

## 3. Distributed Database Systems

**Reading:** Chapter 19

**Goal:** Learn and understand the basic concepts underlying distributed database systems, including design, architecture, query processing, and transaction management. Learn and understand concepts underlying multidatabase systems and database integration.

### What is a Distributed Database System?

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

A *distributed database management system* (DDBMS) is the software that manages the DDB and provides access mechanisms that make the distribution transparent to the user.

### Advantages of a DDBS (= DDB + DDBMS)

- Improved performance, reliability, and availability
- Economics, expandability, shareability
- Local autonomy (of the sites that participate)

Disadvantages: Complexity, Design, Distribution of Control

### Promises

- Transparent management of distributed, fragmented, and replicated data
- Improved reliability/availability through distributed transactions
- Improved performance

## Distributed DBS Issues

- Transparency issues (DDBS should appear as a single system)
- Distributed Database Design (fragmentation, allocation, and replication)
- Distributed Query Processing  
     $\rightsquigarrow \min\{\text{cost} \mid \text{cost} = \text{data transmission} + \text{local processing}\}$
- Distributed Transaction Management & Concurrency Control
- (Heterogeneous) Multidatabases, Client/Server Architectures, and Middleware

## Transparency Issues

*Transparency* is the separation of high level semantics of the system from lower level implementation issues

- ⇒ Hide the implementation details from the higher layers of the system and from the user  
(Fundamental transparency issue: *data independence*)

## Network Transparency

Existence of the network should not be noticed by user applications.

- *Location transparency*: Usage of commands in, e.g., query language, is independent of the location of the data.
- *Naming transparency*: Do not embed location of the data object into the name.

## Implications

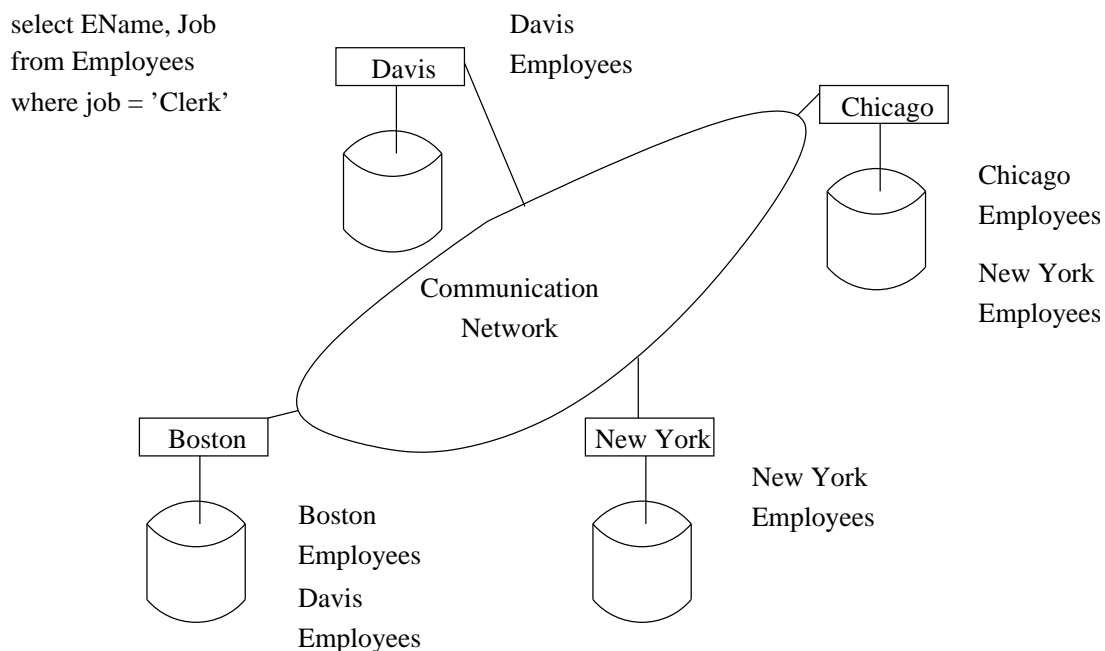
- Every database object must have a system-wide unique name
- It should be possible to find the location of a database object efficiently  
     $\rightsquigarrow$  data dictionary issues: distributed, centralized, or fully replicated.
- It should be possible to change the location of a database object transparently, and to create new database objects autonomously.  
     $\rightsquigarrow$  Use, e.g., aliases.

## Replication Transparency

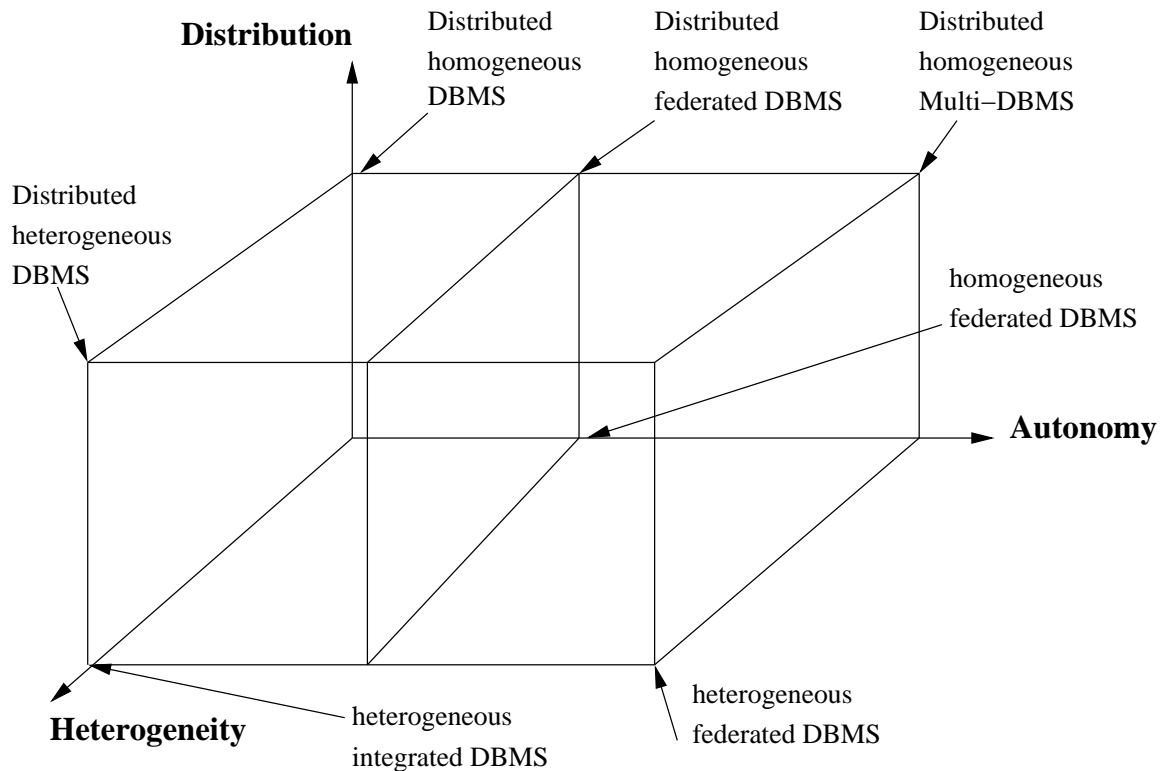
If there are replicas of database objects (which are typically (portions of) relations), their existence should be controlled by the system, not by the user. Includes: data modifications, i.e., each replica must be updated, concurrency control mechanisms must respect replicas with regard to read consistency

