and homogeneous distributed database system. This means that all components of the system, including data models, communication protocols, and software tools, are designed to work cohesively and consistently across all nodes.
- Centralized Control: The top-down approach often enforces centralized control over system design and architecture decisions. A centralized design team makes key choices regarding data distribution, replication strategies, query optimization techniques, and more.

Advantages of Top-Down Approach:

1. Holistic Design: Starting from scratch allows for a holistic design that ensures compatibility and consistency across the entire distributed system.
2. Optimal Resource Utilization: Designing from scratch enables the optimization of resources and performance, tailoring the system to the specific needs of the organization.
3. Uniform User Experience: Homogeneous design results in a consistent user experience regardless of the node being accessed.

Disadvantages of Top-Down Approach:

1. Time and Resources: Designing a distributed system from scratch can be time-consuming and resource-intensive.

2. Limited Adaptability: The initial design may struggle to accommodate unforeseen changes or evolving requirements.

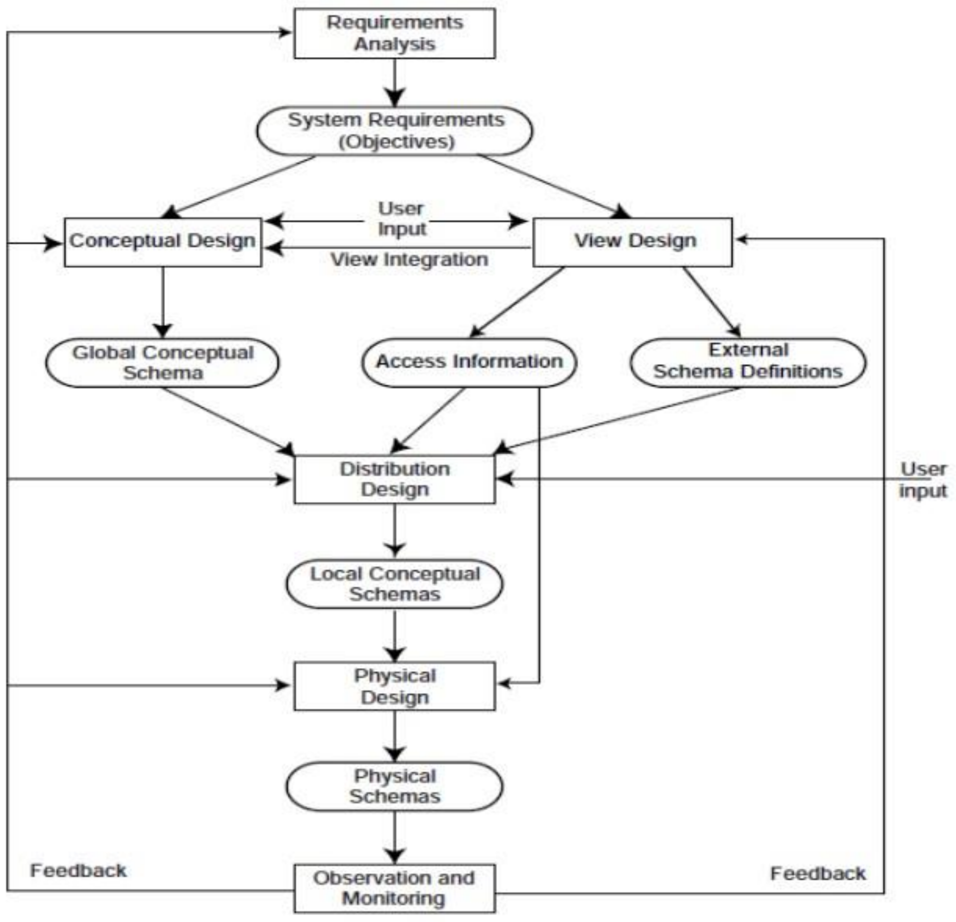


Fig: Top-Down Design Process

Bottom-Up Approach:

In the bottom-up approach to designing distributed database systems:

- The Databases Already Exist: The starting point is the existence of databases at different sites. These databases might have been developed independently and might use different data models, schema designs, and software tools.
- Connecting Existing Databases: The goal is to connect these existing databases to address common tasks or enable data sharing across different nodes.

- Heterogeneous Systems: The bottom-up approach often results in a heterogeneous distributed system where the individual databases might not have been originally designed to work together seamlessly.
- Decentralized Control: In this approach, control is more decentralized. Each site or node might have control over its local database, and decisions about data sharing, integration, and communication might be made more independently.

Advantages of Bottom-Up Approach:

1. Utilization of Existing Resources: This approach leverages existing databases and resources, reducing the need for starting from scratch.
2. Incremental Implementation: The ability to connect existing databases allows for a phased approach to implementing a distributed system.
3. Flexibility: The system can be more flexible and adaptable to the unique characteristics of each local database.

Disadvantages of Bottom-Up Approach:

1. Integration Challenges: Integrating databases with varying data models, schemas, and tools can be complex and require significant effort.
2. Data Inconsistencies: Data inconsistencies and conflicts might arise due to different data models and updates in different databases.

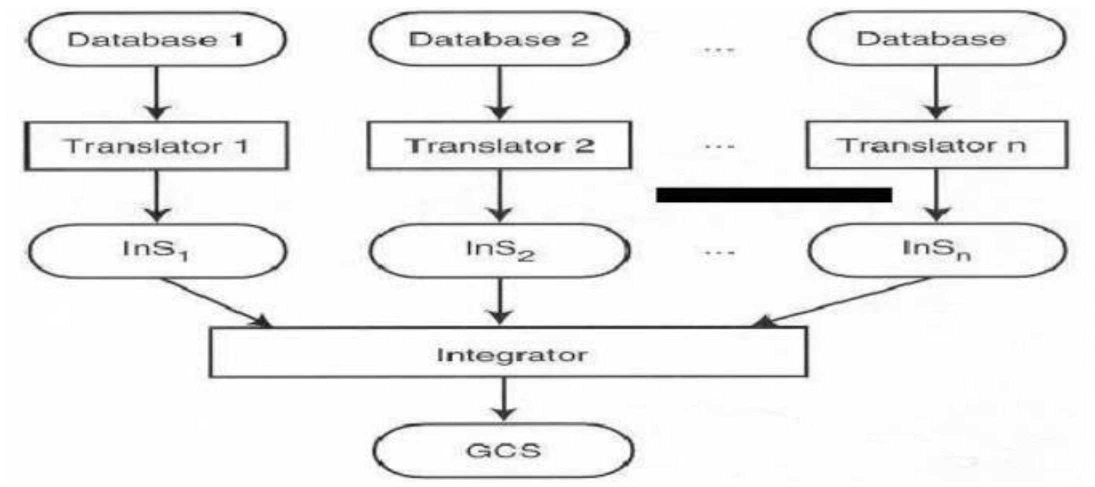


Fig: Bottom-Up Design Process

Distribution Design Issues(Data Replication and fragmentation)

1. Why to fragment?
2. How to fragment?
3. How much to fragment?
4. How to test correctness?
5. How to allocate?
6. Information requirements

Data replication

Data replication involves electronically copying data from one database to another, ensuring consistent information sharing among users. It creates a distributed database where users access task-specific data without disrupting others. Replication eliminates data ambiguity through normalization.

Process:

- Replicates database objects across multiple sites in a distributed system.
- Captures changes locally before forwarding them to remote locations.

- Provides fast local data access and maintains application availability.

Objects and Sites:

- Replication Object: Database element existing on multiple servers.
- Replication Groups: Organize schema objects for specific applications.
- Replication Sites: Include master sites for updates and snapshot sites for read-only copies.

Benefits:

a. Enables geographically distributed applications.
b. Supports load balancing and high availability.

Fragmentation

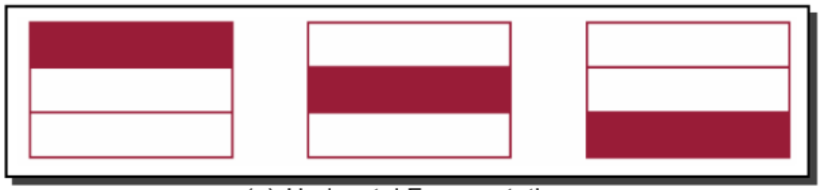
Fragmentation, the process of dividing a relation into smaller fragments, aims to improve various aspects of distributed database systems:

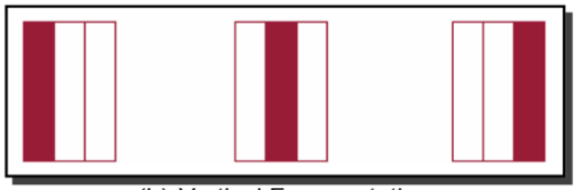
1. Reliability: By distributing data across multiple sites, redundancy can be introduced, enhancing data availability in case of site failures.
2. Performance: Fragments can be placed strategically to minimize data transfer and improve query processing efficiency, leading to better performance.
3. Balanced Storage Capacity and Costs: Fragmentation helps balance the storage capacity and costs among different sites, preventing data overload on any single node.
4. Communication Costs: By placing frequently accessed data locally, communication costs can be reduced as queries often involve less data transfer.

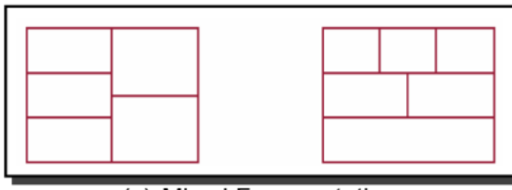
5. Security: Fragmentation can help enforce security measures by placing sensitive data fragments on secure nodes, controlling access to them. Data Replication in Distributed Databases

   Types of fragmentation

   1. Horizontal Fragmentation
   2. Vertical Fragmentation
   3. Hybrid(Mixed) Fragmentation.


(a) Horizontal Fragmentation


(b) Vertical Fragmentation


(c) Mixed Fragmentation

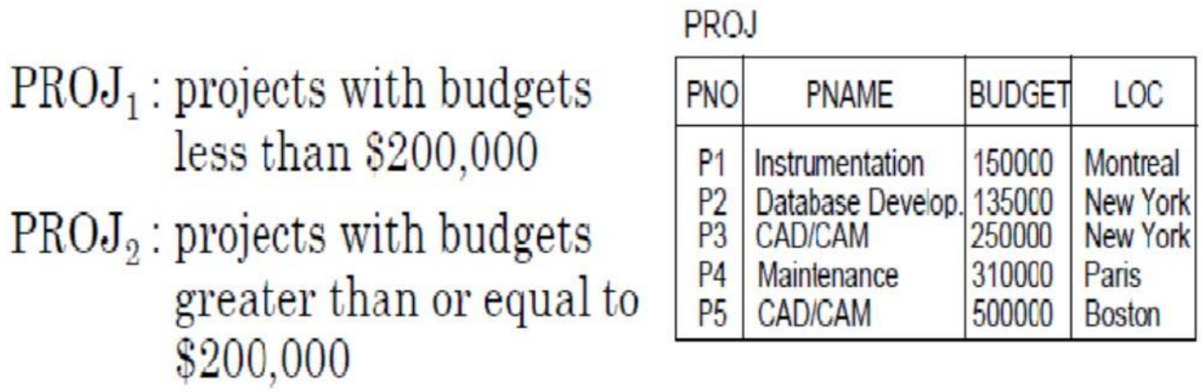
4.

Horizontal Fragmentation:

Involves breaking a table into subsets of rows based on a condition.

Each subset, or fragment, contains rows that satisfy the specified condition.

Example: Dividing an "Employee" table into fragments based on departments, with each fragment containing employees of a specific department.

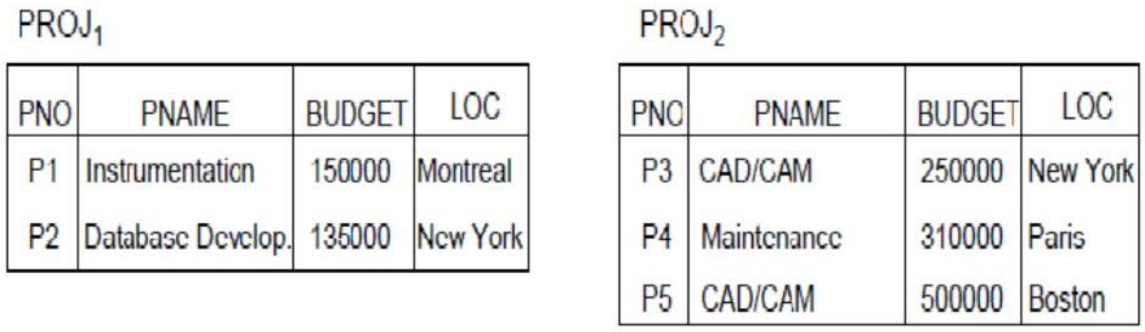


PROJ_1 : projects with budgets less than \$200,000

PROJ_2 : projects with budgets greater than or equal to \$200,000

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston



PROJ_1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

PROJ_2

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

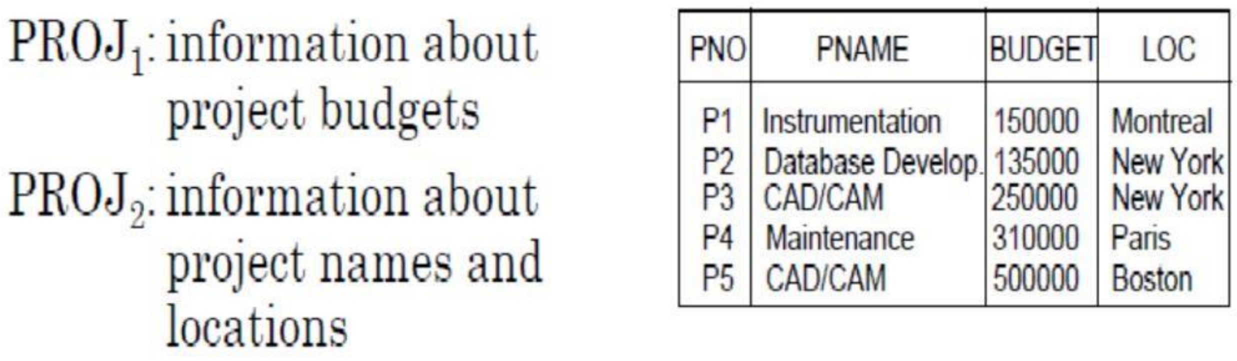
Benefits: Enhanced data distribution, optimized query performance for specific conditions.

Vertical Fragmentation:

Divides a table into subsets of columns (attributes) based on usage patterns.

Each fragment contains a subset of attributes.

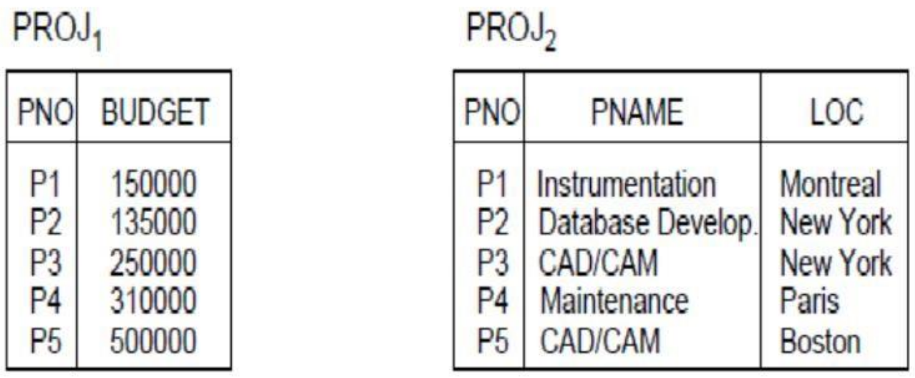
Example: Fragmenting an "Order" table into separate fragments for "Order ID" and "Customer Name" attributes.



$PROJ_1$: information about project budgets

$PROJ_2$: information about project names and locations

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston



$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000
P5	500000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris
P5	CAD/CAM	Boston

Benefits: Reduced storage requirements, minimized I/O operations, improved efficiency for queries involving specific attributes.

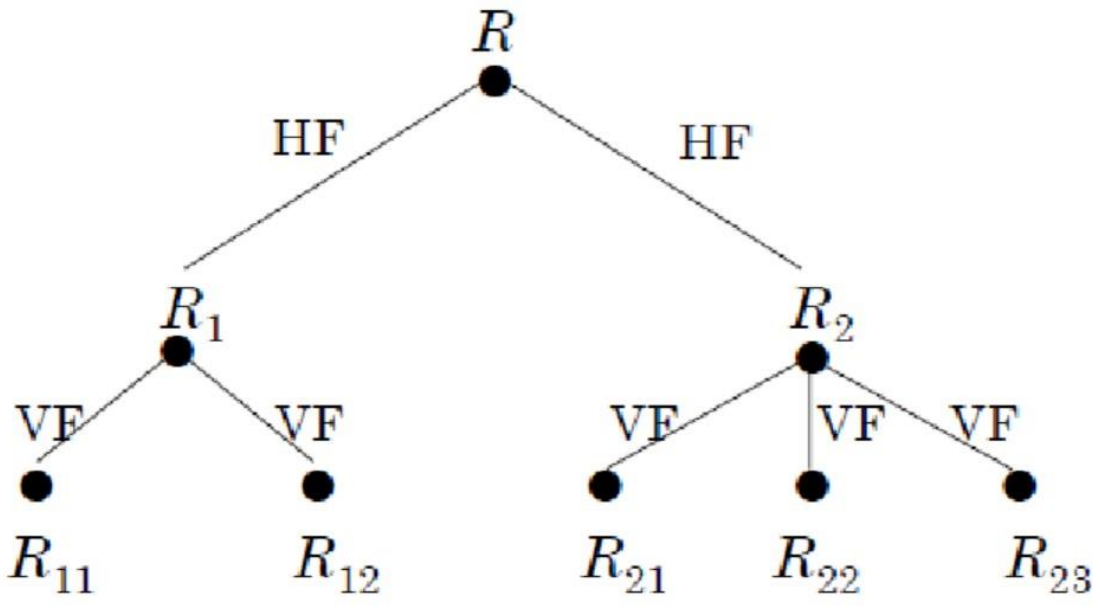
Mixed Fragmentation:

Combines aspects of both horizontal and vertical fragmentation.

