College of Computer Science and Mathematics
Computer Science Department
Third Class

DATABASE

Part (2)

References :

1. Principles of Distributed Database Systems, M. Tamer Özsu and Patrick Valduriez, , 3rd Edition, Springer Science+Business Media, LLC 2015.

1. Introduction:

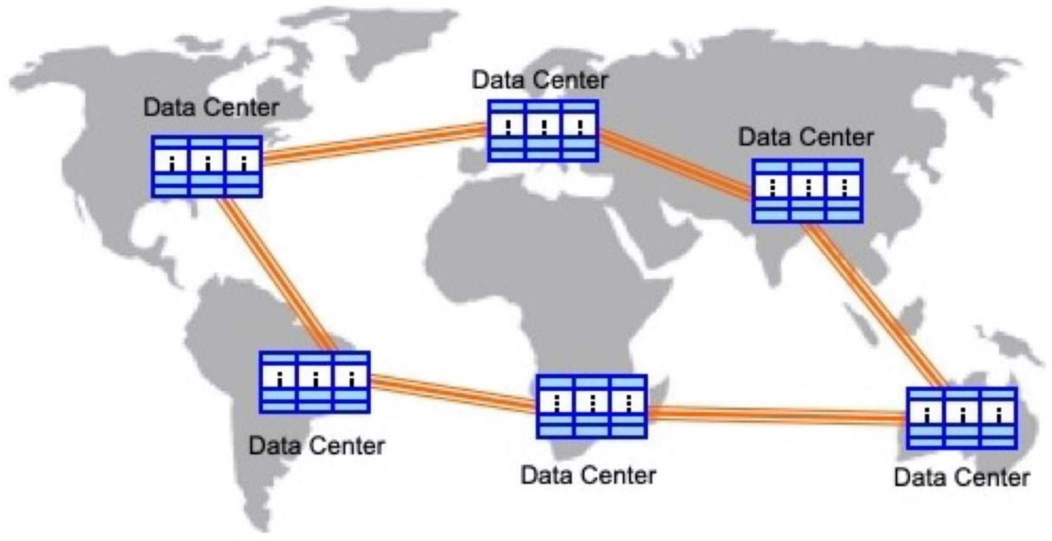
Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: database system and computer network technologies. Database systems have taken us from a paradigm of data processing in which each application defined and maintained its own data to one in which the data are defined and administered. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the data,and vice versa.

1.1. Distributed Computing

A number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks. Where the following are being distributed

1. Processing logic
2. Function
3. Data
4. Control

1.1.1 Current Distribution – Geographically Distributed Data Centers :



1.1.2 Distributed Database System

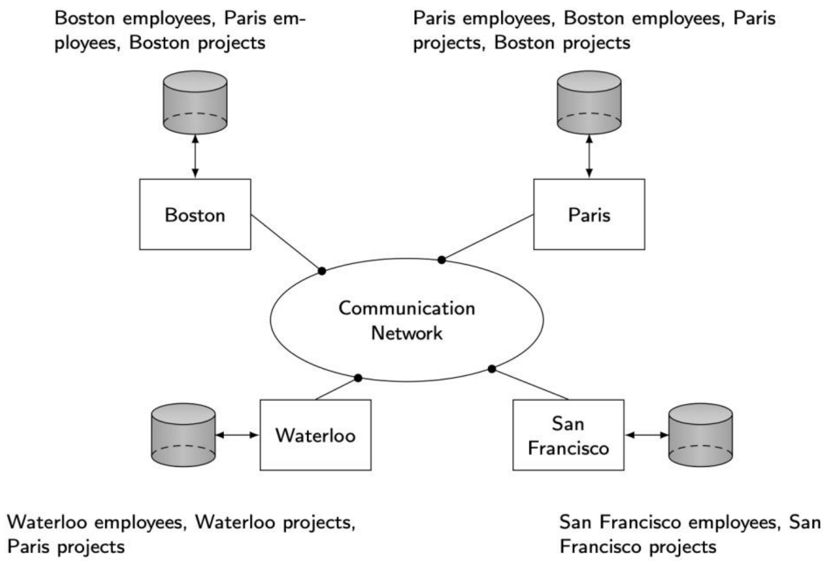
A **distributed database** is a collection of multiple, logically interrelated databases distributed over a computer network

A distributed database management system (Distributed DBMS) is the software that manages the DDB and **provides an access mechanism that makes this distribution transparent to the users.**

The following are not DDBS:

1. A timesharing computer system
2. A loosely or tightly coupled multiprocessor system
3. A database system which resides at one of the nodes of a network of computers - this is a centralized database on a network node

1.1.3 Distributed DBMS Environment:


Boston employees, Paris employees, Boston projects
Paris employees, Boston employees, Paris projects, Boston projects
Boston
Paris
Communication Network
Waterloo
San Francisco
Waterloo employees, Waterloo projects, Paris projects
San Francisco employees, San Francisco projects

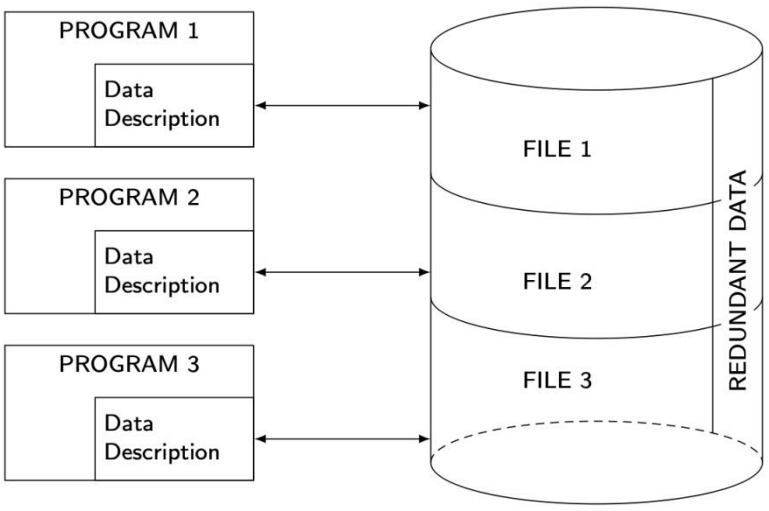
1.1.4 The Implicit Assumptions:

1. **Data stored at a number of sites → each site logically consists of a single processor**
2. **Processors at different sites are interconnected by a computer network → not a multiprocessor system (Parallel database systems)**
3. **Distributed database is a database, not a collection of files → data logically related as exhibited in the users' access patterns(Relational data model)**
4. **Distributed DBMS is a full-fledged DBMS (Not remote file system, not a TP system)**

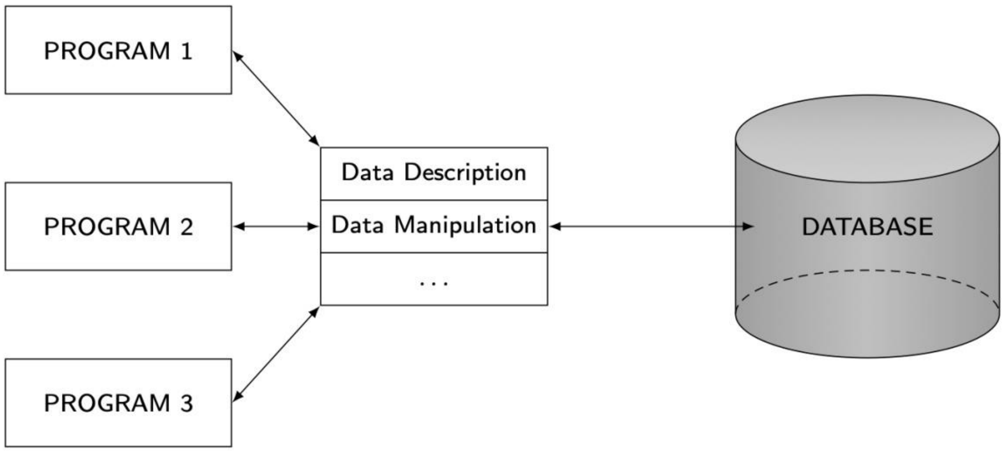
Important point: DDMS are Logically integrated but Physically distributed