## Fragmentation Transparency

- If database relations are fragmented (horizontal, vertical, hybrid), then the system has to handle the conversion of user queries defined on global relations to queries defined on fragments.  
( $\rightsquigarrow$  query decomposition and optimization)
- The system must also put together (sub)query results from multiple sites into a single answer.



## DDBMS Implementation Alternatives

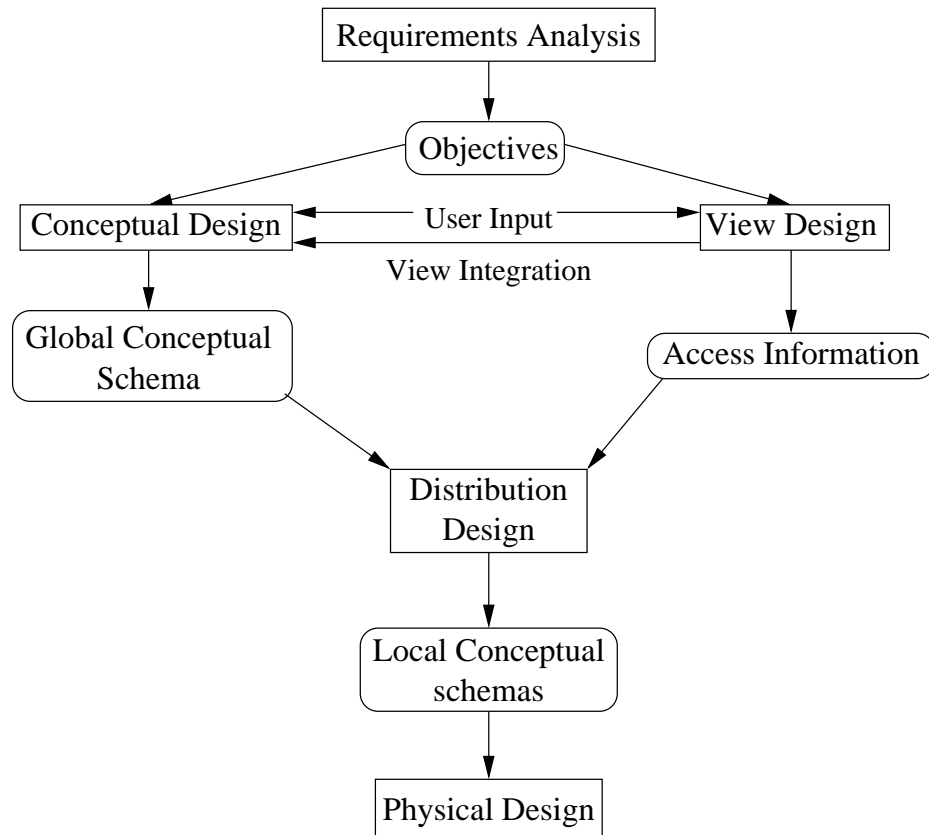


[From Özsu, Valduriez 1999]

### Dimensions of the Problem

- **Distribution:** Whether the components of the system are located on the same machine or not
- **Heterogeneity:**
  - Various levels (hardware/software platform, communication protocols, operating system, . . . )
  - DBMS specific: data model, query language, transaction management, . . .
- **Autonomy:** Design autonomy, communication autonomy, execution autonomy

## Distributed Database Design



## Distribution Design Issues

1. Why fragment at all? Can't we just distribute relations?  
What is a reasonable unit of distribution?
2. Fragmentation techniques and how much to fragment?
3. How to test correctness?
4. How to allocate fragments on the different sites?
5. Why and how to replicate fragments?

## Fragmentation

Partitioning of global relation  $R$  into fragments  $R_1, R_2, \dots, R_n$  which contain sufficient information to reconstruct the original relation. Four types of fragmentation: *primary horizontal, derived horizontal, vertical, and hybrid*.

## Advantages

- allows parallel processing of data
- tuples are located where they are most frequently accessed

**Prerequisite:** Access patterns (typical queries) and application behavior must be known; that is, which applications access which (portions of the) data?

## Primary Horizontal Fragmentation

Based on information requirements; for a given relation  $R(A_1, \dots, A_n)$  a set  $P_R = \{p_1, \dots, p_n\}$  of *simple predicates* is determined. Each  $p_i$  has the pattern  $A_i \theta \langle \text{value} \rangle$ , with  $\theta \in \{=, \neq, <, >\}$ .

Assume the two relations:

$EMP(\underline{EmpNo}, EName, Job, Sal, Deptno \rightarrow DEPT)$

$DEPT(\underline{Deptno}, DName, Location)$

Furthermore, assume that there are two applications. One application typically retrieves information about employees who earn more than \$5,000, the other application typically manages information about clerks.