Divides a table into fragments based on both rows and columns.

Offers flexibility in optimizing data distribution and query performance.

Example: Fragmenting a "Product" table based on categories (horizontal) and selecting specific attributes within each category (vertical).



Correctness Rules of Fragmentation

Completeness

- Decomposition of relation R into fragments R1, R2, ..., Rn is complete if and only if each data item in R can also be found in some Ri.

Reconstruction

- If relation R is decomposed into fragments R1, R2, ..., Rn, there should exist some relational operator ∇ that reconstructs R from its fragments:
  - R = R1 ∇ ... ∇ Rn
  - Union operation is used to combine horizontal fragments.
  - Join operation is used to combine vertical fragments.

Disjointness

- If relation R is decomposed into fragments R1, R2, ..., Rn, and data item di appears in fragment Rj, then di should not appear in any other fragment Rk, where k ≠ j.
  - Exception: Primary key attribute for vertical fragmentation.
  - Data item, in the context of horizontal fragmentation, refers to a tuple.

- Data item, in the context of vertical fragmentation, refers to an attribute.

.

Allocation in Distributed Databases

Allocation involves strategically distributing database fragments across different sites in a distributed database system. The goal is to optimize data access, resource utilization, and overall system performance.

Key Considerations:

1. Minimal Data Transfer: Fragments should be allocated to minimize the need for data transfers across sites during query execution.
2. Load Balancing: Distributing data evenly across sites prevents resource bottlenecks and ensures efficient utilization.
3. Access Patterns: Fragment allocation should align with the access patterns of applications to enhance query performance.
4. Resource Constraints: Allocation must adhere to site capacities, network bandwidth, and hardware limitations.
5. Data Affinity: Related data should be placed together to reduce joins and improve query efficiency.

Data Directory in Distributed Databases

A data directory is a centralized repository that maintains essential information about the distributed database system, including data locations, metadata, and mappings.

Functions:

1. Data Location: Helps users and applications locate data fragments across different sites.
2. Metadata Management: Stores information about data structure, relationships, and characteristics.
3. Mapping Information: Maintains mappings between logical and physical data locations.
4. Resource Management: Tracks site capacities, processing power, and network bandwidth for efficient resource allocation.
5. Security and Access Control: Stores authentication and authorization details to control data access.
6. Transaction Management: Manages ongoing transactions and their status at various sites.
7. Fault Tolerance: Stores information about backup sites for data recovery in case of failures.

Semantic data control typically includes view management, security control, and

semantic integrity control.

• Informally, these functions must ensure that authorized users perform correct

operations on the database, contributing to the maintenance of database integrity. • In RDBMS semantic data control can be achieved in a uniform way

– views, security constraints, and semantic integrity constraints can be defined as rules

that the system automatically enforces

View management

• Views enable full logical data independence

• Views are virtual relations that are defined as the result of a query on base relations

• Views are typically not materialized

– Can be considered a dynamic window that reflects all relevant updates to the

database

• Views are very useful for ensuring data security in a simple way

– By selecting a subset of the database, views hide some data

– Users cannot see the hidden data

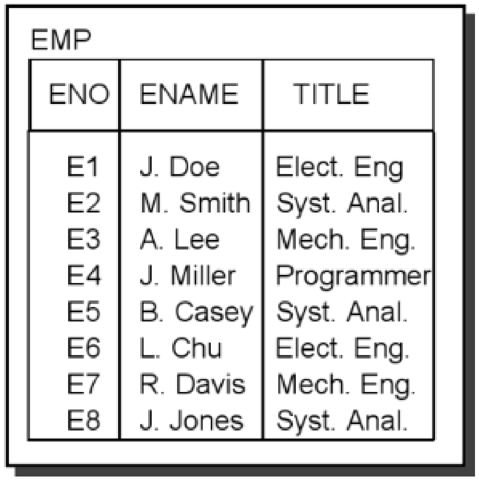
View Management in Centralized Databases

• A view is a relation that is derived from a base relation via a query.

• It can involve selection, projection, aggregate functions, etc.

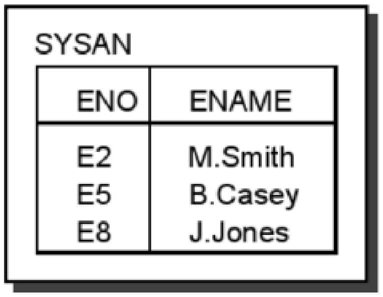
• Example: The view of system analysts derived from relation EMP

```
CREATE VIEW SYSAN(ENO,ENAME) AS

SELECT ENO,ENAME

FROM EMP

WHERE TITLE="Syst. Anal."
```



EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.



SYSAN

ENO	ENAME
E2	M.Smith
E5	B.Casey
E8	J.Jones

Queries expressed on views are translated into queries expressed on base relations

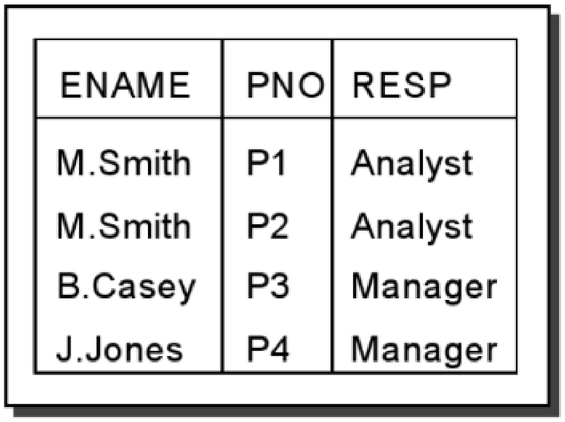
• Example: “Find the names of all the system analysts with their project number and responsibility?”

– Involves the view SYSAN and the relation ASG(ENO,PNO,RESP,DUR)

SELECT ENAME, PNO, RESP

FROM SYSAN, ASG

WHERE SYSN.ENO = ASG.ENO



ENAME	PNO	RESP
M.Smith	P1	Analyst
M.Smith	P2	Analyst
B.Casey	P3	Manager
J.Jones	P4	Manager

is translated into

SELECT ENAME,PNO,RESP

FROM EMP, ASG

WHERE EMP.ENO = ASG.ENO

AND TITLE = "Syst. Anal."

• Automatic query modification is required, i.e., ANDing query qualification with view

Qualification

All views can be queried as base relations, but not all view can be updated as such

– Updates through views can be handled automatically only if they can be propagated

correctly to the base relations

– We classify views as updatable or not-updatable

• Updatable view: The updates to the view can be propagated to the base relations

without ambiguity.

```
CREATE VIEW SYSAN(ENO,ENAME) AS

SELECT ENO,ENAME

FROM EMP

WHERE TITLE="Syst. Anal."
```

– e.g, insertion of tuple (201,Smith) can be mapped into the insertion of a new

employee (201, Smith, "Syst. Anal.")

– If attributes other than TITLE were hidden by the view, they would be assigned the

value null• Non-updatable view: The updates to the view cannot be propagated to the base

relations without ambiguity.

```
CREATE VIEW EG(ENAME,RESP) AS

SELECT ENAME,RESP

FROM EMP, ASG

WHERE EMP.ENO=ASG.ENO
```

– e.g, deletion of (Smith, ”Syst. Anal.”) is ambiguous, i.e., since deletion of “Smith” in

EMP and deletion of “Syst. Anal.” in ASG are both meaningful, but the system cannot

decide.

• Current systems are very restrictive about supportin gupdates through views

– Views can be updated only if they are derived from a single relation by selection and

projection

– However, it is theoretically possible to automatically support updates of a larger class

of views, e.g., joins

View Management in Distributed Databases

Definition of views in DDBMS is similar as in centralized DBMS

– However, a view in a DDBMS may be derived from fragmented relations stored at

different sites

• Views are conceptually the same as the base relations, therefore we store them in the

(possibly) distributed directory/catalogue

– Thus, views might be centralized at one site, partially replicated, fully replicated

– Queries on views are translated into queries on base relations, yielding distributed

queries due to possible fragmentation of data

• Views derived from distributed relations may be costly to evaluate

– Optimizations are important, e.g., snapshots

– A snapshot is a static view

∗ does not reflect the updates to the base relations

∗ managed as temporary relations: the only access path is sequential scan

∗ typically used when selectivity is small (no indices can be used efficiently)

∗ is subject to periodic recalculation

Challenges:

1. Data Distribution: Creating unified views from distributed data requires integrating data from multiple sites.
2. Data Heterogeneity: Different sites might use varying data models, necessitating data transformation.
3. Data Independence: Users should be shielded from underlying changes in data organization.

Data Security in Distributed Databases

Data security protects data against unauthorized acces and has two aspects:

– Data protection

– Authorization control

• **Data protection** prevents unauthorized users from understanding the physical content of

data.

• Well established standards exist

– Data encryption standard

– Public-key encryption schemes

Authorization control must guarantee that only authorized users perform operations they are allowed to perform on the database.

• Three actors are involved in authorization

- users, who trigger the execution of application programms
- operations, which are embedded in applications programs
- database objects, on which the operations are performed

• Authorization control can be viewed as a triple (user, operation type, object) which specifies that the user has the right to perform an operation of operation type on an object.

• Authentication of (groups of) users is typically done by username and password

• Authorization control in (D)DBMS is more complicated as in operating systems

– In a file system: data objects are files

– In a DBMS: Data objects are views, (fragments of) relations, tuples, attributes

Grand and revoke statements are used to authorize triplets (user, operation, data object)

– GRANT <operations> ON <object> TO <users>

– REVOKE <operations> ON <object> TO <users>

• Typically, the creator of objects gets all permissions

– Might even have the permission to GRANT permissions

– This requires a recursive revoke process

• Privileges are stored in the directory/catalogue, conceptually as a matrix

EMP ENAME ASG

Casey UPDATE UPDATE UPDATE

Jones SELECT SELECT SELECT WHERE RESP 6= “Manager”

Smith NONE SELECT NONE

• Different materializations of the matrix are possible (by row, by columns, by element), allowing for different optimizations

– e.g., by row makes the enforcement of authorization efficient, since all rights of a user are in a single tuple

Additional problems of authorization control in a distributed environment stem from the fact that objects and subjects are distributed:

– remote user authentication

– managmenet of distributed authorization rules

– handling of views and of user groups

• Remote user authentication is necessary since any site of a DDBMS may accept programs initiated and authorized at remote sites

• Two solutions are possible:

– (username, password) is replicated at all sites and are communicated between the sites, whenever the relations at remote sites are accessed

∗ beneficial if the users move from a site to a site

– All sites of the DDBMS identify and authenticate themselves similarly as users do

∗ intersite communication is protected by the use of the site password;

∗ (username, password) is authorized by application at the start of the session;

∗ no remote user authentication is required for accessing remote relations once the start site has been authenticated

∗ beneficial if users are static

Semantic Integrity Control

A database is said to be consistent if it satisfies a set of constraints, called semantic integrity constraints
• Maintain a database consistent by enforcing a set of constraints is a difficult problem
• Semantic integrity control evolved from procedural methods (in which the controls were
embedded in application programs) to declarative methods
– avoid data dependency problem, code redundancy, and poor performance of the procedural methods
• Two main types of constraints can be distinguished:
– Structural constraints: basic semantic properties inherent to a data model e.g., unique key constraint in relational model
– Behavioral constraints: regulate application behavior e.g., dependencies (functional, inclusion) in the relational model
• A semantic integrity control system has 2 components:
– Integrity constraint specification
– Integrity constraint enforcement
Integrity constraints specification
– In RDBMS, integrity constraints are defined as assertions, i.e., expression in tuple relational calculus
– Variables are either universally (∀) or existentially ∃) quantified

– Declarative method
– Easy to define constraints
– Can be seen as a query qualification which is either true or false
– Definition of database consistency clear
– 3 types of integrity constraints/assertions are distinguished:
∗ predefined
∗ precompiled
∗ general constraints

• In the following examples we use the following relations:
EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET)
ASG(ENO, PNO, RESP, DUR)

Predefined constraints are based on simple keywords and specify the more common contraints of the relational model
• Not-null attribute:
– e.g., Employee number in EMP cannot be null
ENO NOT NULL IN EMP
• Unique key:
– e.g., the pair (ENO,PNO) is the unique key in ASG
(ENO, PNO) UNIQUE IN ASG

• Foreign key:
– e.g., PNO in ASG is a foreign key matching the primary key PNO in PROJ
PNO IN ASG REFERENCES PNO IN PROJ
• Functional dependency:
– e.g., employee number functionally determines the employee name
ENO IN EMP DETERMINES ENAME
Precompiled constraints express preconditions that must be satisfied by all tuples in A relation for a given update type
• General form:
CHECK ON <relation> [WHEN <update type>] <qualification>
• Domain constraint, e.g., constrain the budget:
CHECK ON PROJ(BUDGET>500000 AND BUDGET≤1000000)
• Domain constraint on deletion, e.g., only tuples with budget 0 can be deleted:
CHECK ON PROJ WHEN DELETE (BUDGET = 0)
• Transition constraint, e.g., a budget can only increase:
CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET AND

NEW.PNO = OLD.PNO)

– OLD and NEW are implicitly defined variables to identify the tuples that are subject to Update

General constraints may involve more than one relation

• General form:

CHECK ON <variable>:<relation> (<qualification>)

• Functional dependency:

```
CHECK ON e1:EMP, e2:EMP
(e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)
```

• Constraint with aggregate function:

e.g., The total duration for all employees in the CAD project is less than 100

```
CHECK ON g:ASG, j:PROJ
( SUM(g.DUR WHERE g.PNO=j.PNO) < 100
IF j.PNAME="CAD/CAM" )
```

Query Processing

Query processing is a complex task in distributed databases, involving the optimization and execution of user queries across multiple distributed sites. However, this process presents several challenges and intricacies due to the distributed nature of the data. Query Processing Problem

1. Data Distribution: Data is distributed across multiple sites, leading to the need for data retrieval from different locations during query execution.
2. Network Latency: Network communication between sites introduces latency, affecting query performance and response times.
3. Query Optimization: Optimizing queries for efficient execution across distributed sites requires considering factors like data distribution, access methods, and join strategies.
4. Data Localization: Identifying the most appropriate site to execute a query can impact performance significantly.
5. Data Heterogeneity: Different sites might use varying data models, leading to the need for data translation and transformation.
6. Load Balancing: Distributing query processing evenly across sites ensures optimal resource utilization.
7. Data Skew: Uneven data distribution can lead to load imbalances and slow query processing.

Objectives of Query processing

The process involves several objectives that collectively contribute to efficient and accurate data retrieval. Here are the key objectives of query processing:

1. Efficiency: Query processing aims to execute queries in the most efficient manner possible to minimize response times and optimize resource usage. This is especially important in large databases with complex queries.
2. Optimization: The process seeks to optimize query execution plans by considering various factors such as access paths, join strategies, indexing methods, and data distribution. The goal is to select the most cost-effective plan.

3. Accuracy: Query processing must ensure that the retrieved data is accurate and consistent with the data in the database. This involves avoiding errors during data retrieval and manipulation.
4. Concurrency Control: In multi-user environments, query processing needs to handle concurrent access to the database. This includes managing locks and ensuring data integrity when multiple users access the same data simultaneously.
5. Data Localization: For distributed databases, query processing aims to minimize data movement between sites by selecting the most appropriate sites for executing parts of the query.
6. Resource Optimization: The process strives to efficiently allocate computational resources such as CPU time, memory, and I/O operations to ensure optimal query execution.
7. Query Transformation: Query processing may involve transforming user queries into equivalent forms that can be processed more efficiently or that adhere to certain constraints.
8. Index Utilization: Utilizing appropriate indexes helps speed up data retrieval. Query processing aims to identify and utilize suitable indexes to enhance query performance.
9. Predicate Pushdown: Pushing down predicates (filter conditions) to lower levels of query processing (e.g., closer to data sources) can reduce the amount of data processed and improve efficiency.
10. Materialized Views: Query processing can take advantage of precomputed materialized views to quickly retrieve results for complex queries, reducing the need for extensive computation.
11. Cost Estimation: Query processing estimates the cost of various execution plans to determine which plan is likely to be the most efficient. This involves considering factors like I/O costs, processing costs, and communication costs in distributed environments.
12. Query Caching: Reusing previously executed query results stored in cache can speed up query processing, especially for recurring queries.

Complexity of Relational Algebra (RA) Operations:

Relational Algebra consists of fundamental operations to manipulate and retrieve data in a relational database. The complexity of these operations depends on factors like data distribution, indexing, and query structure:

1. Selection (σ): Filtering rows based on a condition. Complexity depends on the selectivity of the condition and indexing.
2. Projection (π): Retrieving specific columns. Complexity is generally linear with the number of rows.
3. Union (∪): Combining two sets of rows. Complexity depends on duplicate handling.
4. Intersection (∩): Finding common rows between two sets. Complexity depends on duplicate handling.
5. Difference (−): Finding rows in one set but not in another. Complexity depends on duplicate handling.
6. Cartesian Product (×): Combining all rows from two tables. Complexity is quadratic with the number of rows.
7. Join (⋈): Combining rows from two tables based on a common attribute. Complexity depends on the join type (nested loop, hash join, merge join) and indexing.
8. Division (÷): Finding tuples from one table that match all values in another table. Division is generally complex and can involve multiple steps.