1.2. Database History: database management evolved over history :

1.2.1 File Systems

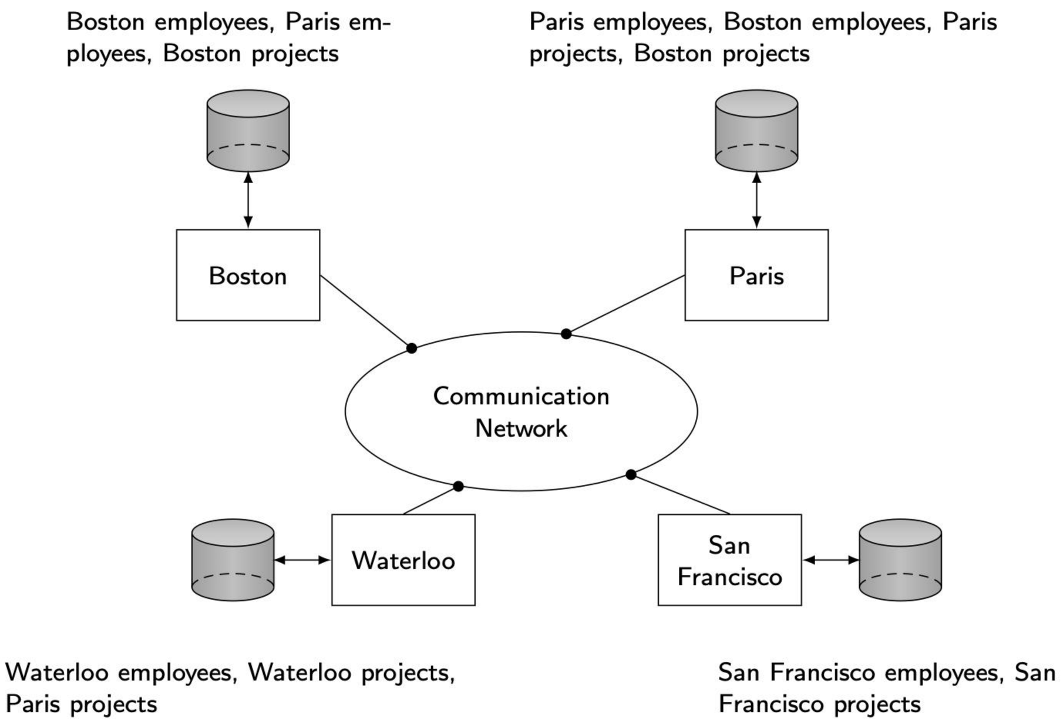


1.2.2 Database Management

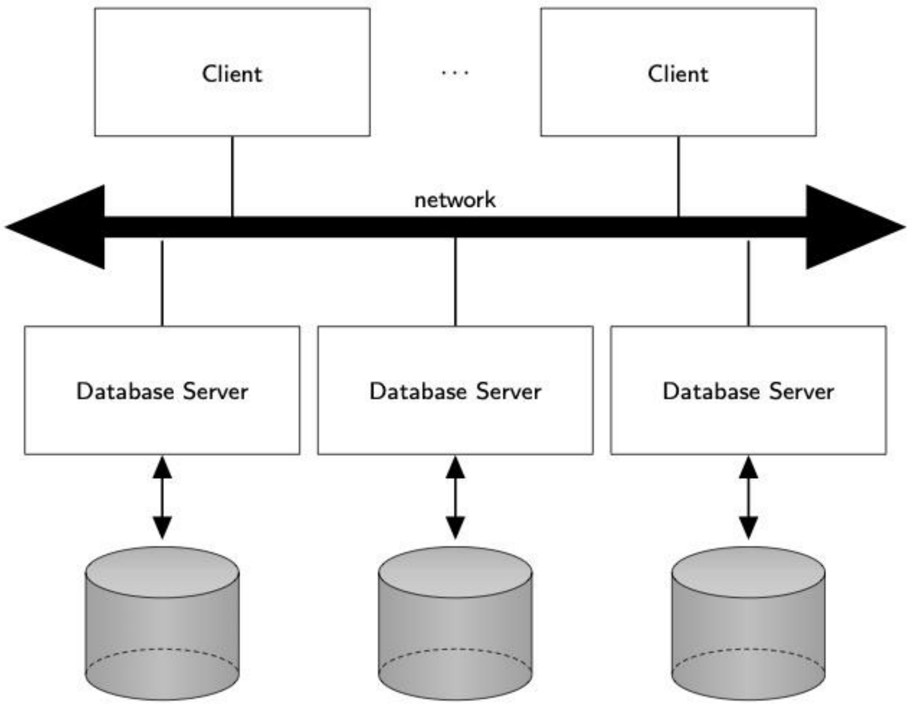


1.2.3 Early Distribution (Concepts)

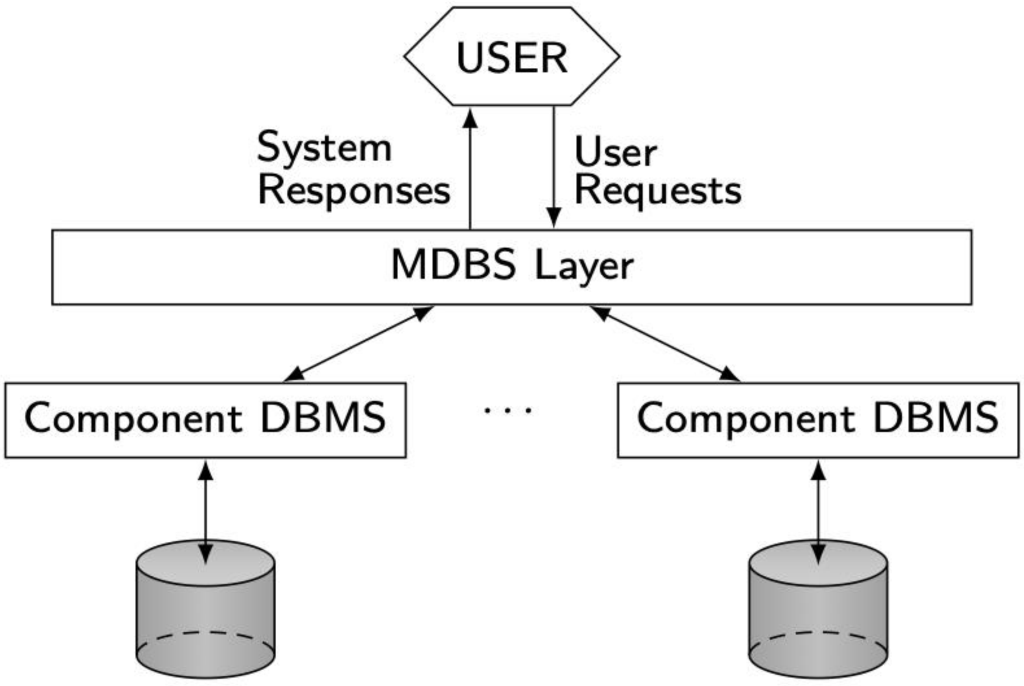
1. **Peer-to-Peer (P2P)**


Boston employees, Paris employees, Boston projects
Paris employees, Boston employees, Paris projects, Boston projects
Boston
Paris
Communication Network
Waterloo
San Francisco
Waterloo employees, Waterloo projects, Paris projects
San Francisco employees, San Francisco projects

2. **Client/Server**

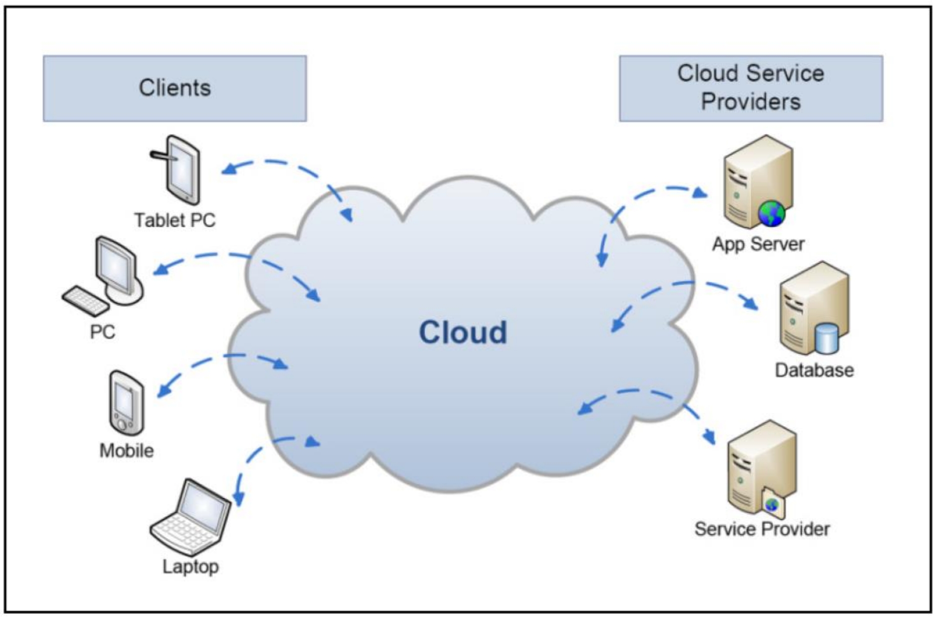


3. Data Integration



4. Cloud Computing

- On-demand, reliable services provided over the Internet in a cost-efficient manner
- Cost savings: no need to maintain dedicated compute power
- Elasticity: better adaptivity to changing workload



1.2.4 Data Delivery Alternatives

- Delivery modes
  1- Pull-only
  2- Push-only
  3- Hybrid
- Frequency
  1- Periodic
  2- Conditional
  3- Ad-hoc or irregular
- Communication Methods
  1- Unicast
  2- One-to-many

Note: not all combinations make sense

1.3. Distributed DBMS Promises

DBMS promise to offer the following :

1- Transparent management of distributed, fragmented, and replicated data
2- Improved reliability/availability through distributed transactions
3- Improved performance
4- Easier and more economical system expansion

1.3.1 Transparency

Transparency is the separation of the higher-level semantics of a system from the lower level implementation issues.

Fundamental issue is to provide data independence in the distributed environment which has :

1- Network (distribution) transparency
2- Replication transparency
3- Fragmentation transparency, which includes:
   - horizontal fragmentation: selection
   - vertical fragmentation: projection
   - hybrid

Example on Transparency :

\ 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.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

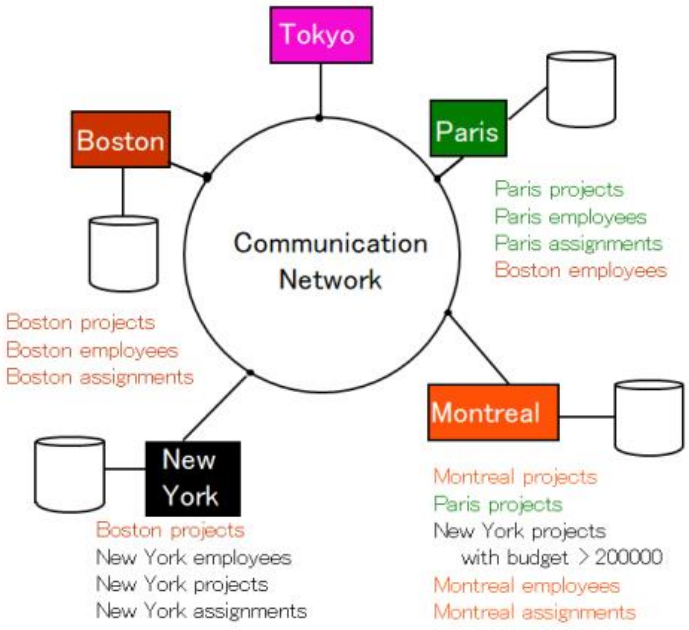
PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

1.3.2 Transparent Access

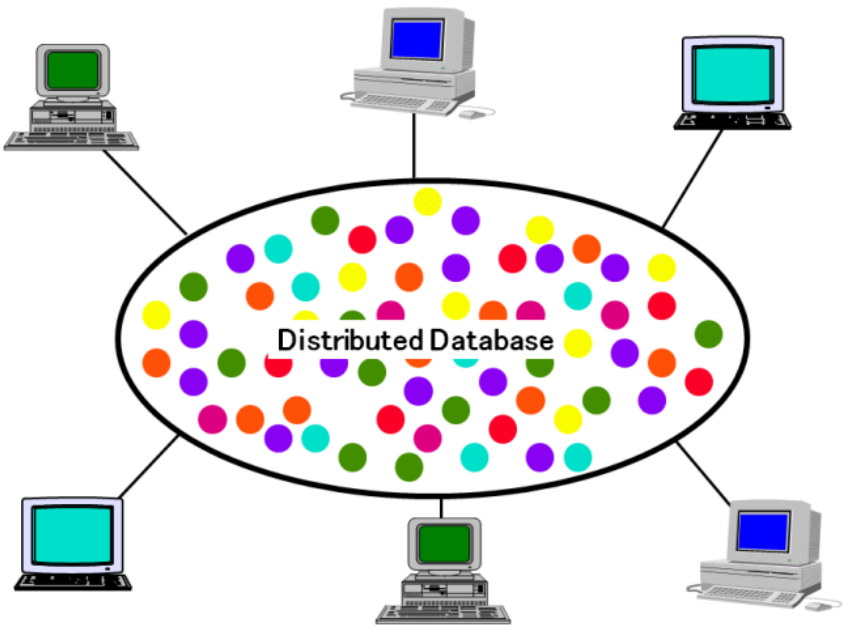
SELECT ENAME,SAL
FROM EMP,ASG,PAY
WHERE DUR > 12
AND EMP.ENO = ASG.ENO
AND PAY.TITLE = EMP.TITLE



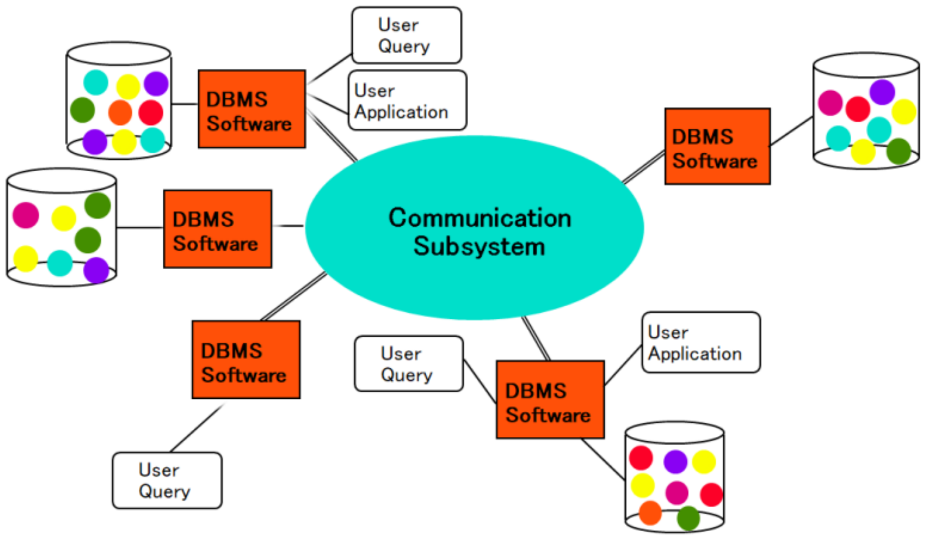
1.3.3 Distributed Database Views :

DDS might be viewed as :

1. User View



2. Real View



1.3.4 Types of Transparency

1- Data independence
2- Network transparency (or distribution transparency)
3- Fragmentation transparency
4- Replication transparency

1.3.5 Reliability Through Transactions

Replicated components and data should make distributed DBMS more reliable.

