# 2

# DATA DISTRIBUTION ALTERNATIVES

In a distributed database management system (DDBMS) data is intentionally distributed to take advantage of all computing resources that are available to the organization. For these systems, the schema design is done top–down. A **top–down** design approach considers the data requirements of the **entire organization** and generates a **global conceptual model (GCM)** of all the information that is required. The GCM is then distributed across all appropriate local DBMS (LDBMS) engines to generate the **local conceptual model (LCM)** for each participant LDBMS. As a result, DDBMSs always have **one and only one** GCM and one or more LCM. Figure 2.1 depicts the top–down distribution design approach in a distributed database management system.

By contrast, the design of a federated database system is done from the bottom–up. A **bottom–up** design approach considers the existing data distributed within an organization and uses a process called schema integration to create at least one unified schema (Fig. 2.2). The **unified schema** is similar to the GCM, except that there can be more than one unified schema. **Schema integration** is a process that uses a collection of existing conceptual model elements, which have previously been exported from one or more LCMs, to generate a semantically integrated model (a single, unified schema). We will examine the details of federated database systems in Chapter 12.

Designers of a distributed database (DDB) will decide what distribution alternative is best for a given situation. They may decide to keep every table intact (all rows and all columns of every table are stored in the same DB at the same Site) or to break up some of the tables into smaller chunks of data called **fragments** or **partitions**. In a distributed database, the designers may decide to store these fragments locally (**localized**) or store these fragments across a number of LDBMSs on the network (**distributed**).

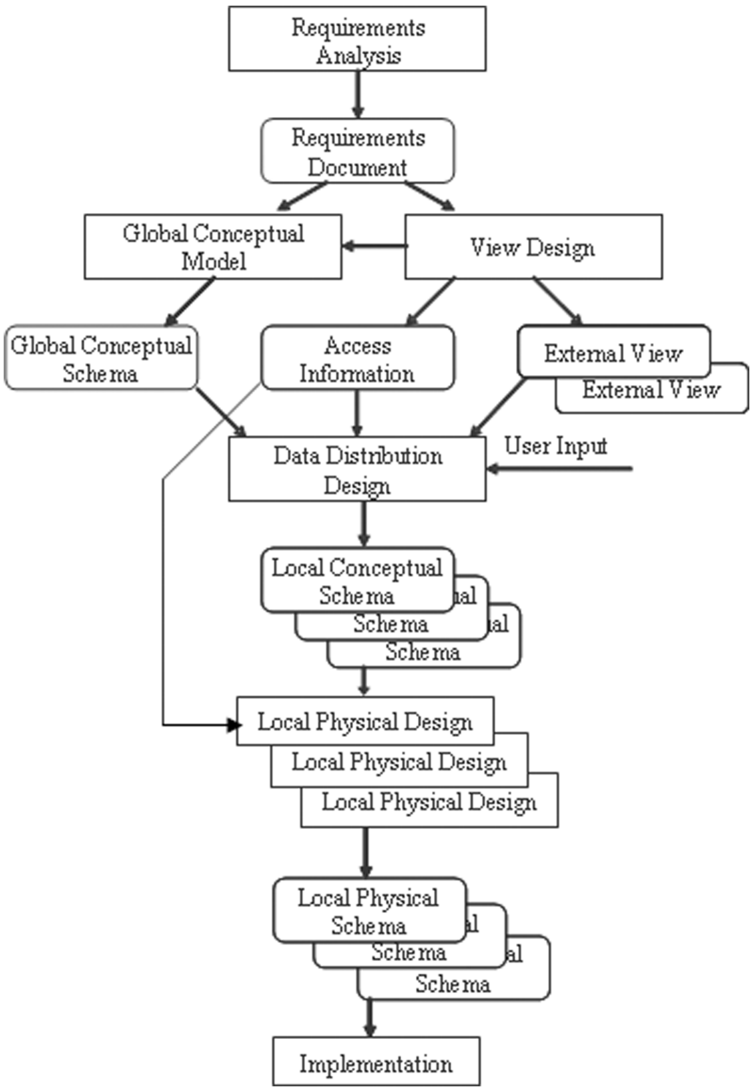


**Figure 2.1** The top–down design process for a distributed database system.

Distributed tables can have one of the following forms:

- Nonreplicated, nonfragmented (nonpartitioned)
- Fully replicated (all tables)
- Fragmented (also known as partitioned)
- Partially replicated (some tables or some fragments)
- Mixed (any combination of the above)

The goal of any data distribution is to provide for increased availability, reliability, and improved query access time. On the other hand, as opposed to query access time,