Simple predicates  $P_{EMP} = \{Job = 'Clerk', Sal > 5000\}$

Set of *minterm predicates*  $M_R = \{m_i \mid [not]p_1 \wedge \dots \wedge [not]p_n\}$

$m_1 = Job = 'Clerk' \wedge Sal > 5000$        $m_2 = Job = 'Clerk' \wedge Sal \leq 5000$

$m_3 = Job \neq 'Clerk' \wedge Sal > 5000$        $m_4 = Job \neq 'Clerk' \wedge Sal \leq 5000$

Fragments  $EMP_i = \sigma_{m_i}(EMP), i = 1, \dots, 4$

## Derived Horizontal Fragmentation

Idea: Partitioning of a relation R is based on the partitioning of another relation S. Typically, some attributes in R reference primary key in S.

**Example:** Assume DEPT has been partitioned based on predicate  
 Location = 'Dallas', i.e., we have 2 partitions for DEPT.

$$DEPT_1 = \sigma_{\text{Location} = \text{'Dallas'}}(\text{DEPT})$$

$$DEPT_2 = \sigma_{\text{Location} \neq \text{'Dallas'}}(\text{DEPT})$$

EMP is partitioned based on the partitions of DEPT using *semi-join*  $\bowtie$ .  
 $R \bowtie S \equiv \pi_{\text{attributes}(R)}(R \bowtie S)$ .

$$EMP_i = EMP \bowtie DEPT_i, \quad i = 1, 2$$

Each  $EMP_i$  is a derived horizontal fragment.

Advantage: Based on the obtained partitions, joining the relations  
 EMP and DEPT can be done more efficiently (parallel processing).

## Vertical Fragmentation

Idea: Split schema of relation R into smaller schemas  $R_i$ . Splitting is based on which attributes are typically accessed together by (different) applications. Each fragment  $R_i$  must contain primary key in order to reconstruct original relation R.

**Example:**  $EMP_1(\underline{\text{EmpNo}}, \text{EName}, \text{Sal})$ ,  $EMP_2(\underline{\text{EmpNo}}, \text{Job}, \text{Deptno})$

More difficult than horizontal fragmentation, because more alternatives exist.  
 Two approaches: (1) grouping attributes to fragments, (2) splitting relation into fragments.

## Correctness of Fragmentation

- **Completeness:** Decomposition of relation  $R$  into fragments  $R_1, R_2, \dots, R_n$  is complete if and only if each tuple in  $R$  can also be found in some  $R_i$ .
- **Reconstruction:** If relation  $R$  is decomposed into fragments  $R_1, R_2, \dots, R_n$ , then there should be some relational operator  $\otimes$  such that  $R = \otimes_{1 \leq i \leq n} (R_i)$ .
- **Disjointness:** If a relation  $R$  is partitioned, and a tuple  $t$  is in fragment  $R_i$ , then  $t$  should be in no other fragment  $R_j, j \neq i$ .

## Allocation

**Problem Statement:** Given

- (1)  $F = \{F_1, F_2, \dots, F_n\}$  fragments,
- (2)  $S = \{S_1, S_2, \dots, S_m\}$  network sites,
- (2)  $Q = \{Q_1, Q_2, \dots, Q_q\}$  applications.

Find the “optimal” distribution of the fragments in  $F$  to the sites  $S$ .

## Optimality

- Minimal cost: communication cost + storage + processing (read & update); Cost in terms of time and/or network cost
- Performance: Response time and/or throughput
- Constraints: Per site constraints (storage & processing)

## Allocation – Information Requirements

- Database Information
  - conceptual database schema, number of sites
  - number, size, and selectivity of fragments per global relation
- Application Information
  - number of queries (applications)
  - average number of read accesses of a query to a fragment
  - average number of update accesses of a query to a fragment
  - matrix indicating which queries update which fragments
  - similar matrix for read accesses
  - originating site of each query
- Site Information
  - unit cost of storing data at a site
  - unit cost of processing at a site
- Network Information
  - communication cost between two sites – (bandwidth, latency, . . . )

### Allocation Model (Most General Form)

min(Total\_Cost) is subject to response time constraints,  
storage constraints, and processing constraints

### Total Cost

$$\sum_{\text{all queries}} \text{query processing cost} +$$

$$\sum_{\text{all sites}} \sum_{\text{all fragments}} \text{cost of storing a fragment at a site}$$

Result: Place fragment  $F_i$  at sites  $S_j, \dots, S_k, i \in \{1, \dots, n\}$

This is an NP-complete problem  $\rightsquigarrow$  employ heuristics

## Replication

System maintains multiple copies of data (fragments), stored at different sites, for faster retrieval and fault tolerance.

- A relation or fragment of a relation is replicated if it is stored *redundantly* at two or more sites.
- *Full replication* of a relation is the case where the relation is stored at all sites.
- *Fully redundant databases* are those in which every site contains a copy of the entire database.
- Rule:  $\frac{\# \text{ read-only requests against fragment } F}{\# \text{ update requests against fragment } F} \gg 1 \implies \text{replicate } F$

## Advantages

- Availability: Failure of a site containing fragment/relation R does not result in unavailability of R if replicas of R exist.
- Parallelism: Queries on R may be processed by several nodes in parallel.
- Reduced data transfer: Relation R is available locally at each site containing a replica of R

## Disadvantages

- Increased cost of updates: Each replica of R must be updated.
- Increased complexity of concurrency control: Concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.

## Strategies

- *Synchronous replication*: Before an update transaction commits, it synchronizes all copies of the modified data ( $\rightsquigarrow$  Read-One-Write-All (ROWA) technique)
- *Asynchronous replication*: Changes to primary copy are propagated (periodically) to secondary copies ( $\rightsquigarrow$  snapshots).

## Distributed Query Processing

- For centralized DBSs, the primary criteria for measuring the cost of a query evaluation strategy is the number of disk accesses (# blocks read / written).
- For distributed databases, additionally the (1) cost of data transmission over the network and the (2) potential gain in performance from having several sites processing parts of the query in parallel must be taken into account.