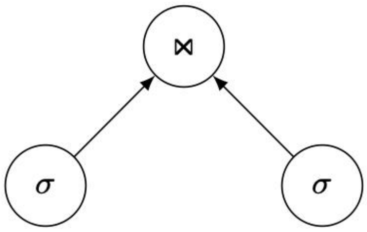
- Distributed transactions provide
    1- Concurrency transparency
    2- Failure atomicity
    3- Distributed transaction support requires implementation of
    4- Distributed concurrency control protocols
    5- Commit protocols
- Data replication
    1- Great for read-intensive workloads, problematic for updates
    2- Replication protocols

1.3.6 Potentially Improved Performance

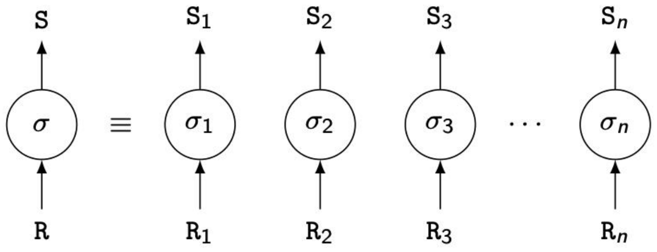
There is proximity of data to its points of use, it requires some support for fragmentation and replication

It offers Parallelism in execution, it has two types :

- ❑ Inter-query parallelism



- ❑ Intra-query parallelism



1.3.7 Scalability

The Issue is database scaling and workload scaling. result in adding processing and storage power. There are two types of scaling :

1- Scale-out: add more servers
2- Scale-up: increase the capacity of one server → has limits

1.4. Distributed DBMS Issues

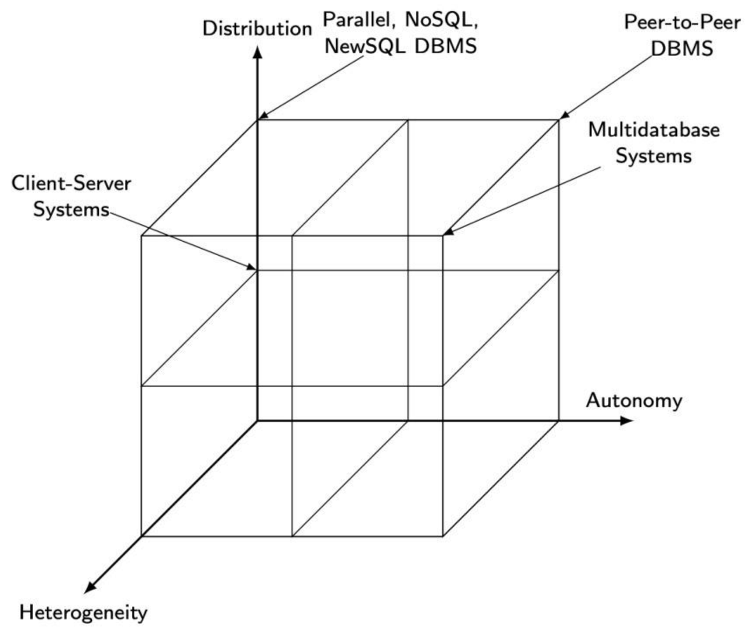
1. **Distributed database design**
   - **a.** How to distribute the database
   - **b.** Replicated & non-replicated database distribution
   - **c.** A related problem in directory management
2. **Distributed query processing**
   - **a.** Convert user transactions to data manipulation instructions
   - **b.** Optimization problem
     min{cost = data transmission + local processing}, where general formulation is NP-hard
3. **Distributed concurrency control**
   - **a.** Synchronization of concurrent accesses
   - **b.** Consistency and isolation of transactions' effects
   - **c.** Deadlock management
4. **Reliability**
   - **a.** How to make the system resilient to failures
   - **b.** Atomicity and durability
5. **Replication**
   - **a.** Mutual consistency
   - **b.** Freshness of copies
   - **c.** Eager vs lazy
   - **d.** Centralized vs distributed
6. **Parallel DBMS**
   - **a.** Objectives: high scalability and performance
   - **b.** Not geo-distributed
   - **c.** Cluster computing

1.4.1 Related Issues

1. Alternative distribution approaches
   - **a.** Modern P2P
   - **b.** World Wide Web (WWW or Web)
2. Big data processing
   - **a.** 4V: volume, variety, velocity, veracity
   - **b.** MapReduce & Spark
   - **c.** Stream data
   - **d.** Graph analytics
   - **e.** NoSQL
   - **f.** NewSQL
   - **g.** Polystores

1.5. Distributed DBMS architecture

1.5.1. DBMS Implementation Alternatives



1.5.2. Dimensions of the Problem in Designing DDMSs

1. Distribution

Whether the components of the system are located on the same machine or not

2. Heterogeneity

- Various levels (hardware, communications, operating system)
- DBMS important ones are : data model, query language,transaction management algorithms

3. Autonomy

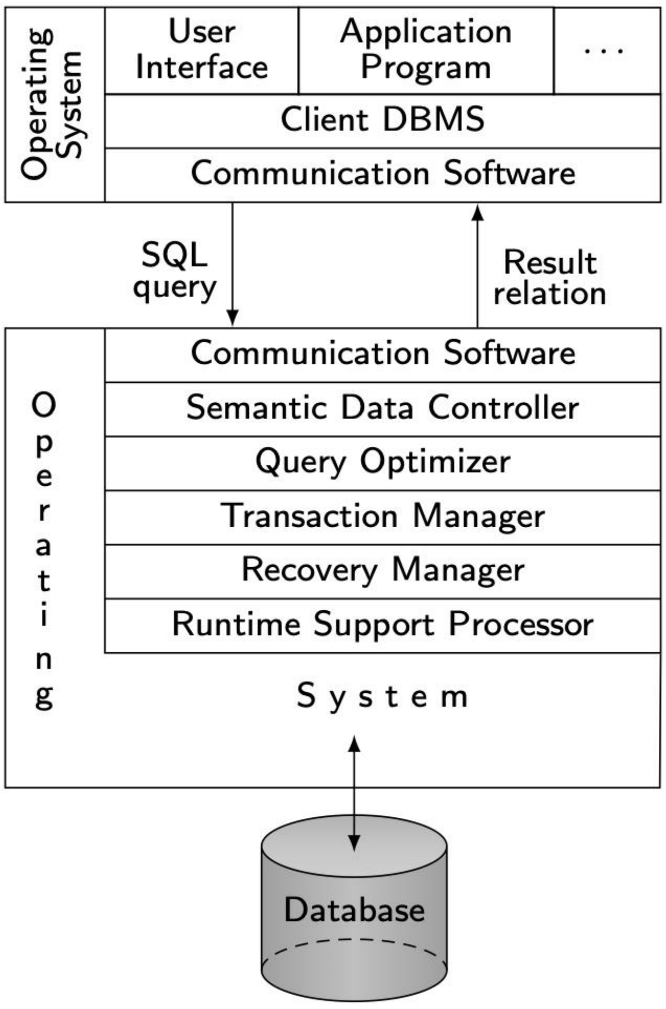
It is not well understood and most troublesome, it has various versions:

a) Design autonomy: Ability of a component DBMS to decide on issues related to its own design.

b) Communication autonomy: Ability of a component DBMS to decide whether and how to communicate with other DBMSs.

c) Execution autonomy: Ability of a component DBMS to execute local operations in any manner it wants to.

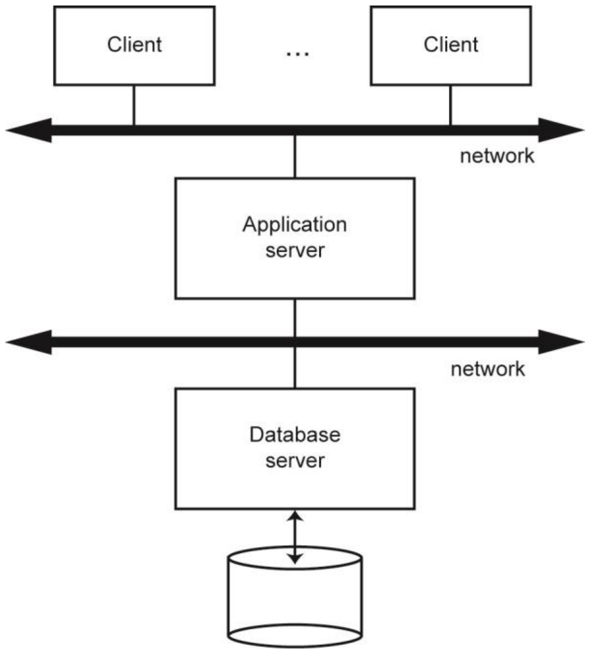
1.5.3. Client/Server Architecture



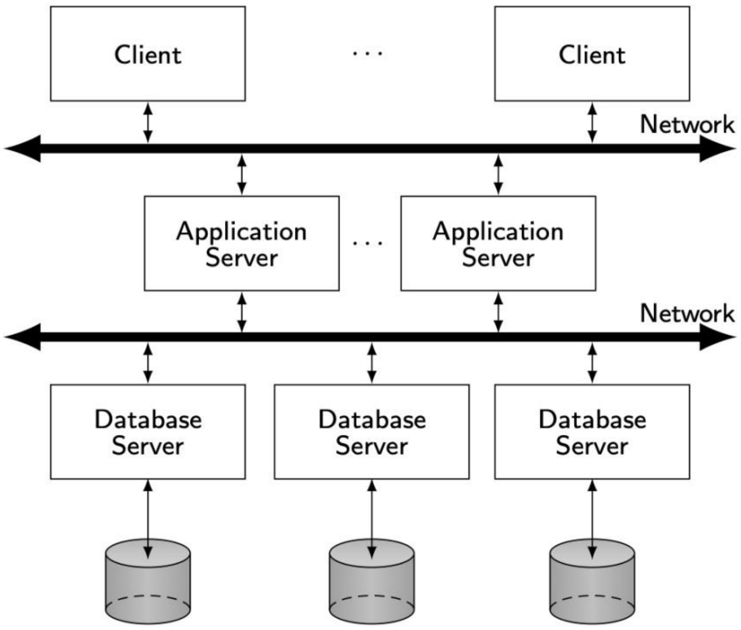
1.5.4. Advantages of Client-Server Architectures