Characteristics of Query Processors

a. Languages:
    - Input language can be relational calculus or algebra.
    - Distributed output language includes relational algebra with communication primitives for mapping.

b. Types of Optimization:
    - Query optimization aims to minimize cost by choosing optimal execution strategies.

   c. Exhaustive search with heuristics is common, focusing on reducing intermediate relation sizes.

b. Optimization Timing:
   a. Optimization can be static (before execution) or dynamic (during execution).
   b. Dynamic optimization considers actual relation sizes, minimizing bad choices but can be expensive.

c. Statistics:
   a. Effective query optimization relies on accurate database statistics.
   b. Dynamic optimization needs statistics to determine operation order, while static optimization estimates relation sizes.
   c. Periodical updates enhance statistical accuracy.

d. Decision Sites:
   a. Centralized decision approach uses a single site for strategy generation.
   b. Distributed approach involves multiple sites for strategy elaboration.
   c. Hybrid approach combines centralized and local decisions.

e. Exploitation of Network Topology:
   a. Distributed query processors utilize network topology to minimize communication costs.
   b. With wide area networks, focus on data communication cost.
   c. With local area networks, parallel execution may increase even if communication costs rise.

f. Exploitation of Replicated Fragments:
   a. Query processors use replication for reliability.
   b. Exploitation is done statically or dynamically to enhance query efficiency.

g. Use of Semi-Joins:
    a. Semi-join operation reduces exchanged data size, lowering communication costs.

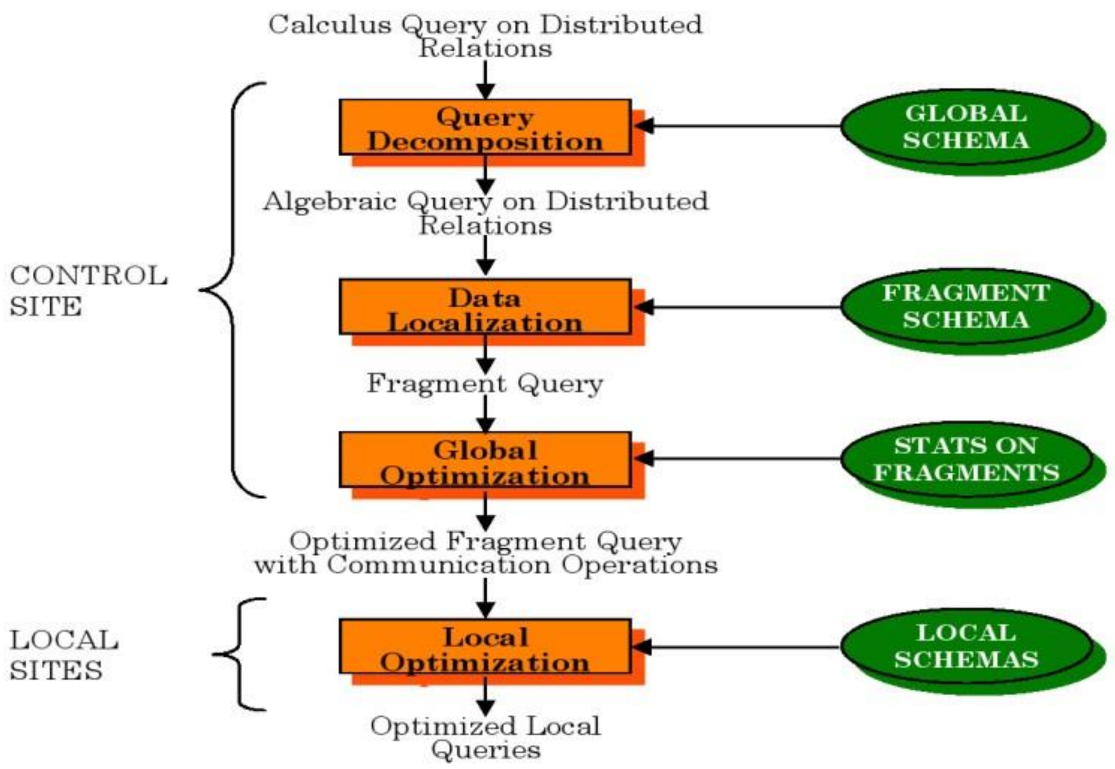


Fig. Generic layering scheme for distributed query processing

Query decomposition: Mapping of calculus query (SQL) to algebra operations (select, project, join, rename)

- Both input and output queries refer to global relations, without knowledge of the distribution of data.
- The output query is semantically correct and good in the sense that redundant work is avoided.
- Query decomposistion consists of 4 steps:

Normalization:

a. Lexical and Syntactic Analysis: Check the validity of the query, attributes, and syntax.
b. Put into Normal Form: Convert the query into a normalized form to facilitate further processing.

Analysis:

Determine the correctness and suitability of the query for further processing.

In some cases, identify and reject "incorrect" queries, typically applicable for a subset of relational calculus.

Elimination of Redundancy:

Identify and eliminate redundant predicates or conditions in the query.

Redundant predicates can be removed without changing the semantics of the query.

Rewriting:

Transform the query into a form suitable for execution using relational algebra operations.

Optimize the query by rearranging and optimizing operations to improve efficiency.

Example: Consider the following query: *Find the names of employees who have been working on project P1 for 12 or 24 months?*

• The query in SQL:

SELECT ENAME
FROM EMP, ASG
WHERE EMP.ENO = ASG.ENO **AND**
ASG.PNO = ‘‘P1’’ **AND**
DUR = 12 **OR** DUR = 24

• The qualification in conjunctive normal form:

EMP.ENO = ASG.ENO ^ ASG.PNO = ”P1” ^ (DUR = 12 _ DUR = 24)

• The qualification in disjunctive normal form:

(EMP.ENO = ASG.ENO ^ ASG.PNO = ”P1” ^ DUR = 12) _
(EMP.ENO = ASG.ENO ^ ASG.PNO = ”P1” ^ DUR = 24)

Query Decomposition – Analysis

Analysis: Identify and reject type incorrect or semantically incorrect queries

• Type incorrect

– Checks whether the attributes and relation names of a query are defined in the global schema

– Checks whether the operations on attributes do not conflict with the types of the attributes, e.g., a comparison > operation with an attribute of type string

• Semantically incorrect

– Checks whether the components contribute in any way to the generation of the result

– Only a subset of relational calculus queries can be tested for correctness, i.e., those that do not contain disjunction and negation

– Typical data structures used to detect the semantically incorrect queries are:

————————————— Connection graph (query graph)

————————————— Join graph

Example: Consider a query:

```
SELECT ENAME,RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND ASG.PNO = PROJ.PNO
AND PNAME = "CAD/CAM"
AND DUR  36
AND TITLE = "Programmer"
```

• Query/connection graph

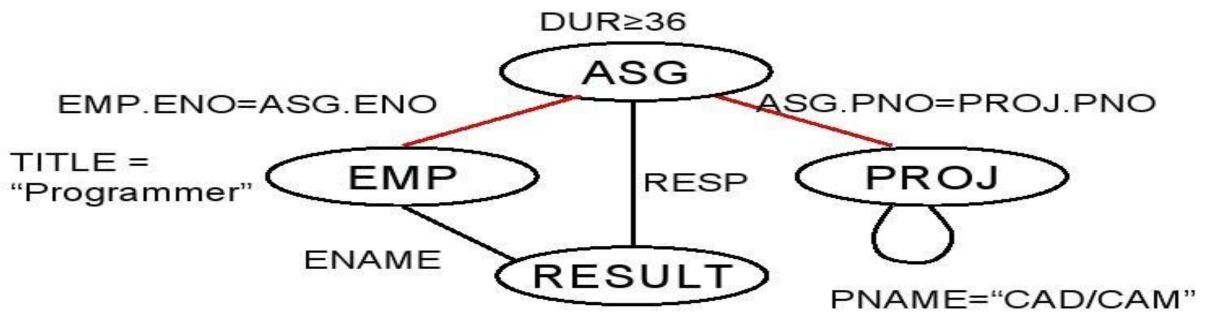
– Nodes represent operand or result relation

– Edge represents a join if both connected nodes represent an operand relation, otherwise it is a projection • Join graph

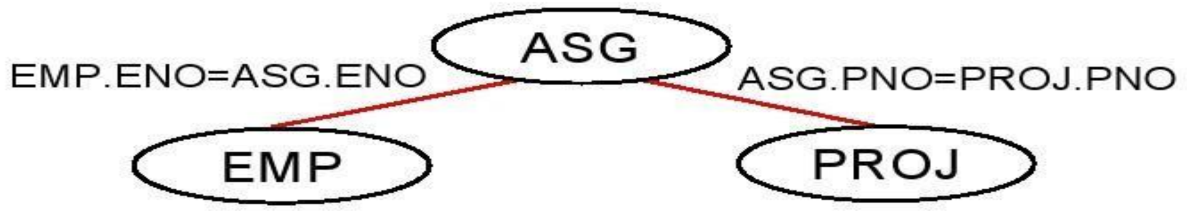
– a subgraph of the query graph that considers only the joins

• Since the query graph is connected, the query is semantically correct

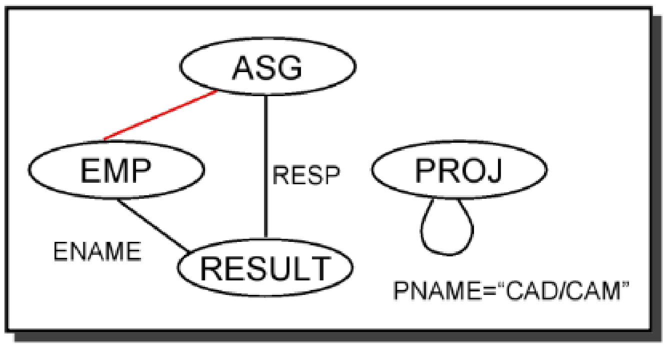
Query graph



Join graph



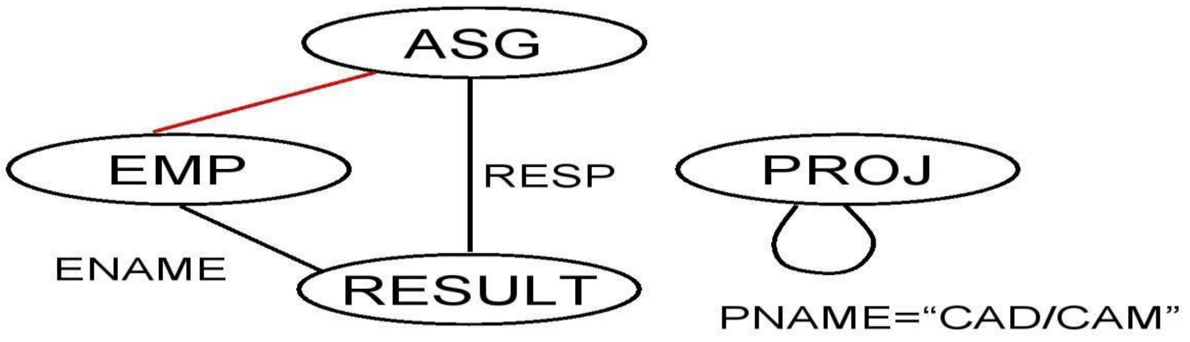
Example: Consider the following query and its query graph:

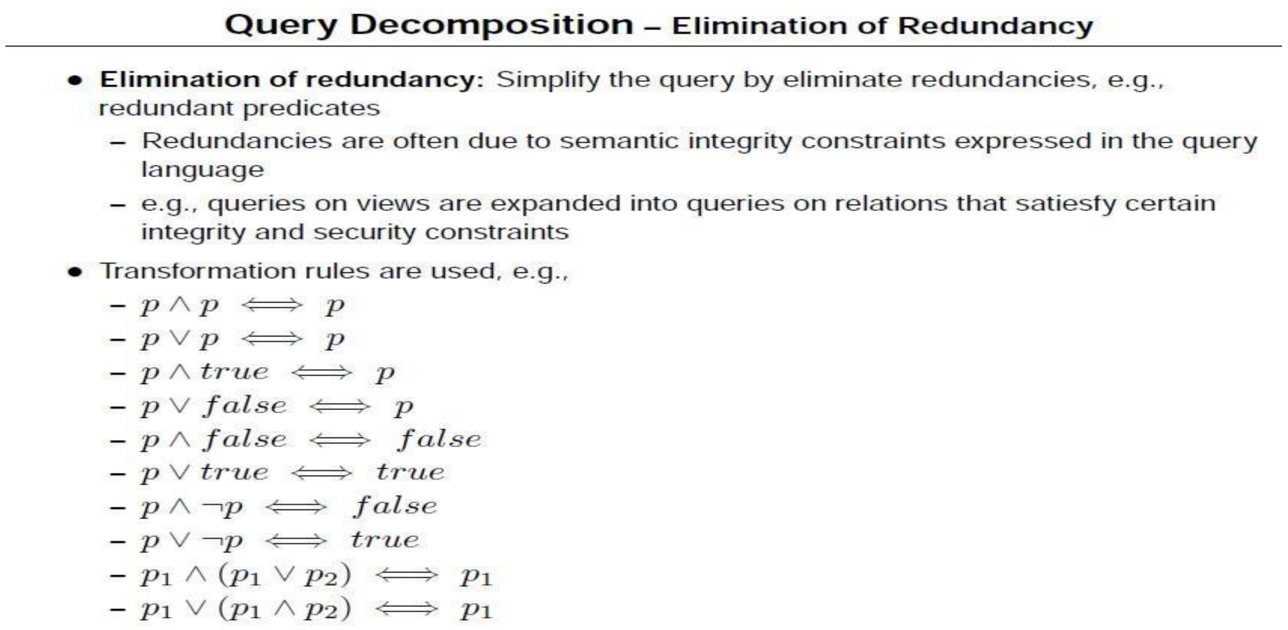


SELECT ENAME,RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND PNAME = "CAD/CAM"
AND DUR _ 36
AND TITLE = "Programmer"

• Since the graph **is not connected**, the query is semantically incorrect.
• 3 possible solutions:
– Reject the query
– Assume an implicit Cartesian Product between ASG and PROJ

– Infer from the schema the

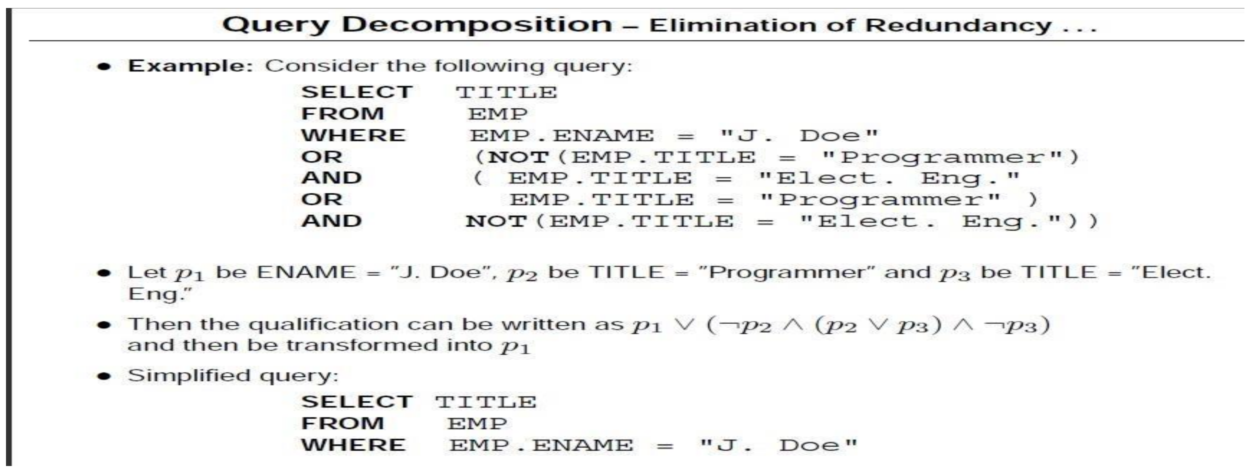




Query Decomposition – Elimination of Redundancy

- **Elimination of redundancy:** Simplify the query by eliminate redundancies, e.g., redundant predicates
  - Redundancies are often due to semantic integrity constraints expressed in the query language
  - e.g., queries on views are expanded into queries on relations that satiesfy certain integrity and security constraints
- Transformation rules are used, e.g.,
  - $p \wedge p \iff p$
  - $p \vee p \iff p$
  - $p \wedge true \iff p$
  - $p \vee false \iff p$
  - $p \wedge false \iff false$
  - $p \vee true \iff true$
  - $p \wedge \neg p \iff false$
  - $p \vee \neg p \iff true$
  - $p_1 \wedge (p_1 \vee p_2) \iff p_1$
  - $p_1 \vee (p_1 \wedge p_2) \iff p_1$

missing join predicate ASG.PNO = PROJ.PNO



Query Decomposition – Elimination of Redundancy . . .