### Issues if relations are fragmented

- Translating (global) queries to queries over fragments:  
Replace relation R in global query by expressions that describe the fragments of R;  
it must be possible to reconstruct relation R from its fragments.
- Simplification of expressions of relational algebra, detection and elimination of redundant subexpressions
- Choosing optimal join strategy (“semi-join programs”)
- Choosing optimal query evaluation plan

### Example

Assume global relation PARTS(PartNo, OrderNo, Price) with

$$\text{PARTS} := \text{PARTS1} \cup \text{PARTS2} \cup \text{PARTS3}$$

and the partitioning:

$$\text{PARTS1} := \sigma_{0 \leq \text{PartNo} \leq 300}(\text{PARTS})$$
$$\text{PARTS2} := \sigma_{301 \leq \text{PartNo} \leq 500}(\text{PARTS})$$
$$\text{PARTS3} := \sigma_{501 \leq \text{PartNo} \leq 999}(\text{PARTS})$$

### Example (cont.)

Assume the query:  $Q = \sigma_{25 \leq \text{PartNo} \leq 350}(\text{PARTS})$

. Possible equivalent evaluation plans based on fragments are:

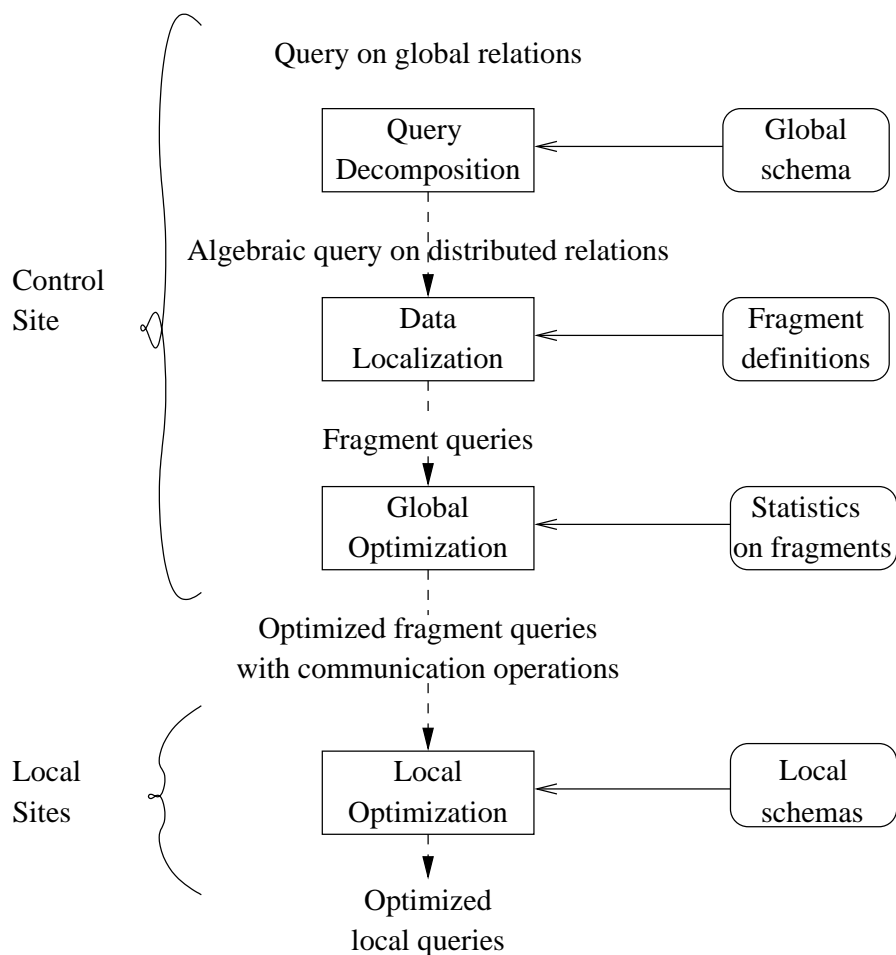
1. Replacing global relation by fragment definitions:

$$Q1 = \sigma_{25 \leq \text{PartNo} \leq 350}(\text{PARTS1} \cup \text{PARTS2} \cup \text{PARTS3})$$

2. Push-down of the selection to fragments:

$$Q2 = \sigma_{25 \leq \text{PartNo} \leq 350}(\text{PARTS1}) \cup \sigma_{25 \leq \text{PartNo} \leq 350}(\text{PARTS2}) \\ \cup \sigma_{25 \leq \text{PartNo} \leq 350}(\text{PARTS3})$$

### Distributed Query Processing Methodology



## Distributed Query Processing Methodology

Algebraic query optimization (e.g., pushing down selections, see 165A) must take definition of fragments (e.g., minterms  $m_i$ ) into account.

Assume, for example, fragment  $R_i$  of relation  $R$

$$\sigma_F(R_i) \implies \sigma_{F \wedge m_i}(R_i).$$

$\implies$  can lead to redundant fragment queries because selection condition  $F \wedge m_i$  is unsatisfiable.

## Parallel Processing of Fragment Queries

- Horizontal fragmentation  $R := R_1 \cup \dots \cup R_n, R_i \cap R_j = \emptyset, i \neq j$

$$\sigma_F(R) \equiv (\sigma_F(R_1)) \cup \dots \cup (\sigma_F(R_n))$$

$$\pi_{\text{attr}}(R) \equiv (\pi_{\text{attr}}(R_1)) \cup \dots \cup (\pi_{\text{attr}}(R_n))$$

- Aggregate functions (let  $Q(R)$  be a single column relation)

$$- \min(Q(R)) \equiv \min(\min(Q(R_1)), \dots, \min(Q(R_n)))$$

$$- \max(Q(R)) \equiv \max(\max(Q(R_1)), \dots, \max(Q(R_n)))$$

$$- \text{sum}(Q(R)) \equiv \sum_{i=1}^n \text{sum}(Q(R_i)), \quad \text{count, avg}$$

- Join Operations  $R \bowtie S$

If fragmentation of  $S$  is derived based on fragmentation of  $R$ , then we have a *simple join graph*, i.e., each fragment from  $R$  needs only be joined with corresponding fragment from  $S$ .

Example: Let  $R = R_1 \cup R_2$  and  $S = S_1 \cup S_2$

$$\implies R \bowtie S \equiv (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2)$$

## Simple Join Processing

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

$$R \bowtie S \bowtie T$$

Assume  $R$  is stored at site  $S_1$ ,  $S$  at site  $S_2$ , and  $T$  at site  $S_3$ . For a query issued at site  $S_i$ , the system must produce the result at site  $S_i$ .