1- More efficient division of labor
2- Horizontal and vertical scaling of resources
3- Better price/performance on client machines
4- Ability to use familiar tools on client machines
5- Client access to remote data (via standards)
6- Full DBMS functionality provided to client workstations
7- Overall better system price/performance

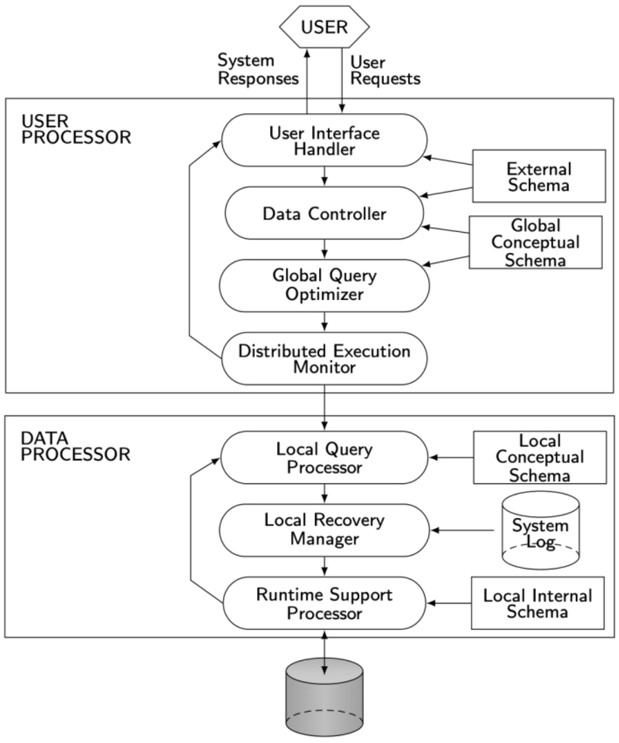
1.5.5. (Centralized) Database Server



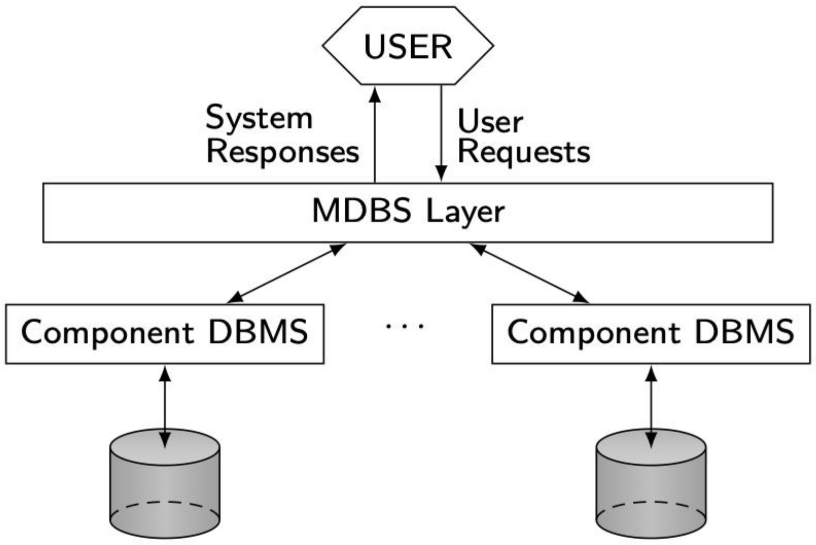
1.5.6. Distributed Database Servers



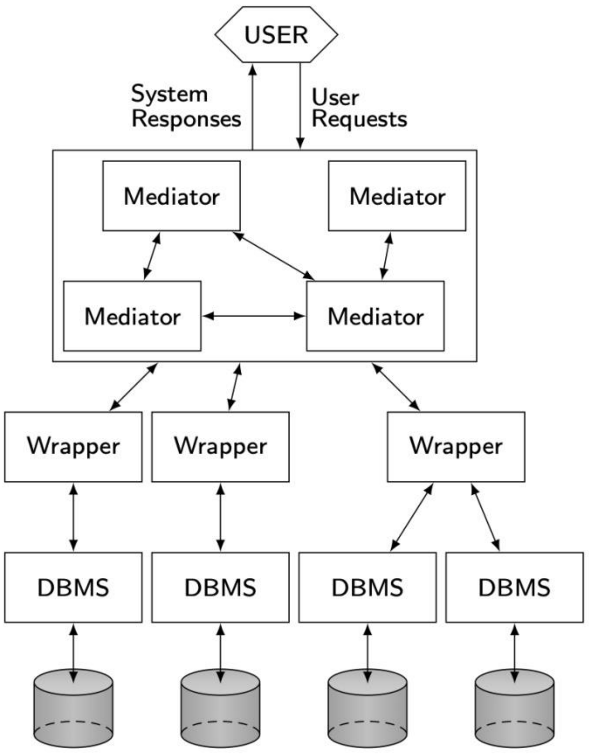
1.5.7. Peer-to-Peer Component Architecture



1.5.8. MDBS Components & Execution



1.5.9. Mediator/Wrapper Architecture

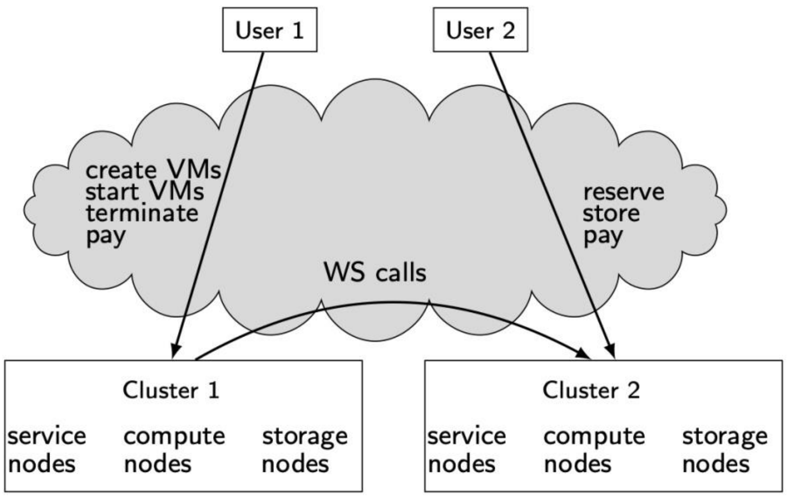


1.5.10. Cloud Computing

It is On-demand, reliable services provided over the Internet in a cost-efficient manner, it proved (but not limited to) the following :

1- IaaS – Infrastructure-as-a-Service
2- PaaS – Platform-as-a-Service
3- SaaS – Software-as-a-Service
4- DaaS – Database-as-a-Service

Simplified Cloud Architecture



2. Distributed and Parallel Database Design

2.1. Distribution Design

The design of a distributed computer system involves making decisions on the placement of data and programs across the sites of a computer network, as well as possibly designing the network itself.

In the case of distributed DBMSs, the distribution of applications involves two things:

1. The distribution of the distributed DBMS software
2. The distribution of the application programs that run on it.

2.1.1 Organization

The organization of distributed systems can be investigated along three orthogonal dimensions

1. Level of sharing
2. Behavior of access patterns
3. Level of knowledge on access pattern behavior

2.1.2 Levels of sharing

In terms of the level of sharing, there are three possibilities:

1. No sharing: each application and its data execute at one site, and there is no communication with any other program or access to any data file at other sites .

This characterizes the very early days of networking and is probably not very common today .

2. Level of data sharing; all the programs are replicated at all the sites, but data files are not . Accordingly, user requests are handled at the site where they originate, and the necessary data files are moved around the network .
3. Finally, in data-plus-program sharing, both data and programs may be shared, meaning that a program at a given site can request a service from another program at a second site, which, in turn, may have to access a data file located at a third site.

Along the second dimension of access pattern behavior, it is possible to identify two alternatives.

1. The access patterns of user requests may be static, so that they do not change over time, or dynamic. It is obviously considerably easier to plan for and manage the static environments than would be the case for dynamic distributed systems.

Unfortunately, it is difficult to find many real-life distributed applications that would be classified as static.

2. The third dimension of classification is the level of knowledge about the access pattern behavior. One possibility, is that the designers do not have any information about how users will access the database.

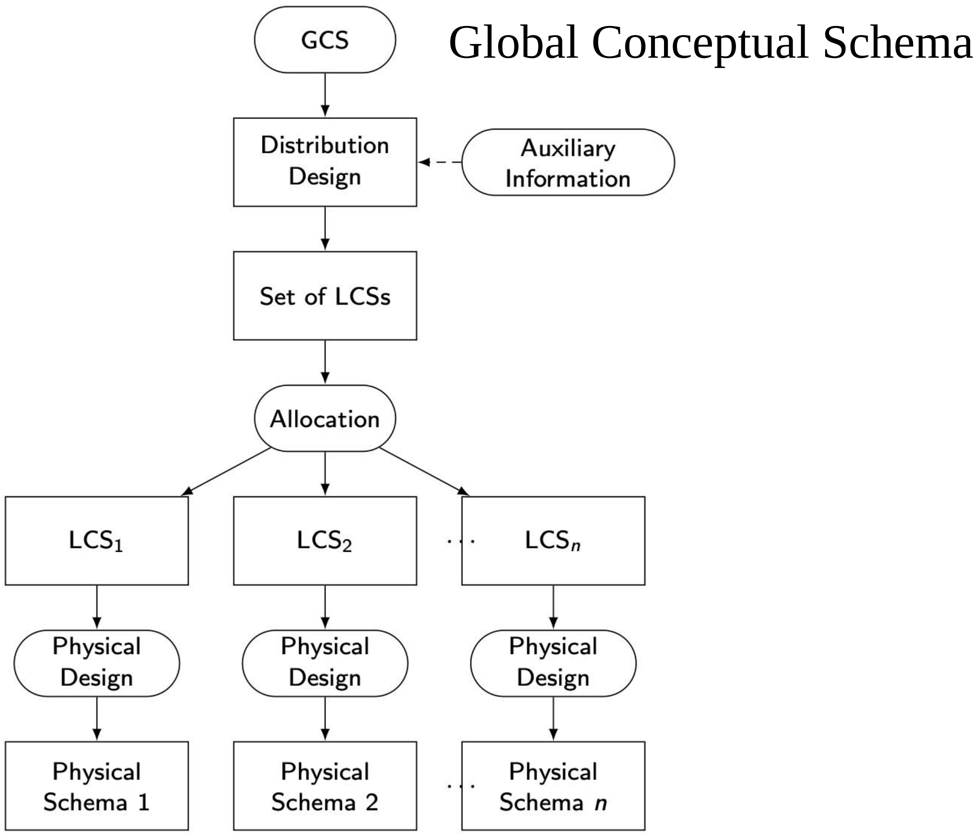
2.1.3 Design Strategies :

Two major strategies that have been identified for **designing distributed databases** are the top-down approach and the bottom-up approach .