- **Example:** Consider the following query:

```
SELECT   TITLE
FROM      EMP
WHERE     EMP.ENAME = "J. Doe"
OR        (NOT(EMP.TITLE = "Programmer")
AND       ( EMP.TITLE = "Elect. Eng."
OR          EMP.TITLE = "Programmer" )
AND       NOT(EMP.TITLE = "Elect. Eng."))
```

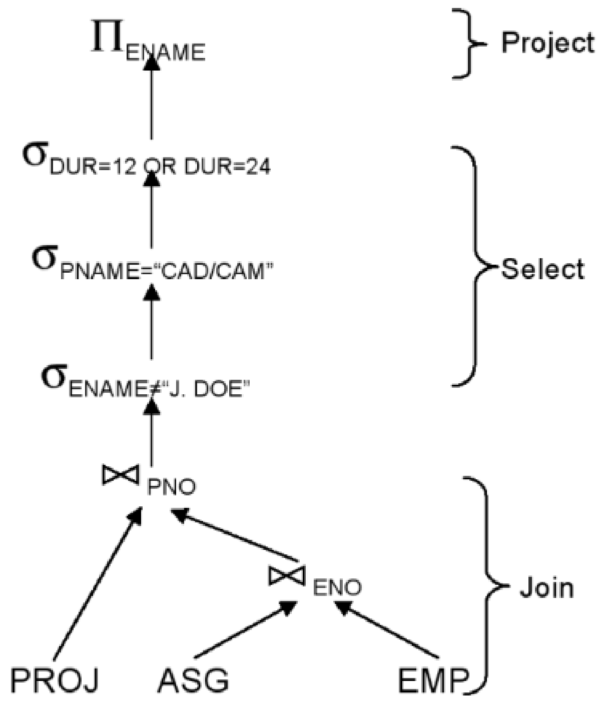
- Let p_1 be ENAME = "J. Doe", p_2 be TITLE = "Programmer" and p_3 be TITLE = "Elect. Eng."
- Then the qualification can be written as $p_1 \vee (\neg p_2 \wedge (p_2 \vee p_3) \wedge \neg p_3)$ and then be transformed into p_1
- Simplified query:

```
SELECT TITLE
FROM    EMP
WHERE   EMP.ENAME = "J. Doe"
```

Query Decomposition – Rewriting

Rewriting: Convert relational calculus query to relational algebra query and find an **efficient** expression.

• **Example:** Find the names of employees other than J. Doe who worked on the CAD/CAM project for either 1 or 2 years.



•

• **SELECT** ENAME
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND ASG.PNO = PROJ.PNO
AND ENAME 6= "J. Doe"
AND PNAME = "CAD/CAM"
AND (DUR = 12 **OR** DUR = 24)

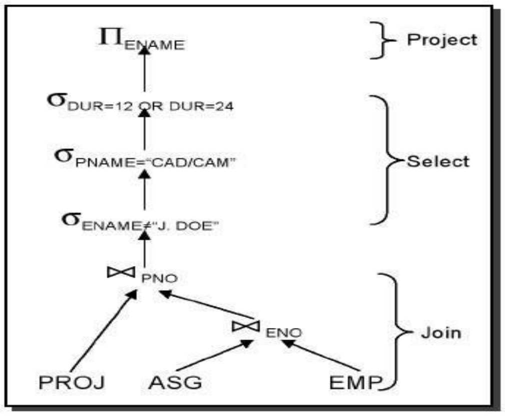
• A **query tree** represents the RA-expression – Relations are leaves (FROM clause)

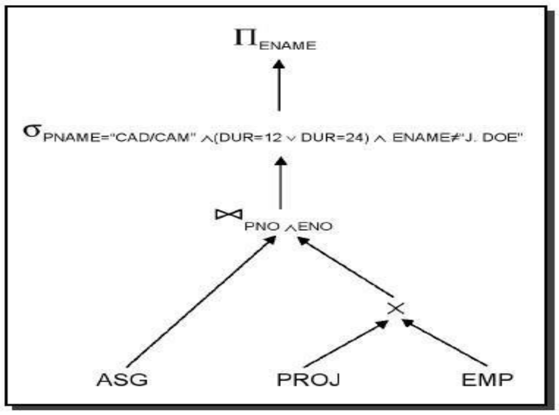
– Result attributes are root (SELECT clause) – Intermediate leaves should give a result from the leaves to the root

- By applying **transformation rules**, many different trees/expressions may be found that are **equivalent** to the original tree/expression, but might be more efficient.
- In the following we assume relations $R(A_1, \ldots, A_n)$, $S(B_1, \ldots, B_n)$, and T which is union-compatible to R.
- **Commutativity** of binary operations
  - $R \times S = S \times R$
  - $R \bowtie S = S \bowtie R$
  - $R \cup S = S \cup R$
- **Associativity** of binary operations
  - $(R \times S) \times T = R \times (S \times T)$
  - $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- **Idempotence** of unary operations
  - $\Pi_A(\Pi_A(R)) = \Pi_A(R)$
  - $\sigma_{p1(A1)}(\sigma_{p2(A2)}(R)) = \sigma_{p1(A1) \wedge p2(A2)}(R)$

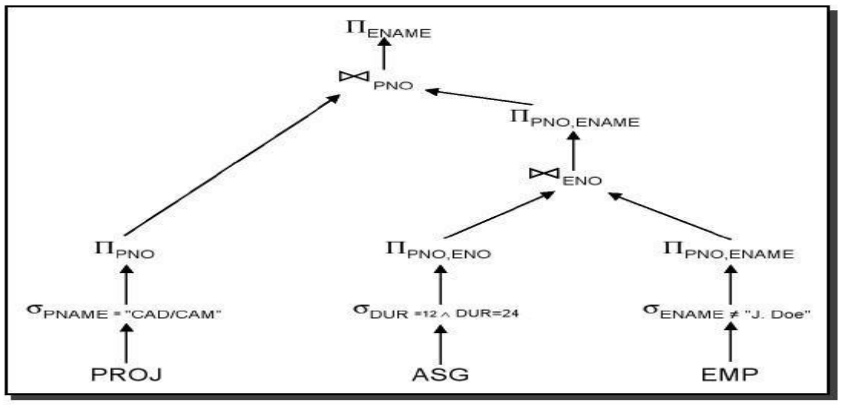
- **Commuting selection** with binary operations
  - $\sigma_{p(A)}(R \times S) \iff \sigma_{p(A)}(R) \times S$
  - $\sigma_{p(A_1)}(R \bowtie_{p(A_2,B_2)} S) \iff \sigma_{p(A_1)}(R) \bowtie_{p(A_2,B_2)} S$
  - $\sigma_{p(A)}(R \cup T) \iff \sigma_{p(A)}(R) \cup \sigma_{p(A)}(T)$
    - $*$ (A belongs to R and T)
- **Commuting projection** with binary operations (assume $C = A' \cup B'$, $A' \subseteq A, B' \subseteq B$)
  - $\Pi_C(R \times S) \iff \Pi_{A'}(R) \times \Pi_{B'}(S)$
  - $\Pi_C(R \bowtie_{p(A',B')} S) \iff \Pi_{A'}(R) \bowtie_{p(A',B')} \Pi_{B'}(S)$
  - $\Pi_C(R \cup S) \iff \Pi_C(R) \cup \Pi_C(S)$

- **Example:** Two equivalent query trees for the previous example
  - Recall the schemas: EMP(ENO, ENAME, TITLE)
    PROJ(PNO, PNAME, BUDGET)
    ASG(ENO, PNO, RESP, DUR)





- **Example (contd.):** Another equivalent query tree, which allows a more efficient query evaluation, since the most selective operations are applied first.



Data Localization

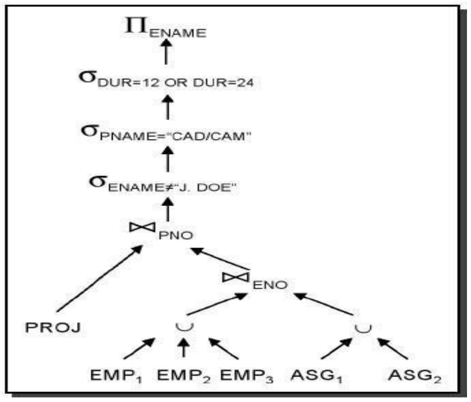
– **Input:** Algebraic query on global conceptual schema

– **Purpose:** Apply data distribution information to the algebra operations and determine which fragments are involved

_ Substitute global query with queries on fragments

_ Optimize the global query

- **Example:**
  - Assume EMP is horizontally fragmented into EMP1, EMP2, EMP3 as follows:
    - $EMP1 = \sigma_{ENO\leq"E3"}(EMP)$
    - $EMP2 = \sigma_{"E3"<ENO\leq"E6"}(EMP)$
    - $EMP3 = \sigma_{ENO>"E6"}(EMP)$
  - ASG fragmented into ASG1 and ASG2 as follows:
    - $ASG1 = \sigma_{ENO\leq"E3"}(ASG)$
    - $ASG2 = \sigma_{ENO>"E3"}(ASG)$
- Simple approach: Replace in all queries
  - EMP by (EMP1∪EMP2∪ EMP3)
  - ASG by (ASG1∪ASG2)
  - Result is also called **generic query**

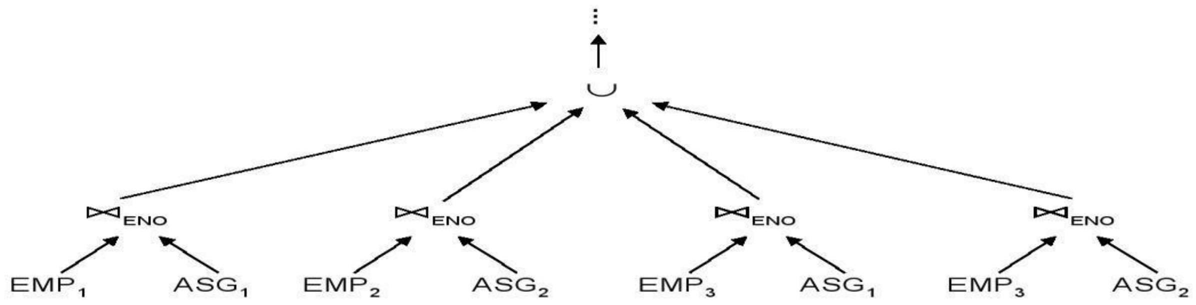


Π_{ENAME}
$\sigma_{DUR=12 \text{ OR } DUR=24}$
$\sigma_{PNAME="CAD/CAM"}$
$\sigma_{ENAME\neq"J. DOE"}$
$\bowtie_{PNO}$
$\bowtie_{ENO}$
PROJ
$\cup$
$\cup$
EMP_1 EMP_2 EMP_3 ASG_1 ASG_2

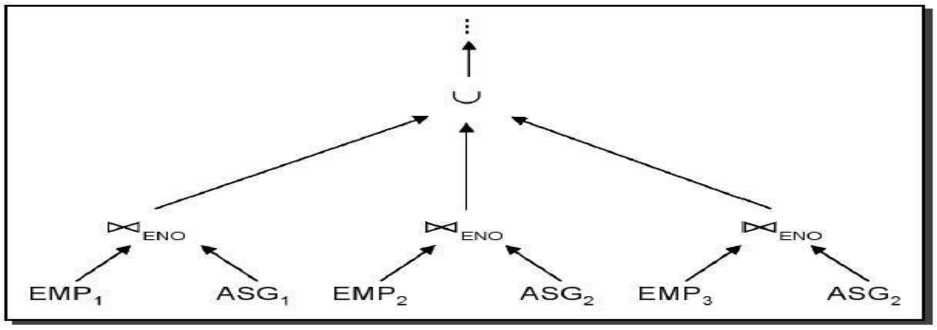
- In general, the **generic query is inefficient** since important restructurings and simplifications can be done.

Example (contd.): Parallelsim in the evaluation is often possible

– Depending on the horizontal fragmentation, the fragments can be joined in parallel followed by the union of the intermediate results.



- **Example (contd.)**: Unnecessary work can be eliminated
  - e.g., $EMP_3 \bowtie ASG_1$ gives an empty result
    - $EMP3 = \sigma_{ENO > "E6"}(EMP)$
    - $ASG1 = \sigma_{ENO \leq "E3"}(ASG)$



Data Localizations Issues

Various more advanced reduction techniques are possible to generate simpler and optimized queries. • Reduction of horizontal fragmentation (HF) – Reduction with selection

– Reduction with join

• Reduction of vertical fragmentation (VF)

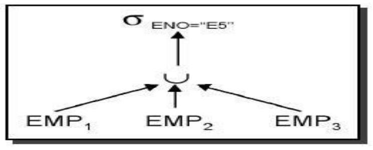
– Find empty relations

Data Localizations Issues – Reduction of HF

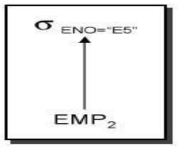
- **Reduction with selection for HF**
  - Consider relation R with horizontal fragmentation $F = \{R_1, R_2, \ldots, R_k\}$, where $R_i = \sigma_{p_i}(R)$
  - **Rule1:** Selections on fragments, $\sigma_{p_j}(R_i)$, that have a qualification contradicting the qualification of the fragmentation generate empty relations, i.e.,

$$\sigma_{p_j}(R_i) = \emptyset \iff \forall x \in R(p_i(x) \wedge p_j(x) = false)$$

  - Can be applied if fragmentation predicate is inconsistent with the query selection predicate.
- **Example:** Consider the query: **SELECT** * **FROM** EMP **WHERE** ENO="E5"



After commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of EMP_1 and EMP_3.

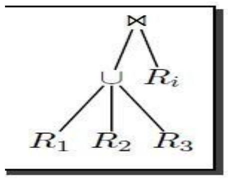


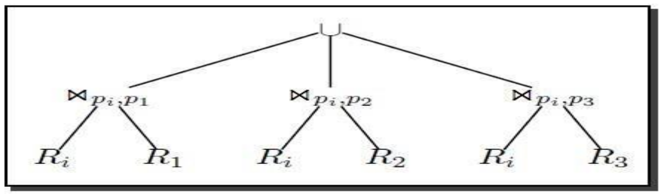
- **Reduction with join for HF**
  - Joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attributes.
  - Distribute join over union

$$(R_1 \cup R_2) \bowtie S \iff (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

  - **Rule 2**: Useless joins of fragments, $R_i = \sigma_{p_i}(R)$ and $R_j = \sigma_{p_j}(R)$, can be determined when the qualifications of the joined fragments are contradicting, i.e.,

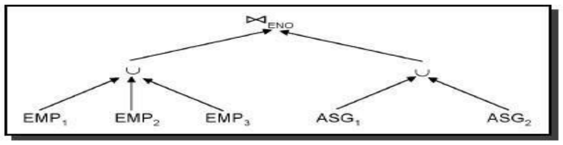
$$R_i \bowtie R_j = \emptyset \iff \forall x \in R_i, \forall y \in R_j (p_i(x) \wedge p_j(y) = false)$$



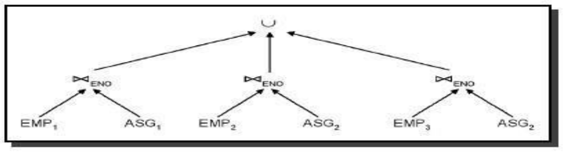


Data Localizations Issues – Reduction for HF . . .

- **Example:** Consider the following query and fragmentation:
  - Query: **SELECT** * **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - Horizontal fragmentation:
    - $EMP1 = \sigma_{ENO \leq "E3"}(EMP)$
    - $EMP2 = \sigma_{"E3" < ENO \leq "E6"}(EMP)$
    - $EMP3 = \sigma_{ENO > "E6"}(EMP)$
    - $ASG1 = \sigma_{ENO \leq "E3"}(ASG)$
    - $ASG2 = \sigma_{ENO > "E3"}(ASG)$
  - Generic query

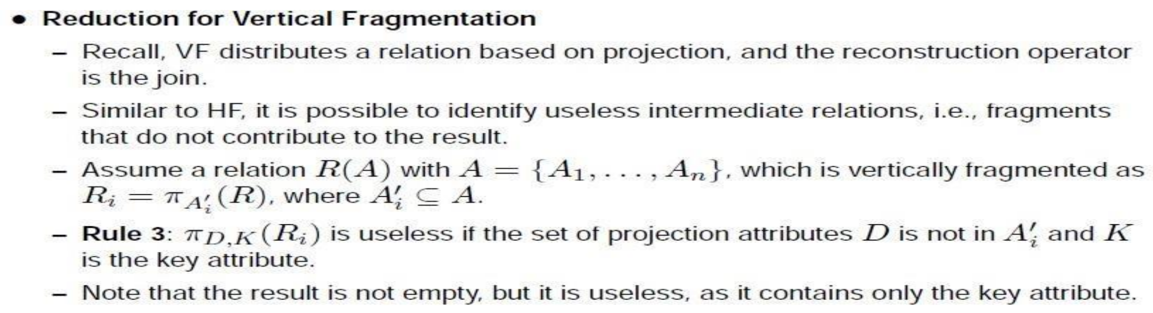


  - The query reduced by distributing joins over unions and applying rule 2 can be implemented as a union of three partial joins that can be done in parallel.