### Possible Query Processing Strategies

- Ship copies of all three relations to site  $S_i$ , and choose strategy to optimize entire query locally at  $S_i$ .
- (1) Ship a copy of  $R$  to  $S_2$  and compute  $\text{temp1} = R \bowtie S$ .  
(2) Ship  $\text{temp1}$  from  $S_2$  to  $S_3$  and compute  $\text{temp2} = \text{temp1} \bowtie T$ .  
(3) Finally ship  $\text{temp2}$  to  $S_i$ .
- Devise similar strategies, exchanging the roles of  $S_1, S_2, S_3$ .

Suitable join processing strategy must consider the following factors:

- amount of data being shipped
- cost of transmitting a data block between sites
- relative processing speed at each site

## Semijoin Strategy

Definition of a semijoin:

$$R \bowtie S \equiv \pi_{\text{attr}(R)}(R \bowtie S)$$

- Let  $R$  be a relation stored at site  $S_1$ , and  $S$  a relation stored at site  $S_2$ .
- Evaluate the query  $R \bowtie S$ , and obtain result at  $S_1$ .
  1. Compute  $\text{temp1} := \pi_{\text{attr}(R) \cap \text{attr}(S)}(R)$  at site  $S_1$ .  
( $\leadsto$  determine values for join attribute(s))
  2. Ship  $\text{temp1}$  from site  $S_1$  to site  $S_2$ .
  3. Compute  $\text{temp2} := S \bowtie \text{temp1}$  at site  $S_2$ .  
( $\leadsto$   $\text{temp2}$  contains only tuples from  $S$  that have matching tuples in  $R$ )
  4. Ship  $\text{temp2}$  from  $S_2$  to  $S_1$ .
  5. Compute  $R \bowtie \text{temp2}$  at  $S_1$ . The result then is  $R \bowtie S$ .

In step 3 above,  $\text{temp2} := S \bowtie R$ .

### **(Semi-)Join Programs**

It is the task of the query processor to partition a series of joins  $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$  into “more efficient” series of semijoins.

Choosing an efficient series of semijoins depends on transmission cost and transmission time (as well as associated constraints, e.g., that a particular query must be answered within 5 minutes).

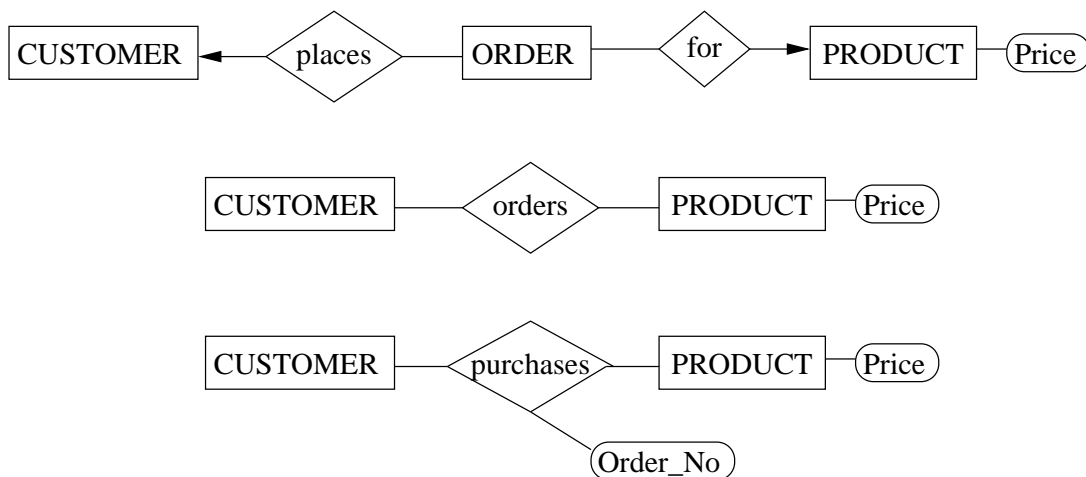
## Design Strategies for Distributed & Centralized Databases

### Top-Down Approach

- Suitable approach when a database is being designed from scratch (centralized DBS ✓, distributed databases ✓)
- Often in combination with *View Integration*: Find all parts of the input external schemas (based on, e.g., ER Model) that refer to the same portion of real world and unify their representation into one conceptual schema.

**Problem** in view integration: The same portion of real world is usually modeled in different ways in each schema  
( $\rightsquigarrow$  schematic and semantic heterogeneity)

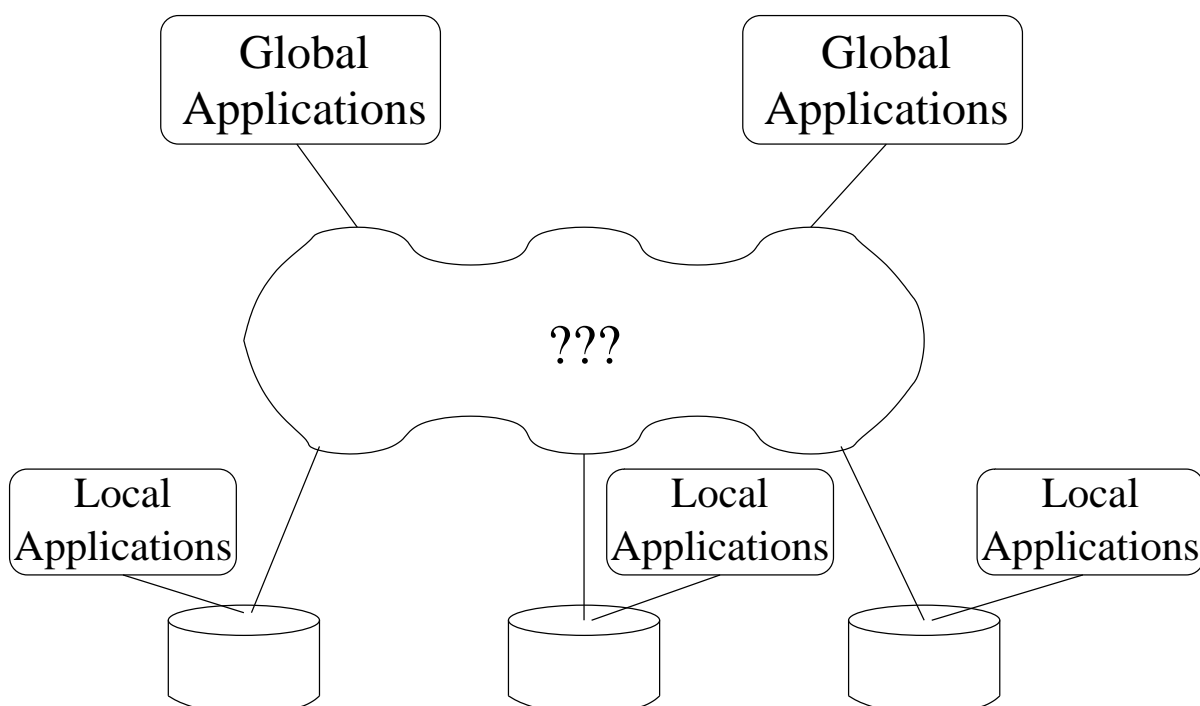
### Example:



## Bottom-Up Approach:

- Used in *database integration* to build a *federated* or *multidatabase system*.
- Typical approach when a number of databases already exist and global interfaces to local databases need to be developed for new global applications
- **Database Integration:** Merge several different databases and schemas (relational, OO, ER, . . . ) into a single database with one global conceptual schema.  
Conceptual schema is a *virtual view* of all databases taken together in a distributed database environment.
- **Problems:** Same as for view integration + *system heterogeneity* (different hardware/software platforms, non-uniformity in data models and languages, . . . )  
Respect autonomy of local databases!

### Integration Scenario

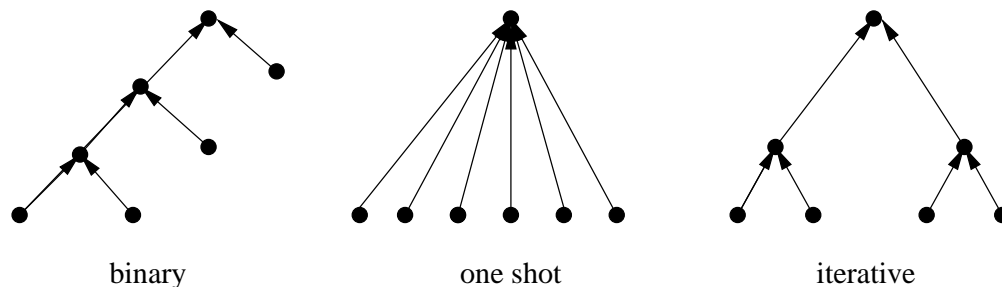


## Reasons for Database Integration

- Structure of database for large enterprise is too complex to be modeled by single designer in a single view.
- User groups typically operate independently in organizations and have different requirements and expectations.
- New “global” applications require controlled and integrated access to different local databases.
- Merging of companies and departments, . . .

## Basic Steps in Database Integration

1. Pre-integration Analysis – choose an integration strategy, strategy depends on number and complexity of local databases to integrate



(in case of heterogeneous schemas, first transform schemas into global data model, typically an object-based model)

2. Comparisons of schemas – detect correspondences/conflicts among the elements of the schemas
3. Conforming the schemas – modify local schemas such that all conflicts are resolved ( $\rightsquigarrow$  export schema)
4. Merging of export schemas and restructuring

## Schematic and Semantic Heterogeneity

### Schema Conflicts (based on the relational data model)

- I. table-table conflicts
  - a. one table vs. one table
    1. table name conflicts: homonyms, synonyms
    2. table structure conflicts: missing attributes
    3. integrity constraints (e.g., primary key constraints)
  - b. multiple tables vs. multiple tables
- II. attribute-attribute conflicts
  - a. one attribute vs. one attribute
    1. attribute name conflicts: homonyms, synonyms
    2. integrity constraint conflicts (e.g., data type constraints)
  - b. multiple attributes vs. multiple attributes
- III. table-attribute conflicts

### Data Conflicts

- I. different representations (precision, measurement, units, . . . )
- II. wrong data: incorrect data, obsolete data

At component database CDB1:

STUDENT	<u>StID</u>	FName	LName	Birthdate
	1011	Scott	Tiger	1965

GRADES	<u>StID</u>	<u>Course</u>	Grade	COURSE	<u>CRN#</u>	Course
	1011	59069	3.7		59069	ECS165B

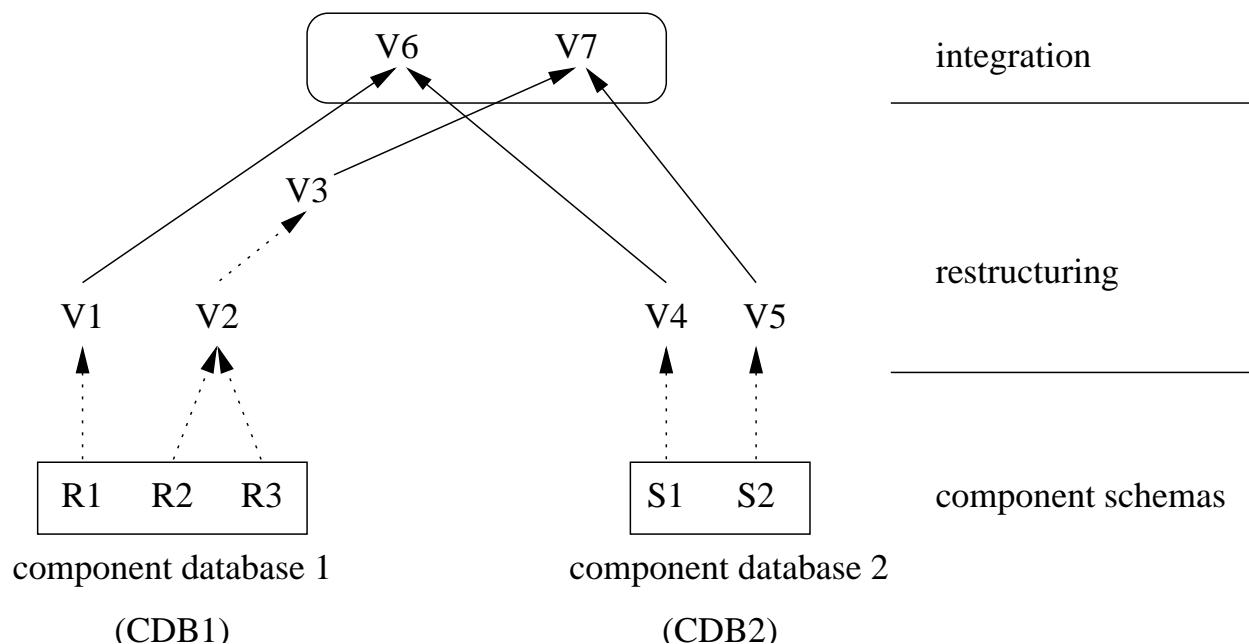
At component database CDB2:

STUDENTS	<u>ID</u>	Name	Birthdate	<u>Course</u>	Grade
	1011	Tiger, Scott	65	ECS165B	A-

## Resolving Conflicts

- Identification of conflicts requires detailed knowledge about the semantics of the relations and data stored at each participating component database
- Conflicts cannot be resolved automatically, i.e., resolving conflicts almost always requires human interaction
- Resolving conflicts typically requires (local) restructuring of participating relation schemas ( $\rightsquigarrow$  preparation for integration)
- Integration typically is based on views that reflect (local) restructuring of participating base relations

### Local restructuring and integration



**Note:** Restructuring already takes place at the integration level  
 (☞ respect autonomy of component databases)

## Views (Review)

A **view** is a virtual relation that is not part of the conceptual schema (i.e., it is not a base relation) but is made visible to users through a query over base relations and other views  
( $\rightsquigarrow$  logical data independence)

### Examples of view definitions

```
create view ECS165B_students(ID, Name, BDate, Grade)
  as select S.StId, LName, Birthdate, Grade
     from STUDENT S, GRADES G, COURSE C
     where S.StId = G.StId and G.Course = C.CRN#
     and C.Course = 'ECS165B';
```

```
create view good_ECS165B_students
  as select * from ECS165B_students
     where Grade >= 3.0;
```

- View definitions are stored in the data dictionary. Views don't require storage (except their definition).
- At any given time, the set of tuples in a view is defined as the result of the query expression that defines the view at that time.
- Whenever a view is used in a query, the query processor replaces it by the stored query expression. Thus, whenever the query is evaluated, the view is recomputed.

For further information about views and in particular updates, insertions, and deletions on views, see Silberschatz et al. Sections 3.7 and 4.8.

## Using Views for Local Restructuring and Integration

**Basic Idea:** Use views to locally restructure and combine base relations in order to resolve conflicts and to prepare base relations for integration.

Most simple case: There are no conflicts between two relations  $R1@CDB1$  and  $R2@CDB2$  that are to be integrated:

```
create view R /* view in the integrated schema */
  as select * from R1@CDB1 union
      select * from R2@CDB2
```

In most cases, however, base relations are not union compatible!!

### **Attribute vs Attribute Conflicts:**

- Assume two relations  $R(A:\text{integer}, B:\text{char})$ , and  $S(A:\text{integer}, D:\text{integer}, C:\text{char})$
- attribute name conflicts can easily be resolved by renaming the attribute in the view definition (e.g., rename  $Re1\_A$  to  $Re1\_B$ )
- attribute data type conflicts often can be resolved using data type conversion function, e.g.,  $to\_char(\langle \text{number} \rangle)$  ( $\text{number} \rightarrow \text{string}$ ),  $to\_number\langle \text{string} \rangle$  ( $\text{string} \rightarrow \text{number}$ ),  $to\_date$  ( $\text{string} \rightarrow \text{date}$ ), etc.

Possible data type conversions depend on the language features provided by the DML language, for example, SQL.

```
substr(replace('ID 10-123', '-', ''), 3, 6) = 10123
```

Other useful functions in Oracle SQL: `translate`, `replace`, `substr`, `instr`, ...

### Multiple Attributes vs Multiple Attributes Conflicts:

- In the relation STUDENT, the name of a student is stored using two attributes (FName, LName) while in STUDENTS it is only one attribute.

There are two possibilities to resolve this conflict:

- I. `select .., LName||', '||FName, .. from STUDENT`
- II. `select ..substr(Name,instr(Name,',')+2) FName,  
          substr(Name,1,instr(Name,',')-1) LName,  
from STUDENTS`

### Table vs Table Conflicts

- Table name conflicts ✓ (resolved in view definition)
- Order of the attributes in the relations ✓ (resolved in view definition)
- Differences in the number of attributes. In the example, R has two attributes, S has three attributes:
  - I. Either don't consider the attribute C for integration, or
  - II. extend R by a matching attribute C (and default value), e.g.,  
`select A, B, 'small' C from R;`
- Table constraint conflicts, e.g., primary keys:

Prod	ProdlId	Prodname	Prod	ProdlId	Prodname
	1014	TinyMac		1012	TinyMac
	2012	MegaPC		2130	CDRom
	3000	CDRom		3000	MegaPC
	...	...		...	...

- Problem: In the integrated schema, a product cannot simply be identified by ProdId. Possible solution: Extend each relation schema by an attribute that reflects, e.g., the origin of the information (can also be helpful in other cases).

```
create view prod(site, ProdId, Name) as
select 'site1', ProdId, Name from prod@site1 union
select 'site2', ProdId, Name from prod@site2
```

site and ProdId build a composite primary key.

### Multiple Tables vs Multiple Tables Conflicts

- Main problem: component databases (local schemas) use different numbers of relations to represent the same information.
- Recall relations that store information about students and grades: information represented in one relation @CDB2 is represented in three relations @CDB1.
- Idea: Combine relations locally (using views) before integration.  
For this, use foreign key dependencies.

```
create view STUD_GRAD_COURSE(StId, FName, LName,
                           Birthdate, Course, CRN#, Grade)
as select S.StId, FName, LName, Birthdate,
         C.Course, CRN#, Grade
   from STUDENT S, GRADES G, COURSE C
  where S.StId = G.StId and G.Course = C.CRN#
```

- Problem: What if for a student no grades exist?  
⇒ Use *outer joins* for view definition

**Example:**

EMP	EmpId	EName	DNo	DEPT	DNo	DName
	100	Smith	10		10	Sales
	102	Jones	20		20	Production
	104	Miller	20		30	Management
	110	Rogers	30		40	Headquarters
	112	Clark	<i>null</i>			

**Outer Join:** Extension of the join operator that deals with missing information (missing matching tuples). Three forms:

*left outer join*  $\bowtie\leftarrow$ ,  
*right outer join*  $\rightarrow\bowtie$  and  
*full outer join*  $\bowtie\leftarrow\rightarrow$ .