As the names indicate, they constitute very different approaches to the design process.

Top-down approach is more suitable for tightly integrated, homogeneous distributed DBMSs, while bottom-up design is more suited to multidatabases

2.1.4 Design Schemas :



The global conceptual schema (GCS) and access pattern information collected as a result of view design are inputs to the *distribution design* step.
The objective at this stage, is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system.
It is possible, to treat each entity as a unit of distribution. Given that we use the relational model as the basis of discussion in this book, the entities correspond to relations.
Rather than distributing relations, it is quite common to divide them into subrelations, called *fragments*, which are then distributed.
Thus, the distribution design activity consists of two steps: *fragmentation* and *allocation*.
The reason for separating the distribution design into two steps is to better deal with the complexity of the problem.

2.1.5 Physical Design

The last step in the design process is the physical design, which maps the local conceptual schemas to the physical storage devices available at the corresponding sites.
The inputs to this process are the local conceptual schema and the access pattern information about the fragments in them.
It is well known that design and development activity of any kind is an ongoing process requiring constant monitoring and periodic adjustment and tuning.
We have therefore included observation and monitoring as a major activity in this process.
Note that one does not monitor only the behavior of the database implementation but also the suitability of user views. The result is some form of feedback, which may result in backing up to one of the earlier steps in the design.

2.2. Fragmentation

- The question is : Can't we just distribute relations?

The reasonable units of distribution are :

1. relation
   - views are subsets of relations ➔ locality
   - extra communication
2. fragments of relations (sub-relations)
   - concurrent execution of a number of transactions that access different portions of a relation
   - views that cannot be defined on a single fragment will require extra processing
   - semantic data control (especially integrity enforcement) more difficult

2.2.1. Example Database

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.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

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

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

2.2.2. Fragmentation Alternatives – Horizontal

Fragment the relation PROJ into two relations horizontally

$PROJ_1$:	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

$PROJ_1$

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

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

2.2.3. Fragmentation Alternatives – Vertical

Fragment the relation PROJ into two relations vertically

$PROJ_1$:information about project budgets
$PROJ_2$:information about project names and locations

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

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

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

2.2.4. Correctness of Fragmentation

- Completeness
    - Decomposition of relation R into fragments $R_1, R_2, ..., R_n$ is complete if and only if each data item in R can also be found in some R_i
- Reconstruction
    - If relation R is decomposed into fragments $R_1, R_2, ..., R_n$, then there should exist some relational operator ∇ such that

$R = \nabla_{1 \le i \le n} R_i$

- Disjointness
    - If relation R is decomposed into fragments $R_1, R_2, ..., R_n$, and data item d_i is in R_j, then d_i should not be in any other fragment R_k ($k \neq j$).

2.2.5. Allocation Alternatives

1. Non-replicated
    - partitioned : each fragment resides at only one site
2. Replicated
    - fully replicated : each fragment at each site
    - partially replicated : each fragment at some of the sites
3. Rule of thumb:

If $\frac{\text{read-only queries}}{\text{update queries}} \ll 1$, replication is advantageous, otherwise replication may cause problems

2.2.6. Comparison of Replication Alternatives

	Full replication	Partial replication	Partitioning
QUERY PROCESSING	Easy	Same difficulty	
DIRECTORY MANAGEMENT	Easy or nonexistent	Same difficulty	
CONCURRENCY CONTROL	Moderate	Difficult	Easy
RELIABILITY	Very high	High	Low
REALITY	Possible application	Realistic	Possible application

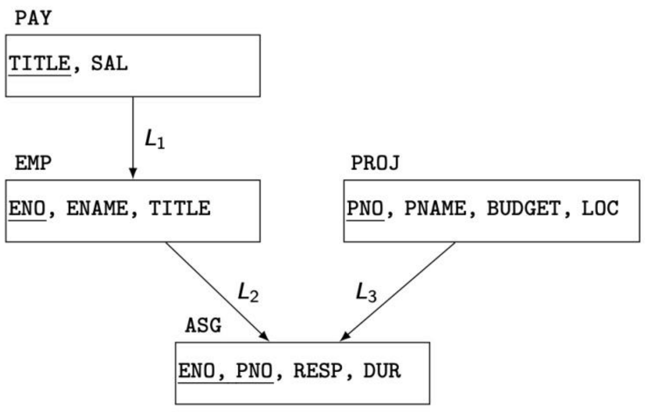
2.3. Fragmentation

1. Horizontal Fragmentation (HF)
   a) Primary Horizontal Fragmentation (PHF)
   b) Derived Horizontal Fragmentation (DHF)
2. Vertical Fragmentation (VF)
3. Hybrid Fragmentation (HF)

2.3.1. Horizontal Fragmentation (HF)

A. PHF – Information Requirements

- Database Information



- ❑ Relationship
- ❑ cardinality of each relation: *card*(*R*)

B. Application Information

- ❑ **minterm selectivitie**s: *sel*(m_i)
  - The number of tuples of the relation that would be accessed by a user query which is specified according to a given minterm predicate m_i.
- ❑ **access frequencies**: *acc*(q_i)
  - The frequency with which a user application *qi* accesses data.
  - Access frequency for a minterm predicate can also be defined.

- **Primary Horizontal Fragmentation**

Definition :

$R_j = \sigma_{F_j}(R),\ 1 \le j \le w$

where F_j is a selection formula, which is (preferably) a minterm predicate.

Therefore,

A horizontal fragment R_i of relation *R* consists of all the tuples of *R* which satisfy a minterm predicate m_i.

$\Downarrow$

Given a set of minterm predicates *M*, there are as many horizontal fragments of relation *R* as there are minterm predicates.

Set of horizontal fragments also referred to as minterm fragments.

C. PHF – Algorithm

Given: A relation *R*, the set of simple predicates *Pr*

Output: The set of fragments of $R = \{R_1, R_2, \ldots, R_w\}$ which obey the fragmentation rules.

Preliminaries :

- ❑ *Pr* should be *complete*
- ❑ *Pr* should be *minimal*

D. Completeness of Simple Predicates

A set of simple predicates *Pr* is said to be *complete* if and only if the accesses to the tuples of the minterm fragments defined on *Pr* requires that two tuples of the same minterm fragment have the same probability of being accessed by any application.

- **Example :**

Assume PROJ[PNO,PNAME,BUDGET,LOC] has two applications defined on it.

- ❑ Find the budgets of projects at each location. (1)
- ❑ Find projects with budgets less than $200000. (2)

- **PHF – Example**

- Fragmentation of relation PROJ
  - ❑ Applications:
    - Find the name and budget of projects given their no.
      - ❑ Issued at three sites
    - Access project information according to budget
      - ❑ one site accesses ≤200000 other accesses >200000
  - ❑ Simple predicates
  - ❑ For application (1)

p_1 : LOC = “Montreal”

p_2 : LOC = “New York”

p_3 : LOC = “Paris”

- ❑ For application (2)

p_4 : BUDGET ≤ 200000

p_5 : BUDGET > 200000

- ❑ $Pr = Pr' = \{p_1, p_2, p_3, p_4, p_5\}$
- ❑ Minterm fragments left after elimination

m_1 : (LOC = “Montreal”) $\wedge$ (BUDGET ≤ 200000)

m_2 : (LOC = “Montreal”) $\wedge$ (BUDGET > 200000)

m_3 : (LOC = “New York”) $\wedge$ (BUDGET ≤ 200000)

m_4 : (LOC = “New York”) $\wedge$ (BUDGET > 200000)

m_5 : (LOC = “Paris”) $\wedge$ (BUDGET ≤ 200000)

m_6 : (LOC = "Paris") ∧ (BUDGET > 200000)

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$PROJ_3$

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York

$PROJ_4$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York

$PROJ_6$

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

E. PHF – Correctness

- Completeness
  - ❑ Since *Pr*' is complete and minimal, the selection predicates are complete
- Reconstruction
  - ❑ If relation *R* is fragmented into $F_R = \{R_1, R_2, \ldots, R_r\}$

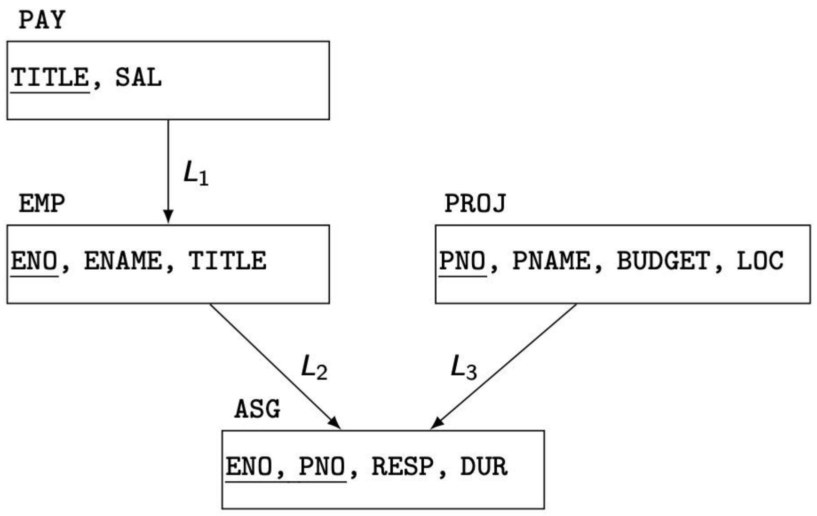
$R = \cup_{\forall R_i \in FR} R_i$

- Disjointness
  - ❑ Minterm predicates that form the basis of fragmentation should be mutually exclusive.

2.3.2. Derived Horizontal Fragmentation

Defined on a member relation of a link according to a selection operation specified on its owner.

- ❑ Each link is an equijoin.
- ❑ Equijoin can be implemented by means of semijoins.



A. DHF – Definition

Given a link L where *owner*(L)=S and *member*(L)=R, the derived horizontal fragments of R are defined as

$$R_i = R \ltimes_F S_i,\ 1 \le i \le w$$

where w is the maximum number of fragments that will be defined on R and

$$S_i = \sigma_{F_i}(S)$$

where F_i is the formula according to which the primary horizontal fragment S_i is defined.

B. DHF – Example

Given link L_1 where owner(L_1)=SKILL and member(L_1)=EMP

$EMP_1 = EMP \ltimes SKILL_1$

$EMP_2 = EMP \ltimes SKILL_2$

where

$SKILL_1 = \sigma_{SAL \le 30000}(SKILL)$

$SKILL_2 = \sigma_{SAL > 30000}(SKILL)$

ASG_1

ENO	PNO	RESP	DUR
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E7	P3	Engineer	36

ASG_2

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E5	P2	Manager	24
E6	P4	Manager	48
E8	P3	Manager	40

C. DHF – Correctness

- Completeness
  - Referential integrity
  - Let R be the member relation of a link whose owner is relation S which is fragmented as $F_S = \{S_1, S_2, ..., S_n\}$. Furthermore, let A be the join attribute between R and S. Then, for each tuple t of R, there should be a tuple t' of S such that

  $t[A] = t'[A]$
- Reconstruction
  - Same as primary horizontal fragmentation.
- Disjointness
  - Simple join graphs between the owner and the member fragments.