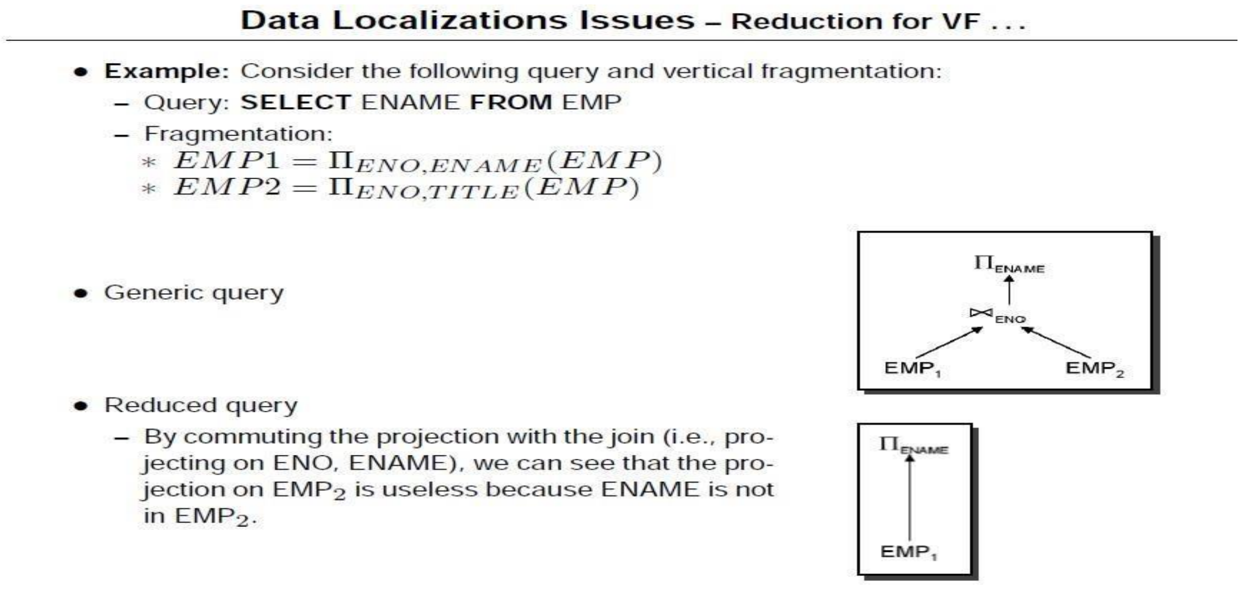


- **Reduction with join for derived HF**
  - The horizontal fragmentation of one relation is **derived** from the horizontal fragmentation of another relation by using semijoins.
- If the fragmentation is not on the same predicate as the join (as in the previous example), derived horizontal fragmentation can be applied in order to make efficient join processing possible.
- **Example:** Assume the following query and fragmentation of the EMP relation:
  - Query: **SELECT** * **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - Fragmentation (**not** on the join attribute):
    - EMP1 = $\sigma_{\text{TITLE="Prgrammer"}}$(EMP)
    - EMP2 = $\sigma_{\text{TITLE}\neq\text{"Prgrammer"}}$(EMP)
  - To achieve efficient joins ASG can be fragmented as follows:
    - ASG1= ASG$\rtimes_{ENO}$EMP1
    - ASG2= ASG$\rtimes_{ENO}$EMP2
  - The fragmentation of ASG is derived from the fragmentation of EMP
  - Queries on derived fragments can be reduced, e.g., $ASG_1 \bowtie EMP_2 = \emptyset$

Data Localizations Issues – Reduction for VF

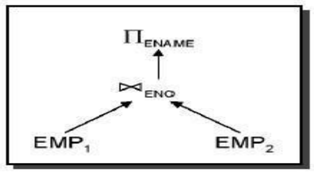


- **Reduction for Vertical Fragmentation**
  - Recall, VF distributes a relation based on projection, and the reconstruction operator is the join.
  - Similar to HF, it is possible to identify useless intermediate relations, i.e., fragments that do not contribute to the result.
  - Assume a relation $R(A)$ with $A = \{A_1, \ldots, A_n\}$, which is vertically fragmented as $R_i = \pi_{A'_i}(R)$, where $A'_i \subseteq A$.
  - **Rule 3**: $\pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A'_i and K is the key attribute.
  - Note that the result is not empty, but it is useless, as it contains only the key attribute.

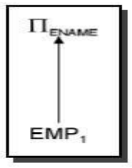


Data Localizations Issues – Reduction for VF ...

- **Example:** Consider the following query and vertical fragmentation:
  - Query: **SELECT** ENAME **FROM** EMP
  - Fragmentation:
    - $EMP1 = \Pi_{ENO,ENAME}(EMP)$
    - $EMP2 = \Pi_{ENO,TITLE}(EMP)$
- Generic query



- Reduced query
  - By commuting the projection with the join (i.e., projecting on ENO, ENAME), we can see that the projection on EMP_2 is useless because ENAME is not in EMP_2.



Unit 4: Distributed Concurrency Control

Serializability Theory:

Serializability is a concept in database management systems that ensures the correctness of concurrent transactions. It guarantees that the execution of concurrent transactions is equivalent to some serial execution of those transactions.

Serializability ensures the following properties:

- Conflict Serializable: Ensures that the order of execution of conflicting operations in concurrent transactions produces the same result as some serial execution.
- View Serializable: Guarantees that the interleaved execution of transactions maintains the same set of read-write relationships as some serial execution.

Taxonomy of Concurrency Control Algorithms:

Concurrency control algorithms are techniques used in database management systems (DBMS) to ensure the correct execution of transactions in a multi-user environment. These algorithms can be categorized into two main classes based on their approach to handling concurrent access: Pessimistic and Optimistic methods.

Pessimistic Concurrency Control:

Pessimistic methods operate under the assumption that conflicts between transactions are likely, and they take measures to prevent these conflicts from happening during the early stages of transaction execution.

1. Two-Phase Locking (2PL):

- In this method, transactions request and acquire locks on data items before accessing them.
- The locks ensure that conflicting operations from different transactions do not occur concurrently.
- Variations of 2PL include centralized 2PL (locks managed by a central entity), primary copy 2PL (locks managed at primary copy locations), and distributed 2PL (locks managed in a distributed manner).

2. Timestamp Ordering (TO):

- Transactions are assigned timestamps that determine their order of execution.
- Older transactions are given priority, ensuring a serialized execution.
- Basic TO, Multiversion TO (maintaining multiple versions of data for different transactions), and Conservative TO (more restrictive timestamp assignment) are different variations of this method.

3. Hybrid Algorithms:

- These algorithms combine aspects of both pessimistic and optimistic methods to achieve a balance.

- They may utilize locks or timestamps, depending on the situation, to provide efficient concurrency control.

Optimistic Concurrency Control:

Optimistic methods work under the assumption that conflicts are less likely to occur, allowing transactions to proceed without immediate locking. Conflicts are detected later, and corrective actions are taken if necessary.

1. Locking-Based Optimistic Concurrency Control:

- Transactions are allowed to proceed without acquiring locks.
- Conflicts are detected during the validation phase before committing.
- If conflicts are found, transactions may be rolled back and re-executed.

2. Timestamp Ordering-Based Optimistic Concurrency Control:

- Transactions execute without locks, and timestamps determine the order.
- Conflicts are identified during the validation phase at commit time.
- Transactions may be rolled back and re-executed if conflicts are detected.

Lock-Based Concurrency Control Algorithms:

Lock-based concurrency control is a widely used method to manage concurrent access to data in a database system. It involves granting and releasing locks on data items to ensure that transactions can be executed safely without causing conflicts. These locks prevent concurrent transactions from accessing the same data in ways that might lead to inconsistent or incorrect results.

Basic Concepts of Locking:

1. Shared Lock (S-lock): Also known as a read lock, it allows a transaction to read the data item but not modify it. Multiple transactions can hold shared locks on the same data item simultaneously.

2. Exclusive Lock (X-lock): Also known as a write lock, it allows a transaction to both read and modify the data item. Only one transaction can hold an exclusive lock on a data item at a time.

Lock-Based Concurrency Control Algorithms:

Lock-Based Concurrency Control:

Lock-based concurrency control is a method used in database systems to manage the concurrent execution of transactions. It employs locks to control access to data items, ensuring that transactions maintain data integrity and consistency when executing simultaneously.

Types of Lock-Based Concurrency Control Methods:

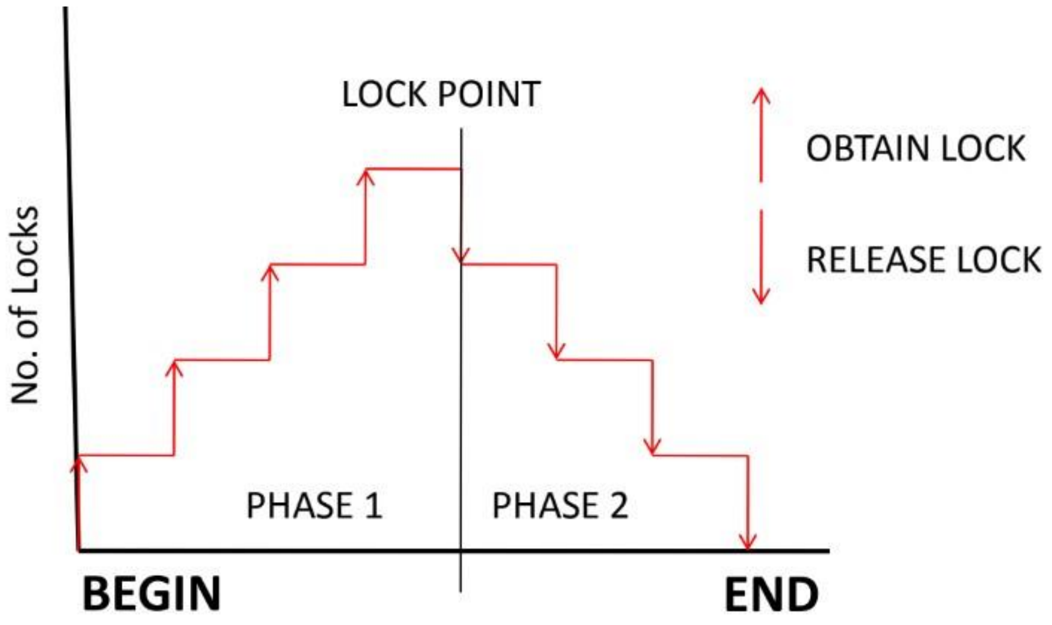
1. Two-Phase Locking Algorithm (2PL):

- Transactions are divided into two phases: the growing phase and the shrinking phase.

- During the growing phase, transactions can request and acquire locks but cannot release them.

- During the shrinking phase, transactions can release locks but cannot acquire new ones.

- Guarantees serializability and avoids conflicts between transactions.

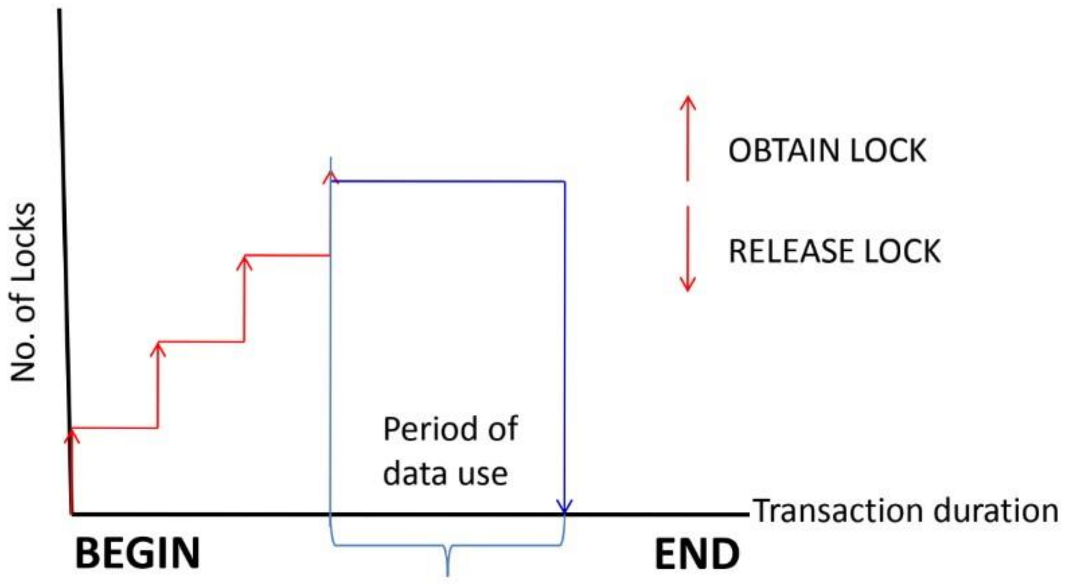


2. Strict Two-Phase Locking Algorithm (S2PL):

- A stricter version of 2PL.

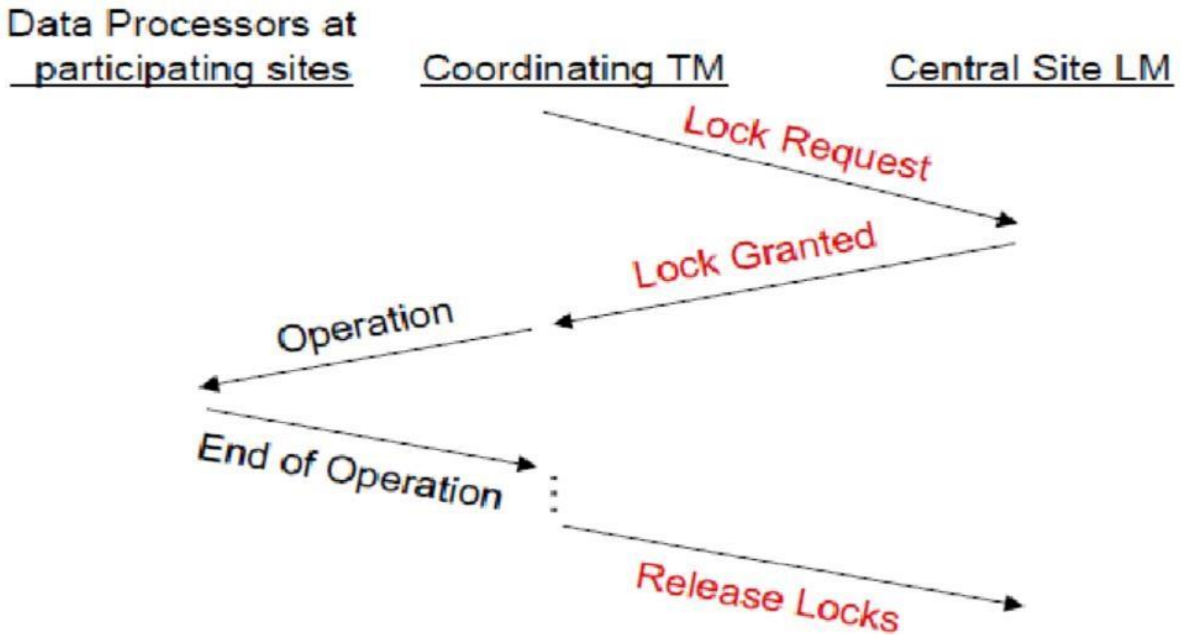
- Transactions hold all locks they acquire until they commit, and no locks are released before then.

- Ensures that there is no overlap in the locks held by different transactions, preventing the possibility of cascading aborts.



3. Centralized Two-Phase Locking (C2PL):

- In a distributed database system, a centralized server manages the lock requests and grants locks to transactions.
- Ensures that all lock requests are coordinated through a single point of control.
- Can be effective for maintaining global consistency but can also introduce a single point of failure and potential performance bottlenecks.

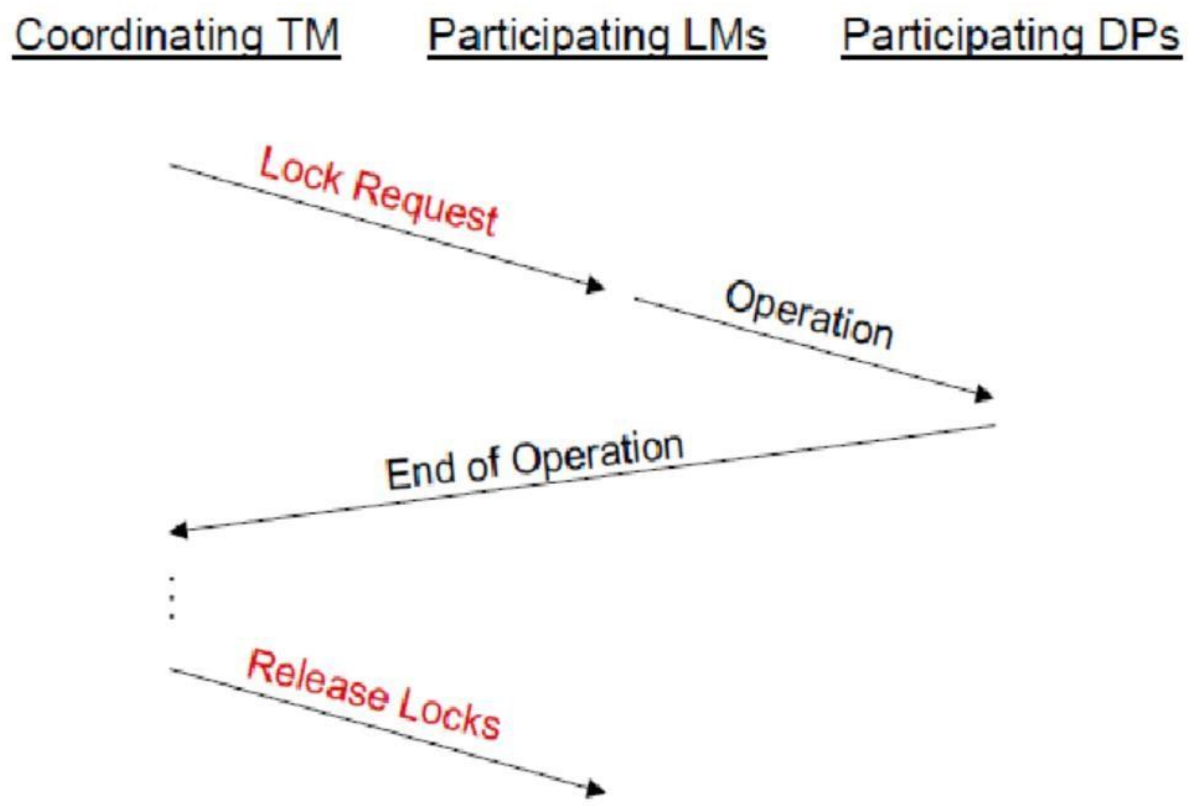


4. Distributed Two-Phase Locking (D2PL):

- In a distributed database system, each site manages its own lock requests and grants locks locally.