Left outer join ( $\text{EMP} \bowtie\leftarrow \text{DEPT}$ ) takes all tuples from the left relation that do not match with any tuple in the right relation, (1) pads these tuples with *null* values for all other attributes from the right relation, and (2) adds them to the result of the natural join  $\text{EMP} \bowtie \text{DEPT}$ .

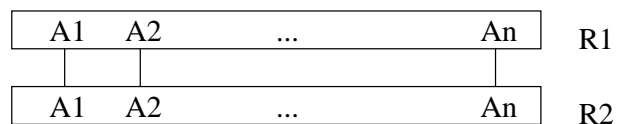
```
select * from EMP E, DEPT D
where E.DNo = D.DNo (+)
```

EMP $\bowtie$ DEPT	EmpId	EName	DNo	DName
	100	Smith	10	Sales
		...		
	112	Clark	<i>null</i>	<i>null</i>

Instead of the last tuple, a right outer join would yield the tuple (*null*, *null*, 40, Headquarters). Full outer join would yield both tuples.

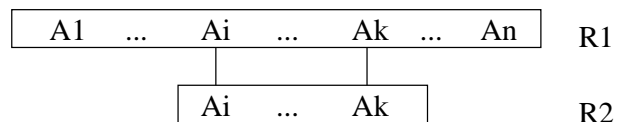
**Integration:** Creating views that represent the elements of the global conceptual schema. Views refer to base relations from the component databases and/or conflict resolving views (R1 and R2).

### I. Identical relation schemas:



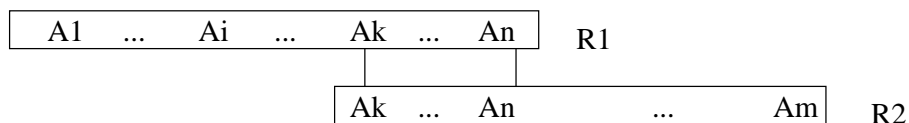
```
create view R as select * from R1
union select * from R2
```

### II. Subset schema:



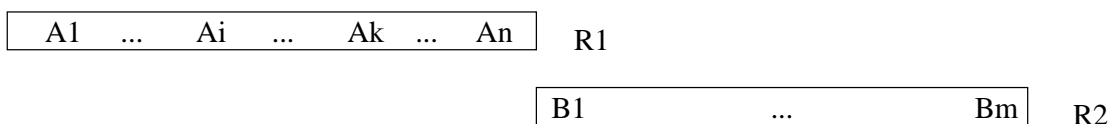
```
create view R as select Ai, ..., Ak from R1
union select Ai, ..., Ak from R2
```

### III. Overlapping schemas:



```
create view R as select Ak, ..., An from R1
union select Ak, ..., An from R2
```

### IV. Non-Overlapping schemas:



No integration.

**Example**

1. Resolve conflicts between the attribute Birthdate from the relation STUDENT and the attribute Birthdate from the relation STUDENTS.

```
create view STUD1(ID, Name, Birthdate, Course, Grade)
as select ID, Name, Birthdate+1900, Course, Grade
   from STUDENTS;
```

2. Resolve conflicts between the attributes FName, LName, and the attribute Name.

```
create view STUD2(StID, Name, Birthdate) as
select StId, LName||', '||FName, Birthdate from STUDENT;
```

3. Combine the three relations from CDB1 into one relation.

```
create view STUD(ID, Name, Birthdate, Course, Grade)
as select StId, Name, Birthdate, C.Course, Grade
   from STUD2 S, GRADES G, COURSE C
   where S.StId=G.StId and G.Course=C.CRN#
```

4. The relations (views) STUD and STUD1 are now union compatible except for the attribute Grade.

Idea: Use temporary relation to get “right values” for the attribute Grade in the view STUD1 (conversion table).

MY_GRADES	GLetter	GNum
	A	4.0
	...	
	D	1.0

## 3.2. Distributed Computing Architectures

### Client/Sever Architectures

- Because of more powerful, faster, and cheaper personal computers, there has been a shift from centralized system architectures (*downsizing*, *rightsizing*).
- Server systems satisfy requests generated at client systems (formerly simple terminals connected to a centralized systems)
- The client/server model (C/S) essentially is a software architecture model.
- Realizations of C/S systems differ in how the functionality of an application is distributed over client and server:
  - presentation functions (input devices, forms etc.)
  - application functions represent the program logic specific to the application
  - data management functions realize access to data stored in a file- or database system



- In C/S systems, database functionality can be divided into
  - *back-end*, which manages access structures, query evaluation and optimization, concurrency control and recovery.
  - *front-end*, which consists of tools such as forms, report-writers and graphical user interface facilities.
- The interface between front-end and back-end is through SQL or through an application program interface (middleware such as ODBC, JDBC).
- Server systems can be categorized into *transaction servers* or *data servers*.

## Transaction Servers

- Also called *query server* systems or *SQL server* systems; clients send requests to the server system where the transaction is executed, and the results are shipped back to the client.
- Request are typically specified in SQL, or through an application program interface (API) such as Open Database Connectivity (ODBC) or JDBC, using a *remote procedure call* (RPC) mechanism.
- Advantages of replacing centralized systems (e.g., mainframes) with networks of workstations or PCs connected to back-end server machine(s):
  - better functionality for the cost
  - flexibility in locating resources and expanding facilities
  - better user interfaces
  - easier maintenance

## Data Servers

- Typically used in LANs providing high-speed connection between the clients and the server.
- Client machines are comparable in processing power to the server machine, and the task to be executed are compute intensive.
- Ship data to client machines where processing is performed (and which may take a while), and then ship back the result to the server machine.
- This architecture requires full back-end functionality at the clients.

- Issues that arise in such an architecture:
  - Page shipping versus item (object, tuple) shipping  
    ↪ prefetching.
  - Locking: techniques for lock deescalation allow the server to request its clients to transfer back locks on prefetched items.
  - Data caching: if transaction finds cached data, it must make sure that those data are up-to-date, since they may have been updated by a different client after they were cached.
  - Lock caching: locks can also be cached at the client machines. But if a client requests a lock from the server, the server must call back all conflicting locks on the data item from any other client machine that has cached the locks.