2.3.3. Vertical Fragmentation (VF)

- Has been studied within the centralized context, in terms of :
  - ❑ design methodology
  - ❑ physical clustering

Vertical fragmentation is more difficult than horizontal, because more alternatives exist. It works in two approaches :

1. Grouping:
   - attributes to fragments
2. Splitting
   - relation to fragments

Vertical fragmentation has two types :

1. Overlapping fragments
   - ❑ grouping
2. Non-overlapping fragments
   - ❑ splitting

We do not consider the replicated key attributes to be overlapping.

Advantage: VF is Easier to enforce functional dependencies for integrity checking etc.)

D. VF – Information Requirements

- Application Information
  - ❑ Attribute affinities
    - a measure that indicates how closely related the attributes are
    - This is obtained from more primitive usage data
  - ❑ Attribute usage values
    - Given a set of queries $Q = \{q_1, q_2,\ldots, q_q\}$ that will run on the relation $R[A_1, A_2,\ldots, A_n]$,

$$use(q_i,A_j) = \begin{cases} 1 \text{ if attribute } A_j \text{ is referenced by query } q_i \\ 0 \text{ otherwise} \end{cases}$$

$use(q_i,\bullet)$ can be defined accordingly

E. VF – Definition of use*(qi,Aj)*

Consider the following 4 queries for relation PROJ

```
q1:  SELECT   BUDGET          q2:  SELECT   PNAME,BUDGET
     FROM  PROJ                    FROM  PROJ
     WHERE    PNO=Value
q3:  SELECT   PNAME           q4:  SELECT   SUM(BUDGET)
     FROM  PROJ                    FROM  PROJ
     WHERE    LOC=Value            WHERE    LOC=Value
```

$$\begin{array}{c|cccc} & PNO & PNAME & BUDGET & LOC \\ \hline q_1 & 1 & 0 & 1 & 0 \\ q_2 & 0 & 1 & 1 & 0 \\ q_3 & 0 & 1 & 0 & 1 \\ q_4 & 0 & 0 & 1 & 1 \end{array}$$

F. VF – Algorithm

Vertical Fragmentation has two problems to solve:

1. Cluster forming in the middle of the Clustered Affinity Matrix.
    - ❑ Shift a row up and a column left and apply the algorithm to find the "best" partitioning point
    - ❑ Do this for all possible shifts
    - ❑ Cost $O(m^2)$
2. More than two clusters
    - ❑ *m*-way partitioning
    - ❑ try 1, 2, …, m–1 split points along diagonal and try to find the best point for each of these
    - ❑ Cost $O(2^m)$

G. VF – Correctness

A relation R, defined over attribute set A and key K, generates the vertical partitioning $F_R = \{R_1, R_2, \ldots, R_r\}$.

- ■ Completeness
    - ❑ The following should be true for A:

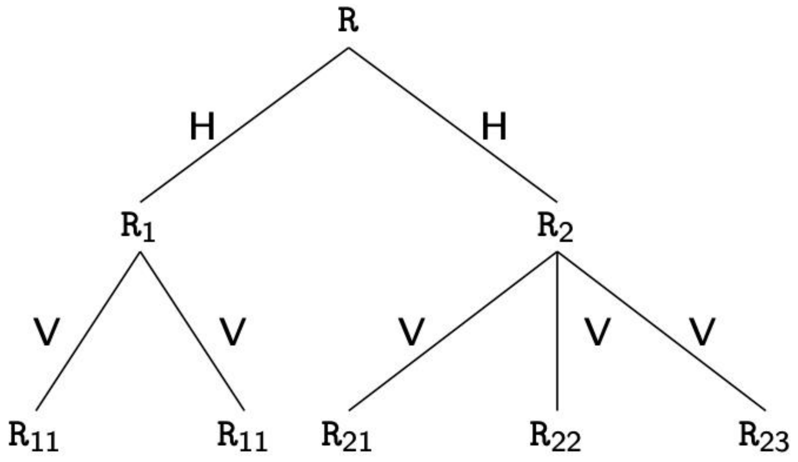
$A = \cup A_{Ri}$

- ■ Reconstruction
    - ❑ Reconstruction can be achieved by

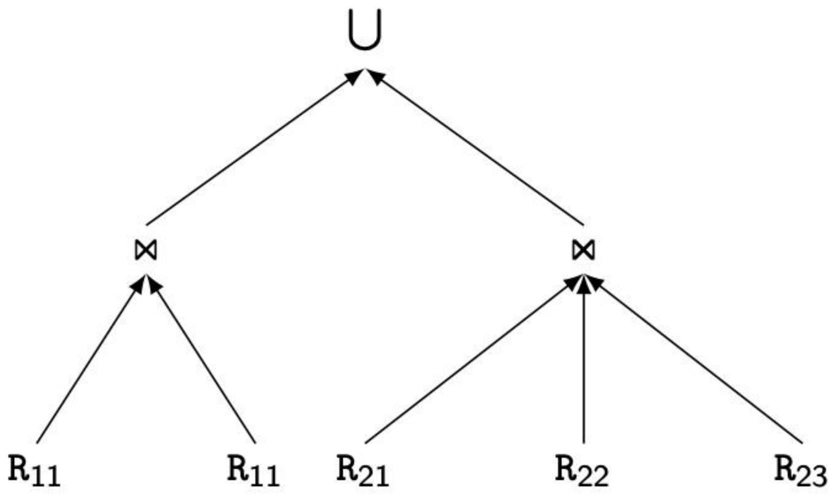
$R = \bowtie_K R_i, \forall R_i \in F_R$

- ■ Disjointness
    - ❑ Tuple Identifiers (TID's) are not considered to be overlapping since they are maintained by the system
    - ❑ Duplicated keys are not considered to be overlapping

2.3.4. Hybrid Fragmentation



Reconstruction of Hybrid Fragmentation



2.4. Distributed Database Systems Fragmentation Examples

2.4.1. Example 1: Horizontal Fragmentation

Problem Statement:

A company database contains the EMPLOYEE table with the following attributes:
EMP_ID (Primary Key)
NAME
DEPARTMENT
SALARY
LOCATION
The company wants to horizontally fragment the table based on employee salaries, where:
EMP_LOW: Employees with SALARY ≤ 50,000
EMP_HIGH: Employees with SALARY > 50,000
Design the horizontal fragmentation and verify its correctness.

Solution:

- Using **Primary Horizontal Fragmentation (PHF)**, we define selection predicates:
- **P1:** SALARY ≤ 50,000
- **P2:** SALARY > 50,000
- The fragmentation is defined as:
- **EMP_LOW = σ (SALARY ≤ 50,000) (EMPLOYEE)**
- **EMP_HIGH = σ (SALARY > 50,000) (EMPLOYEE)**

Correctness Checks:

Completeness: Every tuple in **EMPLOYEE** is included in either **EMP_LOW** or **EMP_HIGH**.
Reconstruction: The original table can be reconstructed using **UNION**:
EMPLOYEE=EMP_LOW∪EMP_HIGHEMPLOYEE =
Disjointness: No tuple appears in both fragments since **P1** and **P2** are mutually exclusive.

```
---------+----------+-------------+--------+-----------+
 EMP_ID | NAME     | DEPARTMENT | SALARY| LOCATION |
---------+----------+-------------+--------+-----------+
 101    | Alice   | HR          | 45000 | NY        |  → EMP_LOW
 102    | Bob     | IT          | 70000 | LA        |  → EMP_HIGH
 103    | Charlie | Finance     | 60000 | SF        |  → EMP_HIGH
 104    | David   | HR          | 48000 | TX        |  → EMP_LOW
---------+----------+-------------+--------+-----------+
```

2.4.2. Example 2: Vertical Fragmentation

Problem Statement

- A hospital database contains the **PATIENT** table with the following attributes:

PATIENT_ID (Primary Key)
NAME
AGE
DIAGNOSIS
TREATMENT
The database is accessed by two different applications:
Administrative System: Requires **PATIENT_ID, NAME, and AGE**.
Medical System: Requires **PATIENT_ID, DIAGNOSIS, and TREATMENT**.

Solution :

- Design a **Vertical Fragmentation** that optimizes access based on application needs.

Using **Vertical Fragmentation**, we define the following fragments:
PATIENT_ADMIN = π (PATIENT_ID, NAME, AGE) (PATIENT)
PATIENT_MEDICAL = π (PATIENT_ID, DIAGNOSIS, TREATMENT) (PATIENT)

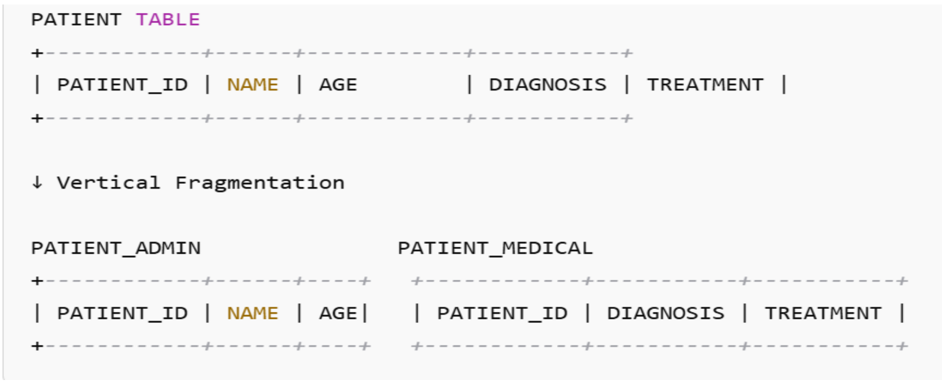
Correctness Checks:

Completeness:

The set of attributes in **PATIENT_ADMIN** and **PATIENT_MEDICAL** covers all attributes of **PATIENT**.

Reconstruction:

The original table can be reconstructed using **JOIN** on **PATIENT_ID**:
PATIENT=PATIENT_ADMIN⋈PATIENT



```
PATIENT TABLE
+------------+------+------------+-----------+
| PATIENT_ID | NAME | AGE        | DIAGNOSIS | TREATMENT |
+------------+------+------------+-----------+

↓ Vertical Fragmentation

PATIENT_ADMIN                PATIENT_MEDICAL
+------------+------+----+   +------------+-----------+-----------+
| PATIENT_ID | NAME | AGE|   | PATIENT_ID | DIAGNOSIS | TREATMENT |
+------------+------+----+   +------------+-----------+-----------+
```

Disjointness:

The **PATIENT_ID** is duplicated in both fragments to maintain integrity, but all other attributes are uniquely assigned.

3. Distributed Data Control:

3.1. Semantic Data Control (SDS)

The SDS Involves:

1. View management
2. Security control
3. Integrity control

The Objective of SDS is to ensure that authorized users perform correct operations on the database, contributing to the maintenance of the database integrity.