- Lock coordination occurs across multiple sites to ensure consistency.

- Reduces the centralized control bottleneck of C2PL but requires careful management of distributed lock management.



- Advantages:

- Provides a systematic way to manage concurrent transactions and prevent data inconsistencies.

- Ensures that transactions are serialized, maintaining the integrity of the database.

- Allows for flexibility in determining the appropriate level of locking granularity.

- Challenges:

- Deadlocks can occur if not managed properly, leading to blocked transactions.

- Overuse of locks can lead to reduced concurrency and performance degradation.
- Proper tuning is required to strike a balance between locking overhead and system performance.

Lock-based concurrency control methods are widely used in database systems to ensure the correct and consistent execution of concurrent transactions. While they provide effective mechanisms for maintaining data integrity, careful design and tuning are essential to avoid issues like deadlocks and excessive locking.

Time-Stamp Based Concurrency Control Algorithms:

Time-stamp based concurrency control is a method used to manage concurrent access to data in a database system. It involves assigning time-stamps to transactions and data items to determine the order of operations and ensure serializability. This approach uses the concept of time-stamps to decide which operation should be executed and when.

Time-Stamp Based Concurrency Control Algorithms:

1. Timestamp Ordering (TO):

- Each transaction is assigned a time-stamp when it enters the system.
- For each data item, a read and a write time-stamp are maintained.
- A transaction can read a data item if its time-stamp is greater than or equal to the item's write time-stamp.
- A transaction can write a data item if its time-stamp is greater than both the item's read and write time-stamps.
- Ensures conflict-serializability but may lead to cascading aborts.

2. Multiversion Timestamp Ordering (MVTO):

- Allows multiple versions of a data item to coexist, each with its own write time-stamp.
- Read operations can access the appropriate version based on the transaction's time-stamp.
- Write operations create a new version of the item with an updated write timestamp.
- Reduces contention and allows better concurrency but increases storage requirements.

3. Conservative Timestamp Ordering:

- Combines timestamp ordering with a locking mechanism.
- A transaction must acquire appropriate locks before performing read or write operations.
- Ensures serializability and avoids cascading aborts.

Advantages and Challenges:

- Advantages:
- Provides a natural way to order transactions without explicit locks.
- Allows greater concurrency compared to strict locking methods.
- Well-suited for distributed environments and systems with dynamic workload.

- Challenges:
- Proper time-stamp management is crucial to ensure correctness.

- Handling of older transactions that arrive late can be complex.
- Handling of system failures and synchronization of global time can be challenging.

Time-stamp based concurrency control algorithms offer an effective way to manage concurrent transactions without the explicit use of locks. However, they require careful design and management of time-stamps and versions to ensure correctness and efficiency.

• Optimistic concurrency control algorithms

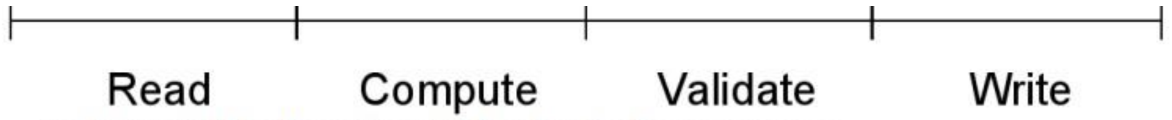
– Delay the validation phase until just before the write phase

– Ti run independently at each site on local copies of the DB (without updating the DB)

– Validation test then checks whether the updates would maintain the DB consistent:

∗ If yes, all updates are performed

∗ If one fails, all Ti's are rejected



 Potentially allow for a higher level of concurrency

Sure, here's the organized and formatted version of the information you provided:

Deadlock Management: Prevention, Avoidance, Detection, and Resolution

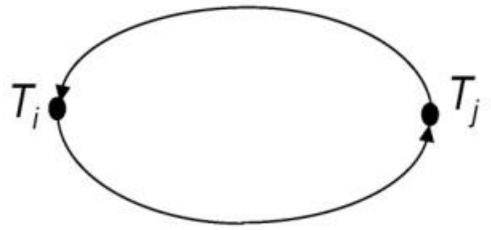
Deadlock:

A deadlock occurs when a set of transactions are waiting for each other, and intervention from outside is needed to resolve the situation.

- Any locking-based concurrency control algorithm can result in a deadlock as transactions wait for locks.
- Some Timestamp Ordering (TO) based algorithms that require transaction waiting can also cause deadlocks.

Wait-for Graph (WFG):

- Nodes represent transactions, edges indicate transactions waiting for each other.
- If the WFG has a cycle, a deadlock is present.



Global Deadlock:

Deadlock management in a Distributed DBMS (DDBMS) is complex, with the potential for global deadlocks involving transactions from different sites.

Deadlock Prevention:

- Guarantee that deadlocks never occur.
- Transaction initiation is conditional on all required resources being available.
- Resources needed by a transaction must be predeclared.
- Advantages: No rollbacks or restarts, no runtime support.

- Disadvantages: Reduced concurrency, evaluating safe allocations adds overhead, predicting required resources is difficult.

Deadlock Avoidance:

- Detect potential deadlocks in advance and prevent their occurrence.
- Two approaches: Ordering data items and sites, or prioritizing transactions.
- Wait-Die Scheme: Transactions wait or die based on their timestamps.
- Wound-Wait Scheme: Older transactions wound (abort) newer ones.
- Advantages: Better concurrency than prevention, transactions not required to pre-allocate resources.
- Disadvantages: Requires runtime support.

Deadlock Detection and Resolution:

- Transactions are allowed to wait freely, deadlocks are checked using the global WFG for cycles.
- Detected deadlocks are resolved by aborting one of the involved transactions (victim).
- Advantages: Maximal concurrency, well-studied method.
- Disadvantages: Can undo considerable work.

Topologies for Deadlock Detection Algorithms:

Certainly, here's the information about the topologies for deadlock detection algorithms:

Topologies for Deadlock Detection Algorithms:

1. Centralized Deadlock Detection:

- In this approach, a single site is designated as the Deadlock Detection Coordinator (DDC) for the entire distributed system.
- Each scheduler periodically sends its Local Wait-for Graph (LWFG) information to the central DDC.
- The DDC merges the LWFGs received from all sites to create a Global Wait-for Graph (GWFG) representing the entire system.
- The DDC then analyzes the GWFG to detect cycles (deadlocks).
- If cycles are found, the DDC identifies transactions involved in the cycles as victims and initiates their rollback and restart to resolve the deadlock.
- This approach is reasonable when the concurrency control algorithm is also centralized, as in some cases.

2. Hierarchical Deadlock Detection:

- In this topology, the sites are organized in a hierarchical structure.
- Each site sends its LWFG information to the site directly above it in the hierarchy.
- The higher-level site collects and merges the LWFGs from its lower-level sites to form a higher-level LWFG.
- This process continues until the top-level site has the merged GWFG.
- This approach reduces dependence on a single centralized detection site, distributing the detection process across the hierarchy.

3. Distributed Deadlock Detection:

- In this topology, sites collaborate in detecting deadlocks, and no single site is designated as the central detector.

- Each site maintains its own LWFG based on the local transactions and waits-for relationships.
- Additionally, each site receives information about potential deadlock cycles from other sites' LWFGs.
- The received information is added to the local LWFG, combining the waiting edges from local and external sources.
- Local deadlock detectors analyze their modified LWFGs to identify two types of situations:
- Local Deadlock: If there's a cycle within the local LWFG, indicating a deadlock involving local transactions.
- Potential Global Deadlock: If there's a cycle involving external edges (edges from other sites), indicating a potential global deadlock.

Overview of Object-Oriented Concepts

Object-Oriented (OO) databases strive to establish a direct and consistent link between real-world entities and database objects. This connection ensures that the integrity and identity of objects remain intact, enabling easy identification and manipulation within the database environment.

Object:

An object is a fundamental building block in OO databases, encompassing two essential components:

1. State (Value): This pertains to the data or information associated with the object. It represents the current attributes and properties that define the object.

2. Behavior (Operations): Objects are not just passive data containers; they encapsulate behavior or operations that can be performed on the object. These operations are akin to functions or methods in programming languages.

Objects in OO databases are more intricate than variables in traditional programming languages. They include both complex data structures and specific operations defined by the programmer. In OO databases, objects can possess intricate structures to accommodate comprehensive information about the object. This stands in contrast to conventional databases where information about a complex object may be spread across multiple relations or records, leading to a disconnect between the object and its database representation.

Encapsulation:

Encapsulation is a central concept in OO databases. It involves grouping an object's state (data) and behavior (operations) into a self-contained unit.

This isolation shields the internal details of an object from external manipulation. To reinforce encapsulation, operations are defined in two distinct parts:

1. Signature or Interface: This specifies the operation's name and the arguments it accepts (parameters).

2. Method or Body: This outlines the actual implementation of the operation.

Operations are invoked by sending a message to an object, containing the operation's name and any necessary parameters. The object then executes the corresponding method for that operation. This approach empowers modifications to an object's internal structure and operation implementations without affecting external programs that interact with these operations.

Versioning:

Certain OO systems facilitate handling multiple versions of the same object. This functionality is indispensable in design and engineering domains. For instance, in fields such as manufacturing process control, architecture, and software systems, retaining previous versions of an object until new versions are validated is critical. This ensures that reliable and proven designs are maintained until new versions are thoroughly tested and verified.

Operator Polymorphism (Operator Overloading):

Operator polymorphism refers to the capacity of an operation to be applied to diverse types of objects. In such cases, an operation's name might correspond to distinct implementations based on the object's type. This feature is also known as operator overloading, as an operation can "overload" its functionality to accommodate different data types or object structures.

In essence, OO databases uphold a comprehensive framework where objects encapsulate both data and behavior, facilitating direct connections between real-world entities and their database representations. This paradigm permits effective encapsulation, version management, and operator polymorphism, enhancing the versatility and integrity of database systems.

Object Identity:

Any object has 2 properties i.e. state & behaviour hence, it's similar to the program variable but object has complex structure. There are two types of object :- 1.) Transient object 2.) Persistent object

1.) **Transient object:** - These objects exist during the execution but destroy ones the program terminates.

2.) Persistent object: - These objects persist & store into database even if program terminate. Whenever any object created DBMS assigns unique identifier called as object identity & its abbreviated as OID.

OID is used for two purposes.

- To identify each objects uniquely.
- Make reference to another object.

Properties of OID

- Its unique
- Its system generated
- Its invisible to the user
- It's not possible t perform operation on OID
- It's inmutable i.e. once its generated
- It cannot be regenerated
- It has along integer values

Object structure

Every object has internal structure that can be specified by using 6 types of type constructor (AST LAM)…

1. ATDM: - Internal structure of object consists of automatic values that may be of type integer float, character, string, data etc.
2. SET:- Internal structure of object, consist of collection of OID's of other objects that may be constructed from different type constructor its denoted by {i1 , i2 , i3 , …}
3. Tuples:- Internal structure of object, consist of collection of OID's other objects that may be constructed from different type constructor its denoted by {a1 : i1 ; a2 : i2 ; a3 : i3, …} where a1 , a2 , a3 , … be the attributers & i1 , i2 , i3 be OID's
4. List :- Its similar to the set but it's an ordered collection its denoted by { i1 , i2 , i3 , …}
5. ARRAY: - It's similar to list but it has limitations I.e. it's of fixed size.
6. MULTISET or bag: - It's similar to the set but it may contain duplicate. Any object structure is represented

By triplet i.e. {I, C, V} I

= Object identifier.

E = type constructer.

V = value.

For e.g.:- Let us consider the following relation.

Dno	**Dname**	**- - -**
5	Research	- - -

Location:-

Dno	**Dname**

5	4
5	12

For above 2 relations object structure can be created as follows:-

01 = <i1, atom,="" 5="">

02 = <i2, atom,="" “research="">”

03 = <i3, atom,="" “I1”&gt;<="" p="">

04 = <i4, atom,="" “I2”&gt;<="" p="">

05 = <i5, set,="" <i3,="" i4="" &gt;<="" p="">

06 = <i6, tuple="" ;&lt;="" dno;="" i="" 1,="" d="" name="" i2,="" coc=""> ▯

Type constructor

It’s used to define the data structure for object oriented, DB there are six types of type constructor. (Refer objects structure) Let us consider employee & department relation.

Type constructor can be created as follows:-

```
Define type employees
Type   (id: integer;
 Name: string,
 Gender: string,
 DOB: date,
 Address: string,
 Salary: Float
 Manager: Employee,
    D no: Department.)
    End;



   Define type name
   Type (fname
             Mname
             Dname)
             End;


     Define type address
       Type (apt no:
                 Area:
                 City:
                 Pin code:
End;   )


    Define type department
    Type (dno: integer
               Dname: string
              Location: set (string), manager
                                    Employees)
               End;
```

Example 1

- One possible relational database state corresponding to COMPANY schema

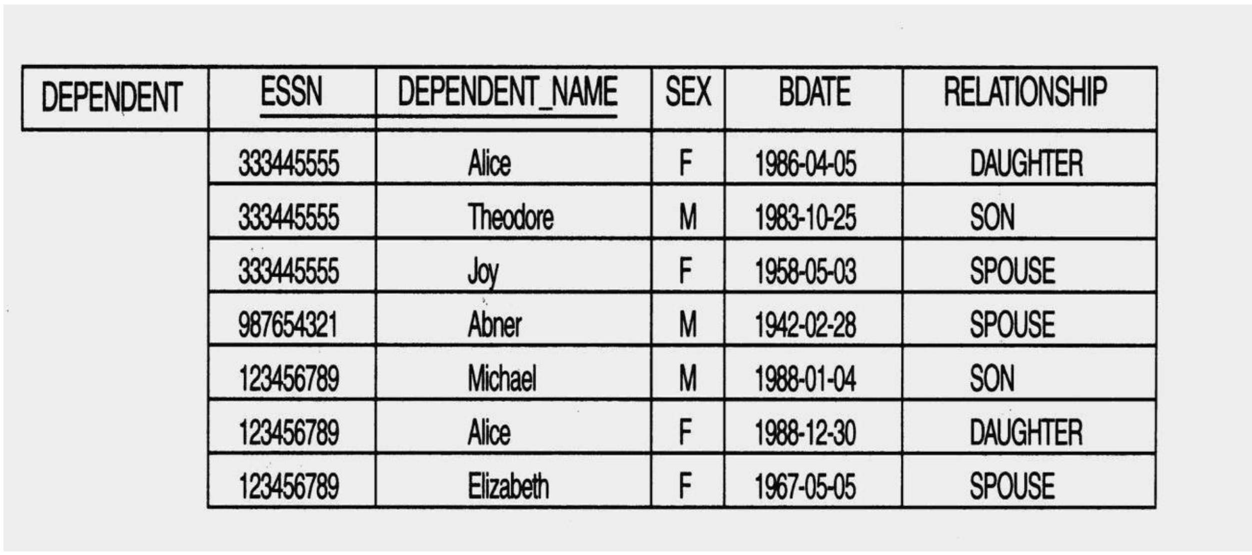
EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4



DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

We use i_1, i_2, i_3, . . . to stand for unique system-generated object identifiers. Consider the following objects: ■ o_1 = (i_1, atom, 'Houston') o_2 = (i_2, atom, 'Bellaire') o_3 = (i_3, atom, 'Sugarland') o_4 = (i_4, atom, 5) o_5 = (i_5, atom, 'Research') o_6 = (i_6, atom, '1988-05-22') o_7 = (i_7, set, {i1, i2, i3})

o_8 = (i_8, tuple, <dname:i_5, dnumber:i_4, mgr:i_9, locations:i_7, employees:i_{10}, projects:i_{11}>)

o_9 = (i_9, tuple, <manager:$i_1$2, manager_start_date:i_6>)

o_{10} = (i_{10}, set, {i_{12}, i_{13}, i_{14}}) o_{11}

= (i_{11}, set {i_{15}, i_{16}, i_{17}})

o_{12} = (i_{12}, tuple, <fname:i_{18}, minit:i_{19}, lname:i_{20}, ssn:i_{21}, . . ., salary:i_{26}, supervisor:i_{27}, dept:i_8>)

- The first six objects listed in this example represent atomic values. ○ Object seven is a set-valued object that represents the set of locations for department 5; the set refers to the atomic objects with values {'Houston', 'Bellaire', 'Sugarland'}.

- Object 8 is a tuple-valued object that represents department 5 itself, and has the attributes DNAME, DNUMBER, MGR, LOCATIONS, and so on.

Example 2:

This example illustrates the difference between the two definitions for comparing object states for equality.