3.2. View Management

View – virtual relation

1. Generated from base relation(s) by a query
2. Not stored as base relations

Example :

```
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

Views can be manipulated as base relations

Example :

```
SELECT  ENAME, PNO, RESP
        FROM    SYSAN, ASG
        WHERE   SYSAN.ENO = ASG.ENO
```

3.2.1. Query Modification

Queries expressed on views

Queries expressed on base relations

Example :

```
SELECT    ENAME, PNO, RESP
          FROM      SYSAN, ASG
                    WHERE     SYSAN.ENO = ASG.ENO
SELECT ENAME,PNO,RESP
          FROM      EMP, ASG
          WHERE     EMP.ENO = ASG.ENO
                              AND   TITLE = "Syst. Anal."
```

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

3.2.2. View Updates

1. Updatable
```
CREATE VIEW    SYSAN(ENO,ENAME)
AS    SELECT   ENO,ENAME
      FROM     EMP
      WHERE    TITLE="Syst. Anal."
```
2. Non-updatable
```
CREATE VIEW    EG(ENAME,RESP)
AS    SELECT   ENAME,RESP
      FROM     EMP, ASG
      WHERE    EMP.ENO=ASG.ENO
```

3.2.3. View Management in Distributed DBMS

1. Views might be derived from fragments.
2. View definition storage should be treated as database storage
3. Query modification results in a distributed query
4. View evaluations might be costly if base relations are distributed

3.3. Materialized View

Static copy of the view, avoid view derivation for each query, but periodic recomputing of the view may be expensive.

Materialized view is an actual version of a view, is stored as a database relation, possibly with indices

Materialized view used much in practice:

a) DDBMS: No need to access remote, base relations
b) Data warehouse: to speed up OLAP

3.3.1. Materialized View Maintenance

1. Process of updating (refreshing) the view to reflect changes to base data
   - ❑ Resembles data replication but there are differences
     - ■ View expressions typically more complex
     - ■ Replication configurations more general
2. View maintenance policy to specify:
   - ❑ When to refresh
   - ❑ How to refresh

3.3.2. When to Refresh a View

A. Immediate mode
   - ❑ Run as part of the updating transaction, e.g. through 2PC
   - ❑ View always consistent with base data and fast queries
   - ❑ But increased transaction time to update base data

B. Deferred mode (preferred in practice)
   - ❑ Through separate refresh transactions
   - ❑ No penalty on the updating transactions

The deferred mode Triggered at different times with different trade-offs:

1. Lazily: just before evaluating a query on the view
2. Periodically: every hour, every day, etc.
3. Forcedly: after a number of predefined updates

3.4. Data security

3.4.1. Data Protection

- ❑ Prevents the physical content of data to be understood by unauthorized users
- ❑ Uses encryption/decryption techniques (Public key)

3.4.2. Access Control

1. Only authorized users perform operations they are allowed to on database objects
2. Discretionary access control (DAC)
   a. Long been provided by DBMS with authorization rules
1. Multilevel access control (MAC)
   b. Increases security with security levels

3.5. Discretionary Access Control (DAC)

3.5.1. DAC Main Actors

1- Subjects (users, groups of users) who execute operations
2- Operations (in queries or application programs)
3- Objects, on which operations are performed

Checking whether a subject may perform an op. on an object:

1- Authorization= (subject, op. type, object def.)
2- Defined using GRANT OR REVOKE
3- Centralized: one single user class (admin.) may grant or revoke
4- Decentralized, with op. type GRANT (More flexible but recursive revoking process which needs the hierarchy of grants)

3.5.2. Problem with DAC

A malicious user can access unauthorized data through an authorized user

Example

- ❑ User A has authorized access to *R* and *S*
- ❑ User B has authorized access to *S* only
- ❑ B somehow manages to modify an application program used by A so it writes *R* data in *S*
- ❑ Then B can read unauthorized data (in *S*) without violating authorization rules

Solution: multilevel security based on the famous Bell and Lapuda model for OS security

3.6. Multilevel Access Control (MAC)

- Different security levels (*clearances*)
  - ❑ *Top Secret > Secret > Confidential > Unclassified*
- Access controlled by 2 rules:
  - ❑ No read up
    - Subject *S* is allowed to read an object of level *L* only if $level(S) \geq L$
    - Protect data from unauthorized disclosure, e.g. a subject with secret clearance cannot read top secret data
  - ❑ No write down:
    - Subject *S* is allowed to write an object of level *L* only if $level(S) \leq L$
    - Protect data from unauthorized change, e.g. a subject with top secret clearance can only write top secret data but not secret data (which could then contain top secret data)

3.6.1. MAC in Relational DB

A relation can be classified at different levels:

1- Relation: all tuples have the same clearance
2- Tuple: every tuple has a clearance
3- Attribute: every attribute has a clearance

A classified relation is thus multilevel - Appears differently (with different data) to subjects with different clearances.

Example:
PROJ*: classified at attribute level

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	DB Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S

PROJ* as seen by a subject with confidential clearance

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	DB Develop.	C	Null	C	Null	C

3.7. Distributed Access Control (DAC)

3.7.1. Additional problems in a distributed environment

1. Remote user authentication:
   Typically using a directory service, it should be replicated at some sites for availability
2. Management of Discretionary (DAC) rules:
   Problem if users' group can span multiple sites:
   a) Rules stored at some directory based on user groups location
   b) Accessing rules may incur remote queries
3. Covert channels in MAC

3.7.2. Covert Channels:

Covers channels are : Indirect means to access unauthorized data
Example
- ❑ Consider a simple DDB with 2 sites: C (confidential) and S (secret)
- ❑ Following the "no write down" rule, an update from a subject with secret clearance can only be sent to S
- ❑ Following the "no read up" rule, a read query from the same subject can be sent to both C and S
- ❑ But the query may contain secret information (e.g. in a select predicate), so is a potential covert channel

Solution: replicate part of the DB/ So that a site at security level *L* contains all data that a subject at level *L* can access (e.g. S above would replicate the confidential data so it can entirely process secret queries)

3.8. Semantic Integrity Control (SIC)

Maintain database consistency by enforcing a set of constraints defined on the database.

A. Structural constraints
   - ❑ Basic semantic properties inherent to a data model e.g., unique key constraint in relational model

B. Behavioral constraints
   - ❑ Regulate application behavior, e.g., dependencies in the relational model

C. SIC has Two components
   - ❑ Integrity constraint specification
   - ❑ Integrity constraint enforcement

3.8.1. Semantic Integrity Control Types :

A. Procedural SIC :

- ❑ Control embedded in each application program

B. Declarative SIC :

- ❑ Assertions in predicate calculus
- ❑ Easy to define constraints
- ❑ Definition of database consistency clear

But Declarative SIC is inefficient to check assertions for each update

- ■ Limit the search space
- ■ Decrease the number of data accesses/assertion
- ■ Preventive strategies
- ■ Checking at compile time

3.9. Constraint Specification Language

3.9.1. Predefined constraints

Specify the more common constraints of the relational model

1- Not-null attribute

   ENO **NOT NULL IN** EMP

2- Unique key

   (ENO, PNO) **UNIQUE IN** ASG

3- Foreign key

   A key in a relation *R* is a foreign key if it is a primary key of another relation *S* and the existence of any of its values in *R* is dependent upon the existence of the same value in *S*

   PNO **IN** ASG **REFERENCES** PNO **IN** PROJ

4- Functional dependency

   ENO **IN** EMP **DETERMINES** ENAME

3.9.2. Precompiled Constraints

Express preconditions that must be satisfied by all tuples in a relation for a given update type (INSERT, DELETE, MODIFY)

NEW - ranges over new tuples to be inserted

OLD - ranges over old tuples to be deleted

General Form

CHECK ON <relation> [**WHEN** <update type>] <qualification>

- **Domain constraint**

CHECK ON PROJ (BUDGET≥500000 **AND** BUDGET≤1000000)

- **Domain constraint on deletion**

CHECK ON PROJ **WHEN DELETE** (BUDGET = 0)

- **Transition constraint**

CHECK ON PROJ (**NEW**.BUDGET > **OLD**.BUDGET **AND NEW**.PNO = **OLD**.PNO)

A. General constraints

Constraints that must always be true. Formulae of tuple relational calculus where all variables are quantified.

General Form:

CHECK ON <Variable>:<Relation>,(<Qualification>)

Variable: A variable representing a tuple (record) within a relation (table).

Relation: The name of the table or relation the variable belongs to.

Qualification: The condition that must always be true.

B. Functional dependency

CHECK ON e1:EMP, e2:EMP

(e1.ENAME = e2.ENAME **IF** e1.ENO = e2.ENO)

This constraint applies to the EMP table, where e1 and e2 represent two records from the same table. It states that **if two records have the same Employee Number (ENO), then their Employee Name (ENAME) must also be the same**.

This enforces a **functional dependency**: ENO → ENAME, meaning an employee number must uniquely determine an employee name.

In a **distributed database**, enforcing this constraint requires ensuring data consistency even when employee records are stored across multiple nodes.

C. **Constraint with aggregate function**

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

a) **Data Distribution:**

The ASG and PROJ tables might be stored on different nodes in a distributed database system.

Querying both tables together (WHERE g.PNO = j.PNO) requires efficient **distributed joins**.

b) Aggregation Across Nodes:

Computing SUM(g.DUR) means that **partial results may need to be computed on different nodes** before being aggregated into a final sum.

A distributed database may use **MapReduce-style operations** to compute partial sums locally before combining them globally.

c) Consistency and Transaction Control:

If updates to ASG.DUR occur frequently, the system must ensure that constraint checks remain **consistent across nodes**.

Some distributed databases might use **eventual consistency**, while others may enforce strict constraints using **global transactions**.

3.10. Challenges in Distributed Databases

1. **Efficient Query Execution:** Ensuring that the sum calculation does not require excessive data shuffling between nodes.
2. **Constraint Enforcement at Scale:** Enforcing the constraint in real-time as new ASG.DUR values are inserted or updated.
3. **Concurrency Control:** Multiple transactions updating ASG.DUR simultaneously might cause violations if not handled correctly.

3.10.1. Integrity Enforcement

Integrity Enforcement has Two methods

1. **Detection**

   Execute update u: $D \rightarrow D_u$

   If D_u is inconsistent then

   ```
       if possible: compensate $D_u \rightarrow D_u'$
       else
          undo $D_u \rightarrow D$
   ```

2. **Preventive**

   Execute u: $D \rightarrow D_u$ only if D_u will be consistent

   - ❑ Determine valid programs
   - ❑ Determine valid states

3.10.2. Query Modification

- Preventive
- Add the assertion qualification to the update query
- Only applicable to tuple calculus formulae with universally quantified variables

```
UPDATE      PROJ
SET         BUDGET = BUDGET*1.1
WHERE       PNAME = "CAD/CAM"
```

⬇

```
UPDATE      PROJ
SET         BUDGET = BUDGET*1.1
WHERE       PNAME = "CAD/CAM"
AND         NEW.BUDGET ≥ 500000
AND         NEW.BUDGET ≤ 1000000
```

3.10.3. Compiled Assertions

Compiled assertions define **constraints** that must be enforced whenever a **relation (R) is updated in a certain way (T)**. They are written as **triples (R, T, C)**:

- **R** → The relation (table) affected by the update.
- **T** → The type of update (INSERT, DELETE, MODIFY).
- **C** → The assertion that must hold true based on **differential relations** (changes caused by the update).

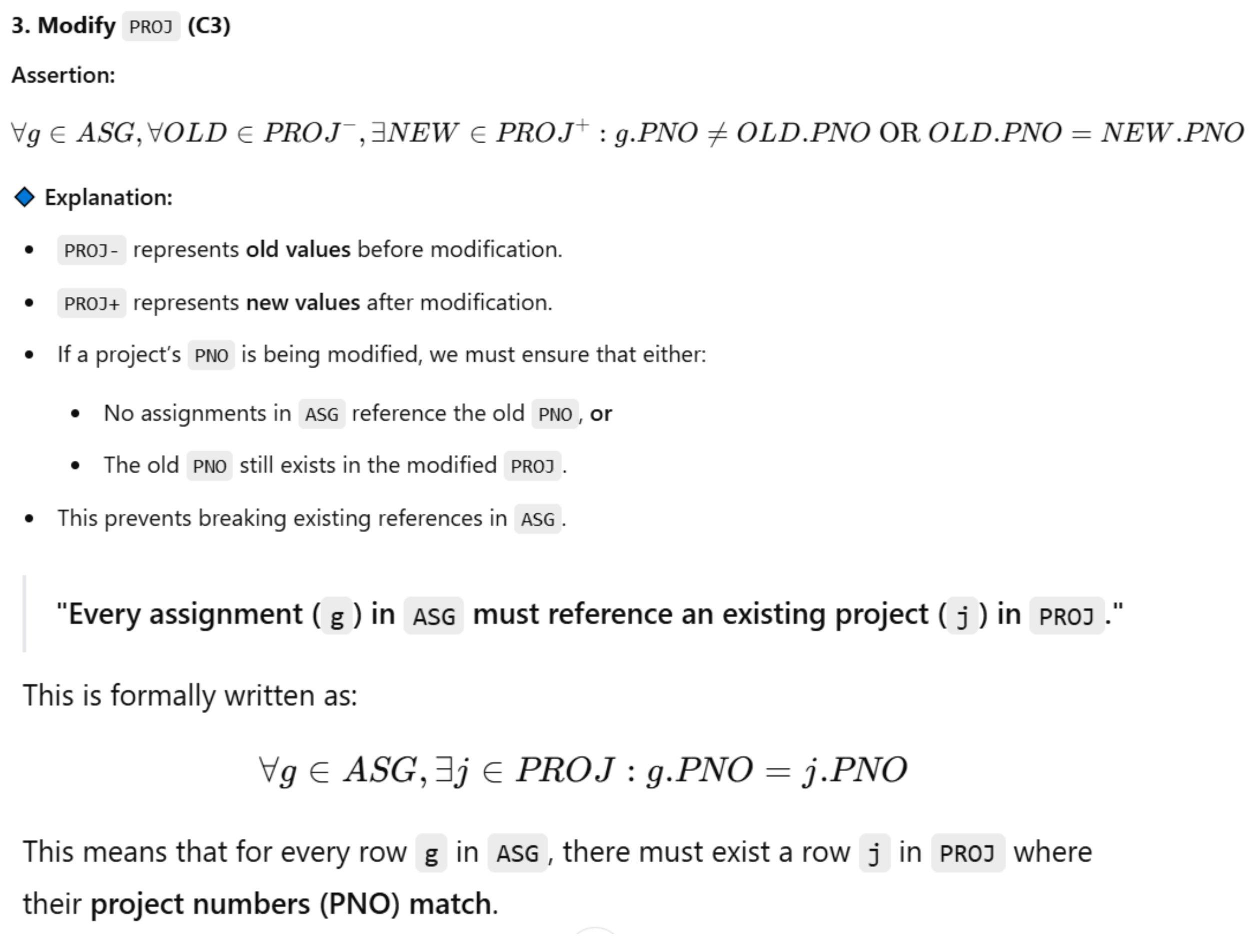
Compiled assertions define **foreign key constraints** across **INSERT, DELETE, and MODIFY** operations.

They ensure **referential integrity** in a database.
Enforcing them in **distributed systems** requires efficient constraint checking, distributed transactions, and possible use of eventual consistency mechanisms.

- **Example: Foreign key assertion**

$\forall g \in ASG, \exists j \in PROJ : g.PNO = j.PNO$

The basic **foreign key constraint** is:



3. Modify `PROJ` (C3)

Assertion:

$$\forall g \in ASG, \forall OLD \in PROJ^{-}, \exists NEW \in PROJ^{+} : g.PNO \neq OLD.PNO \text{ OR } OLD.PNO = NEW.PNO$$

🔷 **Explanation:**

- `PROJ-` represents **old values** before modification.
- `PROJ+` represents **new values** after modification.
- If a project's `PNO` is being modified, we must ensure that either:
  - No assignments in `ASG` reference the old `PNO`, **or**
  - The old `PNO` still exists in the modified `PROJ`.
- This prevents breaking existing references in `ASG`.

> **"Every assignment (`g`) in `ASG` must reference an existing project (`j`) in `PROJ`."**

This is formally written as:

$$\forall g \in ASG, \exists j \in PROJ : g.PNO = j.PNO$$

This means that for every row `g` in `ASG`, there must exist a row `j` in `PROJ` where their **project numbers (PNO) match**.

The Compiled Assertions:

(ASG, **INSERT**, C1), (PROJ, **DELETE**, C2), (PROJ, **MODIFY**, C3)

where

C1: $\forall$**NEW** $\in$ ASG+ $\exists j \in$ PROJ: NEW.PNO = j.PNO

C2: $\forall g \in$ ASG, $\forall$**OLD** $\in$ $PROJ^{-}$: g.PNO $\neq$ **OLD**.PNO

C3: $\forall g \in$ ASG, $\forall$**OLD** $\in$ $PROJ^{-}$ $\exists$**NEW** $\in$ $PROJ^{+}$:

g.PNO $\neq$**OLD**.PNO OR **OLD**.PNO = **NEW**.PNO

1. Insert into `ASG` (C1)

Assertion:

$$\forall NEW \in ASG^{+}, \exists j \in PROJ : NEW.PNO = j.PNO$$

🔷 **Explanation:**

- When a new row (`NEW`) is inserted into `ASG`, its `PNO` must already exist in `PROJ`.
- `ASG+` represents the newly inserted records in `ASG`.
- The system must check that for every new assignment, a matching project exists.

2. Delete from `PROJ` (C2)

Assertion:

$$\forall g \in ASG, \forall OLD \in PROJ^{-} : g.PNO \neq OLD.PNO$$

🔷 **Explanation:**

- `PROJ-` represents **deleted records** from `PROJ`.
- When a project is deleted, we must check if there are any assignments (`g`) referencing it in `ASG`.
- If such an assignment exists, deleting the project would leave an orphaned assignment, violating the foreign key constraint.
- The system must **prevent the deletion** or take corrective action (e.g., cascade delete or restrict delete).

3.11. Differential Relations

Given relation *R* and update *u*

R^{+} contains tuples inserted by *u*

R^{-} contains tuples deleted by *u*

Type of *u*

Insert	R^{-} empty
Delete	R^{+} empty
Modify	$R^{+} \cup (R - R^{-})$

- **R+ stores inserted tuples**
- **R- stores deleted tuples**
- **Insert:** Only R+ changes (new rows added)
- **Delete:** Only R- changes (rows removed)
- **Modify:** Both R+ and R- change (old rows removed, new rows inserted)

3.11.1. Algorithm:

Input: Relation *R*, update *u*, compiled assertion C_i

R → A relation (table) in the database.
u → An update operation (INSERT, DELETE, or MODIFY).
Ci → A compiled assertion that needs to be checked.

Steps of the Algorithm

1 Generate Differential Relations:

- Identify **R+ (inserted tuples)** and **R– (deleted tuples)** based on the update `u`.

2 Check for Constraint Violations:

- Retrieve all tuples from `R+` and `R-` that **do not satisfy** the assertion `Ci`.

3 Validate the Assertion:

- If no such violating tuples are found, the assertion holds **(i.e., the database remains consistent)**.

Example :

u is delete on J. Enforcing (EMP, DELETE, C2) :

retrieve all tuples of EMP^-
into RESULT
where not(C2)

If RESULT = {}, the assertion is verified

💡 **Scenario:**

- An **EMPLOYEE (EMP)** table exists.
- An update `u` **deletes tuples from another table J**.
- A compiled assertion **(EMP, DELETE, C2)** must be enforced.

🛠 **Execution Steps:**

1. Retrieve all tuples in `EMP-` (the deleted rows).
2. Check if any of these tuples violate `C2`.
3. If no tuples violate `C2` (`RESULT = {}`), then the assertion is **valid and enforced**.

3.12. Distributed Integrity Control (DIC)

DIC has Problems:

1. Definition of constraints : Consideration for fragments
2. Where to store: Replication and Non-replicated : fragments
3. Enforcement: Minimize costs

3.13. Types of Distributed Assertions

1. Individual assertions
   - ❑ Single relation, single variable
   - ❑ Domain constraint
2. Set oriented assertions
   - ❑ Single relation, multi-variable
     - ■ functional dependency
   - ❑ Multi-relation, multi-variable
     - ■ foreign key
3. Assertions involving aggregates

3.14. Assertions in DDBS

Assertion Definition: Similar to the centralized techniques, it Transform the assertions to compiled assertions

3.14.1. Assertion Storage

A. Individual assertions

   a) One relation, only fragments

   b) At each fragment site, check for compatibility

   c) If compatible, store; otherwise reject

   d) If all the sites reject, globally reject

B. Set-oriented assertions

   a) Involves joins (between fragments or relations)

   b) May be necessary to perform joins to check for compatibility

   c) Store if compatible

3.14.2. Assertion Enforcement

Where to enforce each assertion depends on

a) Type of assertion

b) Type of update and where update is issued

A. Individual Assertions
   a) If update = insert
      - ❑ Enforce at the site where the update is issued
   b) If update = qualified
      - ❑ Send the assertions to all the sites involved
      - ❑ Execute the qualification to obtain R^+ and R^-
      - ❑ Each site enforces its own assertion

B. Set-oriented Assertions
   a) Single relation
      - ❑ Similar to individual assertions with qualified updates
   b) Multi-relation
      - ❑ Move data to perform joins; then send the result to query master site

3.11. Conclusion

Solutions initially designed for centralized systems have been significantly extended for distributed systems

Materialized views and group-based discretionary access control

Semantic integrity control has received less attention and is generally not well supported by distributed DBMS products

Full data control is more complex and costly in distributed systems

a) Definition and storage of the rules (site selection)

b) Design of enforcement algorithms which minimize communication costs

Disclaimer: All content in this material is sourced from slides created by M.T. Özsu and P. Valduriez © 2020. These materials are derived from the reference text: Principles of Distributed Database Systems (3rd Edition), authored by M. Tamer Özsu and Patrick Valduriez, published by Springer Science+Business Media, LLC in 2015.