$o_1 = (i_1, \text{tuple}, <a_1{:}i_4, a_2{:}i_6>)$

$o_2 = (i_2, \text{tuple}, <a_1{:}i_5, a_2{:}i_6>)$

$o_3 = (i_3, \text{tuple}, <a_1{:}i_4, a_2{:}i_6>)$

$o_4 = (i_4, \text{atom}, 10)$ $o_5 = (i_5,$

atom, 10) $o_6 = (i_6,$ atom,

20)

- In this example, The objects o1 and o2 have equal states, since their states at the atomic level are the same but the values are reached through distinct objects o4 and o5.
- However, the states of objects o1 and o3 are identical, even though the objects themselves are not because they have distinct OIDs.
- Similarly, although the states of o4 and o5 are identical, the actual objects o4 and o5 are equal but not identical, because they have distinct OIDs.

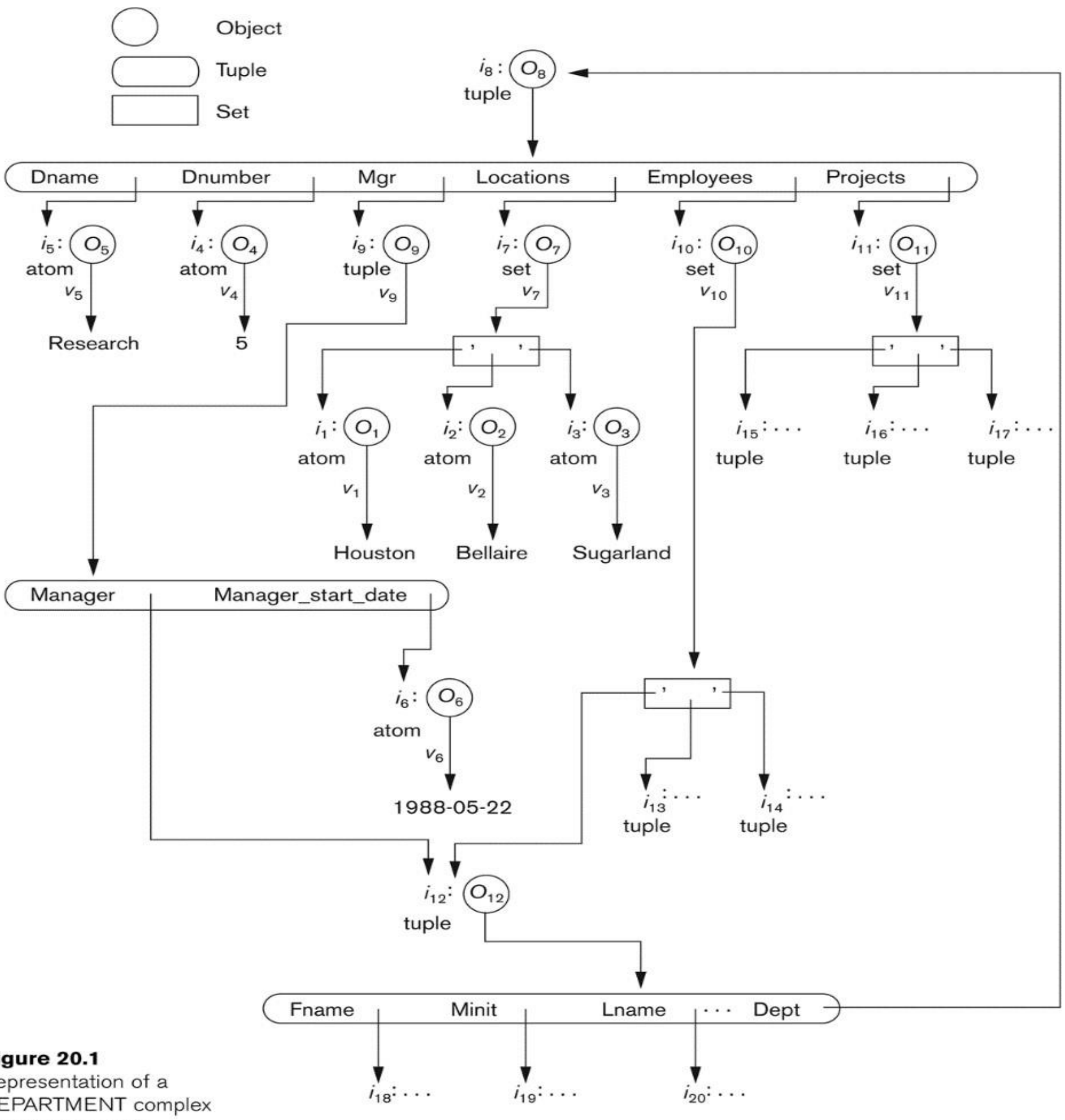


Figure 20.1
Representation of a DEPARTMENT complex object as a graph.

```
define type EMPLOYEE
   tuple (  Fname:        string;
            Minit:        char;
            Lname:        string;
            Ssn:          string;
            Birth_date:   DATE;
            Address:      string;
            Sex:          char;
            Salary:       float;
            Supervisor:   EMPLOYEE;
            Dept:         DEPARTMENT;

define type DATE
   tuple (  Year:         integer;
            Month:        integer;
            Day:          integer; );

define type DEPARTMENT
   tuple (  Dname:        string;
            Dnumber:      integer;
            Mgr:          tuple (  Manager:     EMPLOYEE;
                                   Start_date:  DATE;     );
            Locations:    set(string);
            Employees:    set(EMPLOYEE);
            Projects      set(PROJECT);    );
```

Figure 20.2
Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

Encapsulation of Operations, Methods, and Persistence

- **Encapsulation** ○ One of the main characteristics of OO languages and systems ○ Related to the concepts of **abstract data types** and **information hiding** in programming languages
- Specifying **Object Behavior** via Class Operations:

1. The main idea is to define the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type.
2. In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.
3. For database applications, the requirement that all objects be completely encapsulated is too stringent.
4. One way of relaxing this requirement is to divide the structure of an object into visible and **hidden attributes** (**instance variables**).

```
define class EMPLOYEE
   type tuple  (   Fname:          string;
                   Minit:          char;
                   Lname:          string;
                   Ssn:            string;
                   Birth_date:     DATE;
                   Address:        string;
                   Sex:            char;
                   Salary:         float;
                   Supervisor:     EMPLOYEE;
                   Dept:           DEPARTMENT;   );
   operations      age:            integer;
                   create_emp:     EMPLOYEE;
                   destroy_emp:    boolean;
end EMPLOYEE;

define class DEPARTMENT
   type tuple  (   Dname:          string;
                   Dnumber:        integer;
                   Mgr:            tuple (   Manager:       EMPLOYEE;
                                             Start_date:    DATE;     );
                   Locations:      set(string);
                   Employees:      set(EMPLOYEE);
                   Projects        set(PROJECT);    );
   operations      no_of_emps:     integer;
                   create_dept:    DEPARTMENT;
                   destroy_dept:   boolean;
                   assign_emp(e:   EMPLOYEE): boolean;
                   (* adds an employee to the department *)
                   remove_emp(e: EMPLOYEE): boolean;
                   (* removes an employee from the department *)
end DEPARTMENT;
```

Figure 20.3
Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

Specifying Object Persistence via Naming and Reachability:

- **Naming Mechanism**:

  Assign an object a unique persistent name through which it can be retrieved by this and other programs.

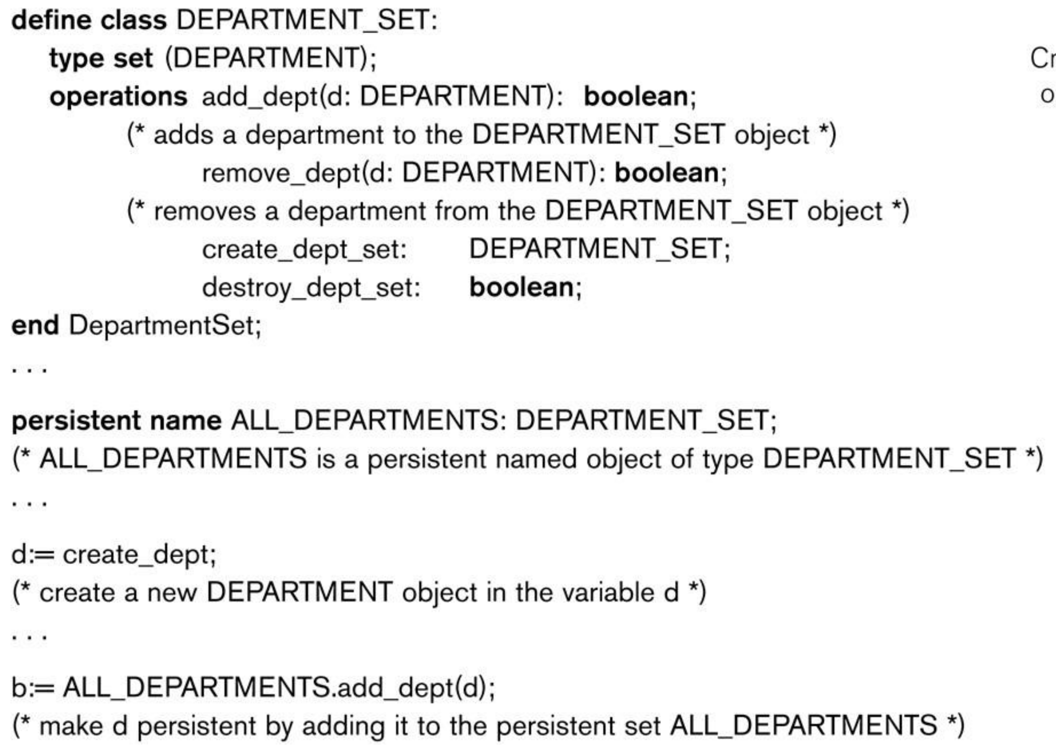
- **Reachability Mechanism**:

  Make the object reachable from some persistent object.

  An object B is said to be **reachable** from an object A if a sequence of references in the object graph lead from object A to object B.

In traditional database models such as relational model or EER model, all objects are assumed to be persistent.

In OO approach, a class declaration specifies only the type and operations for a class of objects. The user must separately define a **persistent object** of **type set** (DepartmentSet) or **list** (DepartmentList) whose value is the collection of references to all persistent DEPARTMENT objects



```
define class DEPARTMENT_SET:
   type set (DEPARTMENT);
   operations add_dept(d: DEPARTMENT):  boolean;
         (* adds a department to the DEPARTMENT_SET object *)
               remove_dept(d: DEPARTMENT): boolean;
         (* removes a department from the DEPARTMENT_SET object *)
               create_dept_set:      DEPARTMENT_SET;
               destroy_dept_set:     boolean;
end DepartmentSet;
. . .
persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
. . .
d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
. . .
b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

Figure 20.4
Creating persistent objects by naming and reachability.

Type and Class Hierarchies and Inheritance

- **Type (class) Hierarchy**

  A type in its simplest form can be defined by giving it a type name and then listing the names of its visible (**public**) functions

  When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

  TYPE_NAME: function, function, . . . , function ■

  Example:

  PERSON: Name, Address, Birthdate, Age, SSN

- **Subtype**:

When the designer or user must create a new type that is similar but not identical to an already defined type ■ **Supertype**:

It inherits all the functions of the subtype ■

Example (1):

PERSON: Name, Address, Birthdate, Age, SSN

EMPLOYEE: Name, Address, Birthdate, Age, SSN, Salary, HireDate, Seniority

STUDENT: Name, Address, Birthdate, Age, SSN, Major, GPA

OR:

EMPLOYEE **subtype-of** PERSON: Salary, HireDate, Seniority

STUDENT **subtype-of** PERSON: Major, GPA

■ Example (2):

Consider a type that describes objects in plane geometry, which may be defined as follows:

GEOMETRY_OBJECT: Shape, Area, ReferencePoint

Now suppose that we want to define a number of subtypes for the GEOMETRY_OBJECT type, as follows:

RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height

TRIANGLE **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle

CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

An alternative way of declaring these three subtypes is to specify the value of the Shape attribute as a condition that must be satisfied for objects of each subtype:

RECTANGLE **subtype-of** GEOMETRY_OBJECT (Shape='rectangle'): Width, Height

TRIANGLE **subtype-of** GEOMETRY_OBJECT (Shape='triangle'): Side1, Side2, Angle

CIRCLE **subtype-of** GEOMETRY_OBJECT (Shape='circle'): Radius

Extents:

1. In most OO databases, the collection of objects in an extent has the same type or class.
2. However, since the majority of OO databases support types, we assume that **extents** are collections of objects of the same type for the remainder of this section.

- **Persistent Collection**:

  This holds a collection of objects that is stored permanently in the database and hence can be accessed and shared by multiple programs

- **Transient Collection**:

  This exists temporarily during the execution of a program but is not kept when the program terminates

Complex Objects

- **Unstructured complex object**:

These is provided by a DBMS and permits the storage and retrieval of large objects that are needed by the database application.

Typical examples of such objects are bitmap images and long text strings (such as documents); they are also known as binary large objects, or BLOBs for short.

This has been the standard way by which Relational DBMSs have dealt with supporting complex objects, leaving the operations on those objects outside the RDBMS.

- **Structured complex object**:

This differs from an unstructured complex object in that the object's structure is defined by repeated application of the type constructors provided by the OODBMS.

Hence, the object structure is defined and known to the OODBMS.

The OODBMS also defines methods or operations on it.

Other Objected-Oriented Concepts

- **Polymorphism** (**Operator Overloading**):
- This concept allows the same **operator name or symbol** to be bound to two or more **different implementations** of the operator, depending on the type of objects to which the operator is applied For example + can be:

  Addition in integers

  Concatenation in strings (of characters)

- Multiple Inheritance and Selective Inheritance

  Multiple inheritance in a type hierarchy occurs when a certain subtype T is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes.

  For example, we may create a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER.

  This leads to the creation of a type lattice rather than a type hierarchy.

- **Versions and Configurations**

  Many database applications that use OO systems require the existence of several **versions** of the same object There may be more than two **versions** of an object.

- **Configuration**:

  A **configuration** of the complex object is a collection consisting of one version of each module arranged in such a way that the module versions in the configuration are compatible and together form a valid version of the complex object.

Summary:

- **Object identity**:

Objects have unique identities that are independent of their attribute values.

- **Type constructors**:

Complex object structures can be constructed by recursively applying a set of basic constructors, such as tuple, set, list, and bag.

- **Encapsulation of operations**:

Both the object structure and the operations that can be applied to objects are included in the object class definitions.

- **Programming language compatibility**:

Both persistent and transient objects are handled uniformly. Objects are made persistent by being attached to a persistent collection.

- **Type hierarchies and inheritance**:

Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods of previously defined types.

- **Extents**:

All persistent objects of a particular type can be stored in an extent. Extents corresponding to a type hierarchy have set/subset constraints enforced on them.

- **Support for complex objects**:

Both structured and unstructured complex objects can be stored and manipulated.

- **Polymorphism and operator overloading**:

Operations and method names can be overloaded to apply to different object types with different implementations.

- **Versioning**:

Some OO systems provide support for maintaining several versions of the same object.

Object Model

The object model is a foundational framework used in object-oriented databases to provide a standardized way of representing and managing data. It encompasses various components that define how data is organized, defined, and queried within the database. Here's an organized explanation of the key components and characteristics of the object model:

1. Standardized Representation:

The object model offers a standardized representation for object-oriented databases. It establishes a consistent framework that ensures compatibility and uniformity across different databases and systems.

2. Object Definition and Querying:

The object model facilitates the definition of objects within the database through Object Definition Language (ODL). Additionally, it supports querying operations using Object Query Language (OQL), enabling efficient retrieval and manipulation of data.

3. Data Types and Type Constructors:

The object model supports a diverse range of data types and type constructors. This flexibility allows developers to define complex data structures that accurately mirror real-world entities and relationships.

4. Basic Building Blocks:

At the core of the object model are two fundamental building blocks:

a. Objects:

Objects are the primary units of data in the object model. They encapsulate both data and behavior, resembling real-world entities. Each object possesses four key characteristics:

- Identifier: A system-wide unique identifier that distinguishes the object from all others.
- Name: An optional attribute that provides a unique identifier within a specific database or program context.
- Lifetime: Objects can have either a persistent or transient lifetime. Persistent objects are stored in the database and retain their state across sessions, while transient objects are short-lived and not stored.
- Structure: Specifies the object's composition using a type constructor. It defines whether the object is atomic or composed of other objects or values.

b. Literals:

Literals represent constant values that do not have identifiers. They are used to define the content of objects and serve as the data within the database. There are three types of literals:

- Atomic Literals: These are predefined basic data type values, such as integers, floats, booleans, and characters.
- Structured Literals: Values constructed using type constructors, like dates or structured variables, which group related data together.
- Collection Literals: These represent collections of values or objects, such as arrays or lists.

Object Query Language (OQL)

Object Query Language (OQL) is a specialized query language designed for querying object-oriented databases. OQL allows users to retrieve and manipulate data stored in object-oriented database systems using a syntax that is tailored to the principles and structures of object-oriented programming. OQL is analogous to SQL (Structured Query Language), which is used for querying relational databases, but OQL is adapted to the complex structures and behaviors of objects in object-oriented databases.

Key Features of OQL

1. Object-Centric Approach OQL revolves around objects as its primary entities. It allows users to perform queries and retrieve information from objects, considering both their data (attributes or properties) and behaviors (methods or operations).

2. Navigation OQL supports navigation through object relationships. In objectoriented databases, objects are often interconnected through relationships, such as composition, aggregation, or inheritance. OQL provides syntax to traverse these relationships and access related objects.

3. Selection and Projection OQL offers mechanisms for selecting specific objects based on certain criteria and projecting specific attributes or properties from those objects. This is similar to the SELECT and PROJECT operations in SQL.

4. Filtering OQL enables users to filter objects based on conditions, allowing them to retrieve objects that satisfy specific criteria. This is akin to the WHERE clause in SQL.

5. Joining Just as SQL supports joining tables based on common attributes, OQL supports joining objects based on relationships between them.

6. Aggregation OQL includes aggregation functions that allow users to perform calculations on groups of objects and retrieve summary information, such as sums, averages, and counts.

7. Sorting OQL permits users to sort the results of queries based on specified attributes, similar to the ORDER BY clause in SQL.

8. Subqueries Users can nest queries within queries, enabling more complex and refined data retrieval operations.

9. Polymorphism OQL supports querying based on polymorphic relationships, allowing users to retrieve objects of different types that share a common relationship.

10. Encapsulation OQL respects the encapsulation principles of object-oriented programming by providing access to object data and methods only as defined by the object's interface.

Example OQL Query

Suppose we have an object-oriented database of employees and departments, with relationships between employees and their respective departments. An example OQL query might look like this:

```
SELECT e.name, d.departmentName
FROM Employee e
WHERE e.salary > 50000
  AND e.department = Department d
  AND d.location = 'New York'
ORDER BY e.name ASC;
```

In this query, we're retrieving the names of employees and the names of their departments for employees with a salary greater than $50,000 and who work in the 'New York' department. The results are sorted alphabetically by the employees' names.

Object Definition Language (ODL)

Object Definition Language (ODL) is a specialized language used for defining the structure, attributes, relationships, and behavior of objects within an objectoriented database (OODB). ODL provides a standardized and formal way to create, modify, and manage the schema of object-oriented databases, much like how SQL (Structured Query Language) is used for relational databases.

Key Features and Aspects of ODL:

1. Schema Definition: ODL allows developers to define the schema of objects and their relationships in an object-oriented database. This includes specifying the attributes (data members) and methods (operations) that an object type will have.

2. Object Types: ODL enables the creation of user-defined object types. These types serve as blueprints for creating actual instances of objects in the database.

3. Inheritance: Similar to object-oriented programming languages, ODL supports inheritance. Objects can inherit attributes and methods from other object types, promoting code reuse and hierarchical relationships.

4. Encapsulation: ODL enforces encapsulation principles by allowing access control to attributes and methods. It defines whether an attribute or method is private, protected, or public.

5. Relationships: ODL supports the specification of relationships between object types. This includes associations (relationships between objects), aggregations (whole-part relationships), and generalizations (subtyping relationships).

6. Collections: ODL facilitates the definition of collection types, such as arrays or lists, within object types. This allows objects to hold multiple values or references to other objects.

7. User-Defined Methods: ODL enables the definition of methods that can be invoked on objects. These methods encapsulate the behavior associated with the object type.

8. Constraints: ODL allows the definition of integrity constraints that govern the validity and consistency of data within the database.

9. Data Types: ODL supports a wide range of data types, both predefined and user-defined. This includes primitive types (e.g., integers, strings) as well as complex types (e.g., custom classes).

Example ODL Code:

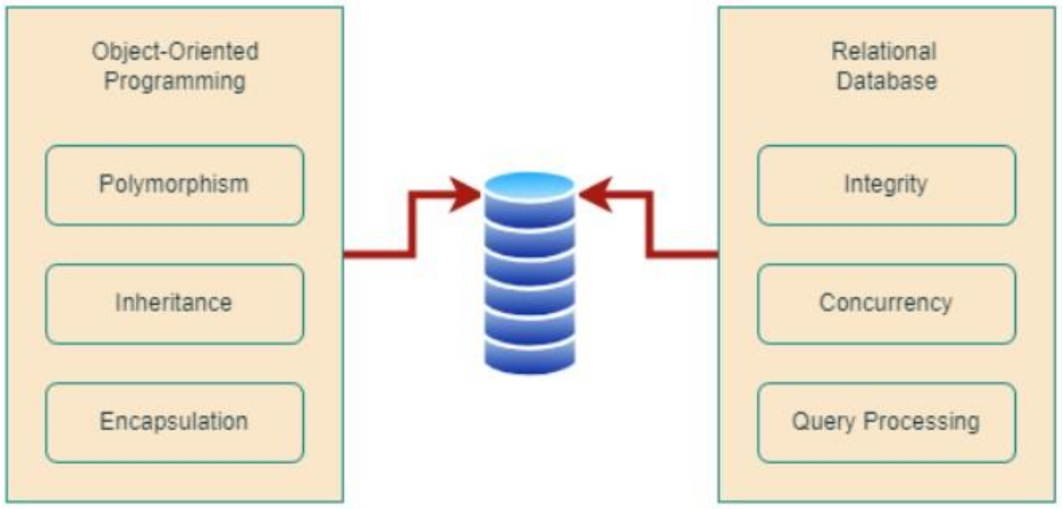
Consider an example where we define an object type "Student" with attributes "name," "rollNumber," and "courses" (an array of courses the student is enrolled in):

```
odl
OBJECT TYPE Student
{
   ATTRIBUTES        name:
STRING,        rollNumber:
INTEGER,        courses: ARRAY
OF Course     METHODS
      enroll(course: Course),
drop(course: Course)
}
```

In this ODL code snippet, we define the structure of the "Student" object type with its attributes and methods. The "Course" type mentioned within the "courses" attribute is assumed to be defined elsewhere in the schema.

Object database conceptual design

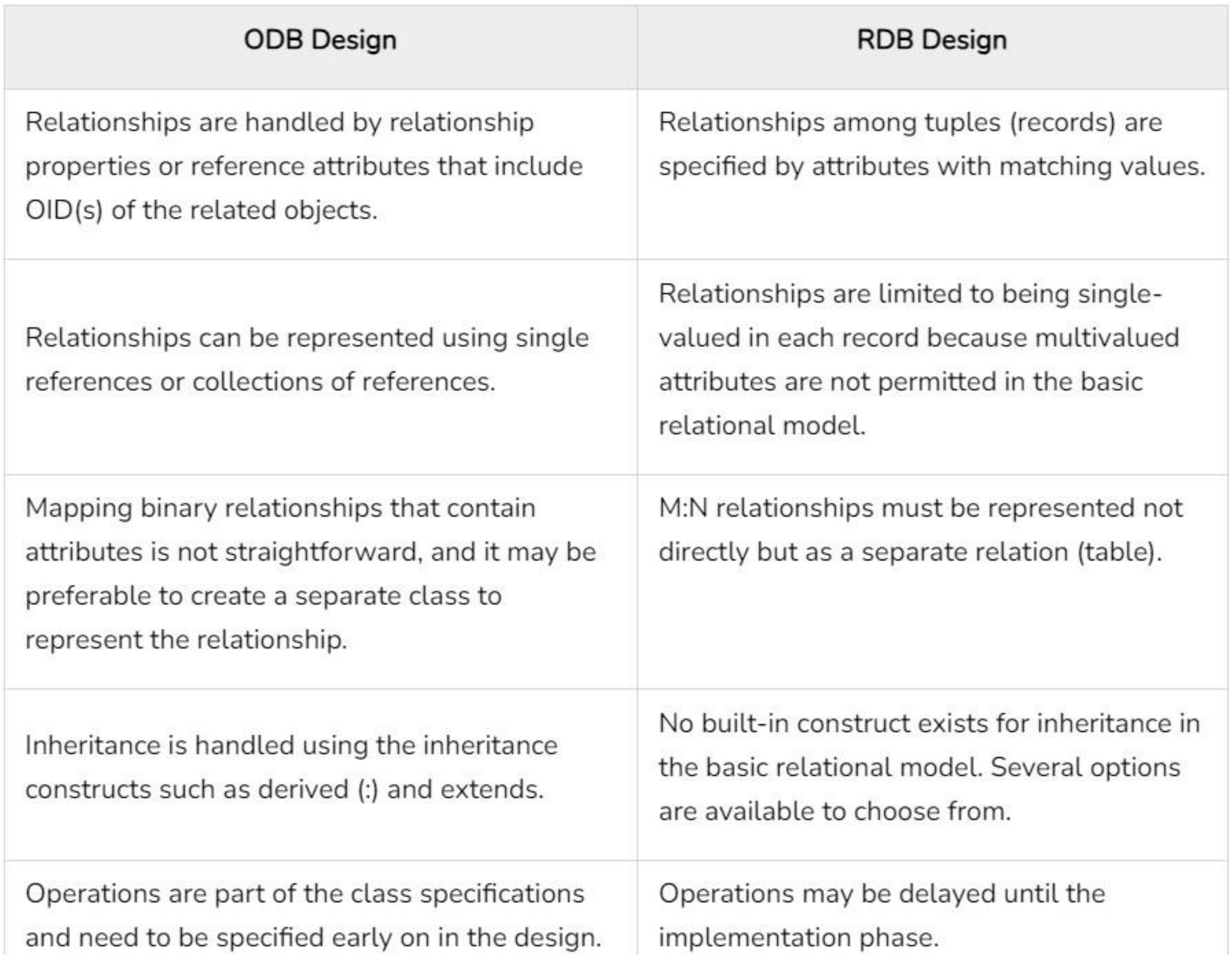
Object-Oriented Model is process of representing real-world entities as objects in computer programs. It is fundamental aspect of Object Database Conceptual Design. Object-oriented modeling is more flexible and dynamic than traditional relational model. It enables developers to model complex relationships between objects more efficiently. Object-oriented modeling allows for inheritance, encapsulation, and polymorphism. It is easier to maintain and modify code.



Object Database

An object database stores data in the form of objects. These objects can contain both data and methods for accessing and manipulating that data. Object databases work with object-oriented programming languages, like Java and Python

Differences between Conceptual Design of ODB and RDB:



ODB Design	RDB Design
Relationships are handled by relationship properties or reference attributes that include OID(s) of the related objects.	Relationships among tuples (records) are specified by attributes with matching values.
Relationships can be represented using single references or collections of references.	Relationships are limited to being single-valued in each record because multivalued attributes are not permitted in the basic relational model.
Mapping binary relationships that contain attributes is not straightforward, and it may be preferable to create a separate class to represent the relationship.	M:N relationships must be represented not directly but as a separate relation (table).
Inheritance is handled using the inheritance constructs such as derived (:) and extends.	No built-in construct exists for inheritance in the basic relational model. Several options are available to choose from.
Operations are part of the class specifications and need to be specified early on in the design.	Operations may be delayed until the implementation phase.

Mapping an EER Schema to an ODB Schema

Mapping an EER schema to an ODB schema is a process of designing the type declarations of object classes for an Object Database Management System (ODBMS) from an Entity-Relationship (ER) diagram. Here are the step-wise instructions to perform this mapping:

Step-1

Create an ODL class for each EER entity type or subclass. The ODL class should include all the attributes of the EER class. Multivalued attributes are typically declared by using set, bag, or list constructors. Declare an extent for each class and specify any key attributes as keys of the extent.

Step-2

Add relationship properties or reference attributes for each binary relationship into the ODL classes that participate in the relationship.

Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types.

Step-3

Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements.

Step-4

An ODL class that corresponds to a subclass in the EER schema inherits the type and methods of its superclass in the ODL schema. Its specific (noninherited) attributes, relationship references, and operations are specified.

Step-5

Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship.

Step-6

Categories (union types) in an EER schema are difficult to map to ODL. One way is to create a class to represent the category and define 1:1 relationships between the category and each of its superclasses.

Step-7

An n-ary relationship with degree n > 2 can be mapped into a separate class, with appropriate references to each participating class. An M:N binary relationship may also use this mapping option if desired.

After completing the structural mapping, add operations for each class, including constructor and destructor methods that check for constraints. The mapping can be applied to a subset of the database schema in the context of the ODMG object database standard.

Challenges in Object Database Design

Object Database Conceptual Design has many benefits and challenges. It can be complex. It may require significant amount of time and effort to create a robust and efficient object model. Integration with existing systems can also be challenging. As object databases may need to work with other databases that use the relational model. Performance issues can arise due to the complex relationships between objects.

Examples of ODBMSs

There are several Object-Oriented Database Management Systems (ODBMSs) that have been developed over the years to support the storage, retrieval, and management of data in an object-oriented manner. Here are some examples of ODBMSs:

1. ObjectStore: Developed by Object Design Inc., ObjectStore is one of the early and prominent ODBMSs. It offers a comprehensive solution for managing complex data structures, including support for persistence, transactions, and query capabilities.

2. Versant: Versant is another well-known ODBMS that provides support for managing complex, large-scale object-oriented applications. It offers features like transparent persistence, support for distributed systems, and advanced query optimization.

3. db4o (database for objects): db4o is an open-source ODBMS that focuses on simplicity and ease of use. It allows developers to store and retrieve objects as if they were native data structures in programming languages, making it particularly suitable for embedded systems and lightweight applications.

4. GemStone/S: GemStone/S is a high-performance ODBMS designed for enterprise-level applications. It offers features like distributed object management, concurrency control, and support for running Smalltalk applications in a distributed environment.

5. OODBMS (Object-Oriented Database Management System): OODBMS is an open-source project that aims to create a high-quality, fully-featured, and scalable ODBMS. It provides persistence for Java objects and includes features like ACID transactions, query capabilities, and versioning.

6. Zope Object Database (ZODB): ZODB is an ODBMS specifically designed for the Zope web application framework in Python. It stores Python objects directly, allowing them to be retrieved with their state intact, which is useful for web applications and content management systems.

7. Perst: Perst is an open-source, object-oriented embedded database system for Java and .NET. It provides both object-oriented and relational access methods, making it suitable for a variety of application types.

8. Objectivity/DB: Objectivity/DB is designed for handling large-scale, complex datasets. It's known for its support for complex data models and advanced querying capabilities.

9. Neo4j: While Neo4j is often categorized as a graph database, it also incorporates object-oriented concepts, as it represents data in the form of nodes, relationships, and properties. It's widely used for graph-based data representation and querying.

10. OrientDB OrientDB is another example that combines elements of graph databases and object-oriented databases. It offers support for graph data models, document data models, and object-oriented data models, making it versatile for various application scenarios.

These are just a few examples of Object-Oriented Database Management Systems that have been developed to provide more intuitive and efficient ways to handle complex data structures and relationships in applications that require object-oriented modeling.

EXTRA

A reliable DDBMS is one that can continue to process user requests even when the underlying system is unreliable, i.e., failures occur

• Failures

– Transaction failures

– System (site) failures, e.g., system crash, power supply failure

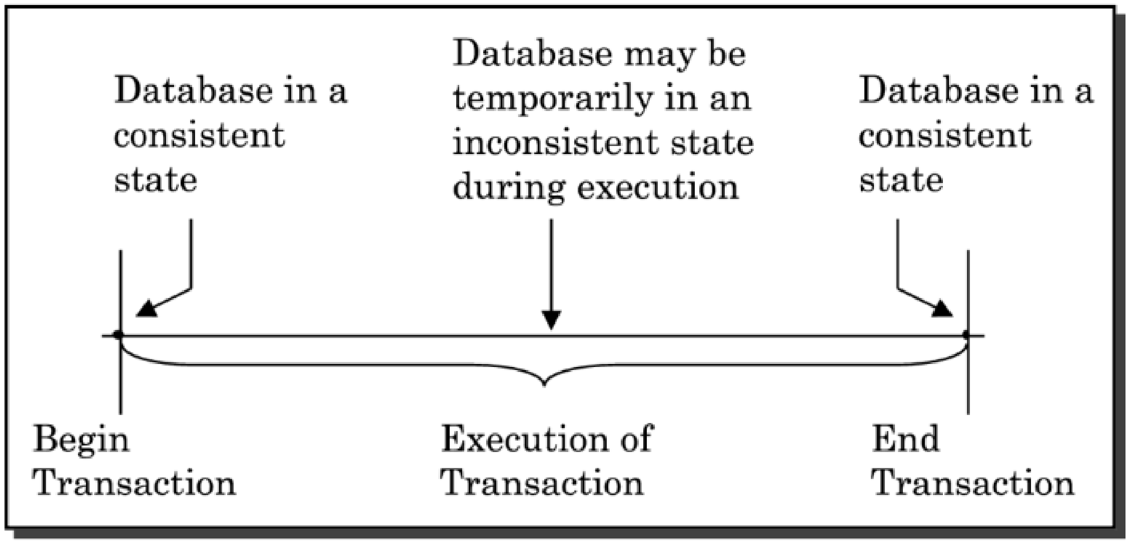
– Media failures, e.g., hard disk failures

– Communication failures, e.g., lost/undeliverable messages

• Reliability is closely related to the problem of how to maintain the atomicity and durability properties of transactions

Transaction:

A collection of actions that transforms the DB from one consistent state
into another consistent state; during the exectuion the DB might be inconsistent.


Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction

States of a transaction

– Active: Initial state and during the execution

– Paritally committed: After the final statement has been executed

– Committed: After successful completion
– Failed: After the discovery that normal execution can no longer proceed
– Aborted: After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.
Example: Consider an SQL query for increasing by 10% the budget of the CAD/CAM project. This query can be specified as a transaction by providing a name for the transaction and inserting a begin and end tag.

```
Transaction BUDGET_UPDATE
begin
EXEC SQL
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = "CAD/CAM"
end.
```

Properties of Transactions
The ACID properties
–
Atomicity
∗ A transaction is treated as a single/atomic unit of operation and is either executed completely or not at all
–
Consistency
∗ A transaction preserves DB consistency, i.e., does not violate any integrity constraints
– Isolation
∗ A transaction is executed as if it would be the only one.
–
Durability
∗ The updates of a committed transaction are permanent in the Db