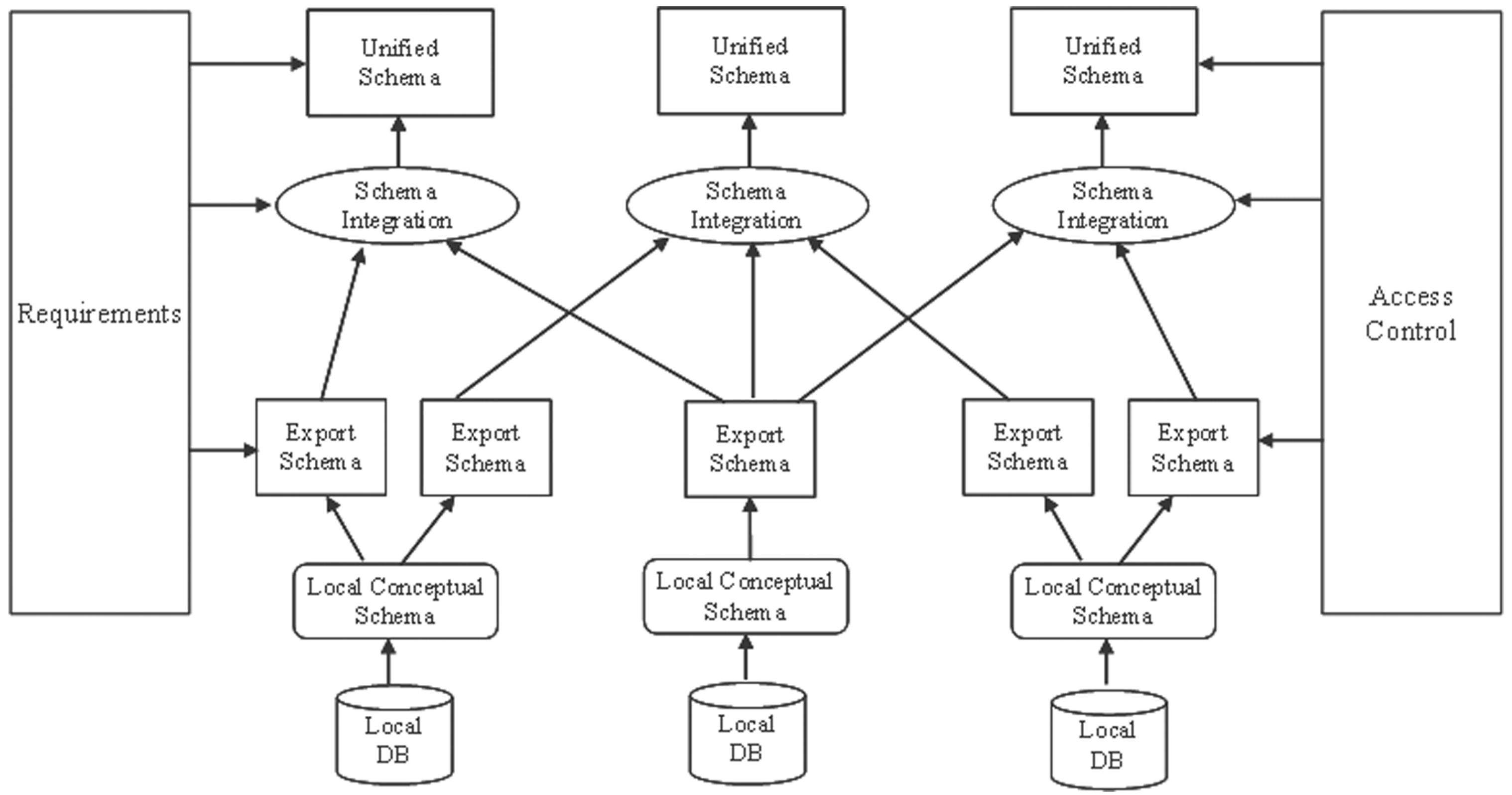


**Figure 2.2** The bottom–up design process for a federated database system using schema integration.

distributed data generally takes more time for modification (update, delete, and insert). The impact of distribution on performance of queries and updates is well studied by researchers [Ceri87], [Özsu99]. It has been proved that for a given distribution design and a set of applications that query and update distributed data, determining the optimal data allocation strategy for database servers in a distributed system is an NP-complete problem [Eswaren74]. That is why most designers are not seeking the best data distribution and allocation design but one that minimizes some of the cost elements. We will examine the impact of distribution on the performance of queries and updates in more detail in Chapter 4.

## 2.1 DESIGN ALTERNATIVES

In this section, we will explore the details of each one of the design alternatives mentioned earlier.

### 2.1.1 Localized Data

This design alternative keeps all data logically belonging to a given DBMS at one site (usually the site where the controlling DBMS runs). This design alternative is sometimes called "not distributed."

### 2.1.2 Distributed Data

A database is said to be distributed if any of its tables are stored at different sites; one or more of its tables are replicated and their copies are stored at different sites; one or more of its tables are fragmented and the fragments are stored at different sites; and so on. In general, a database is distributed if not all of its data is localized at a single site.

***2.1.2.1 Nonreplicated, Nonfragmented*** This design alternative allows a designer to place different tables of a given database at different sites. The idea is that data should be placed close to (or at the site) where it is needed the most. One benefit of such data placement is the reduction of the communication component of the processing cost. For example, assume a database has two tables called "EMP" and "DEPT." A designer of DDBMS may decide to place EMP at Site 1 and DEPT at Site 2. Although queries against the EMP table or the DEPT table are processed locally at Site 1 and Site 2, respectively, any queries against both EMP and DEPT together (join queries) will require a distributed query execution. The question that arises here is, "How would a designer decide on a specific data distribution?" The answer depends on the usage pattern for these two tables. This distribution allows for efficient access to each individual table from Sites 1 and 2. This is a good design if we assume there are a high number of queries needing access to the entire EMP table issued at Site 1 and a high number of queries needing access to the entire DEPT table issued at Site 2. Obviously, this design also assumes that the percentage of queries needing to join information across EMP and DEPT is low.

***2.1.2.2 Fully Replicated*** This design alternative stores one copy of each database table at every site. Since every local system has a complete copy of the entire database, all queries can be handled locally. This design alternative therefore provides for the best possible query performance. On the other hand, since all copies need to be in sync—show the same values—the update performance is impacted negatively. Designers of a DDBMS must evaluate the percentage of queries versus updates to make sure that deploying a fully replicated database has an overall acceptable performance for both queries and updates.

***2.1.2.3 Fragmented or Partitioned*** Fragmentation design approach breaks a table up into two or more pieces called fragments or partitions and allows storage of these pieces in different sites.

There are three alternatives to fragmentation:

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

This distribution alternative is based on the belief that not all the data within a table is required at a given site. In addition, fragmentation provides for increased parallelism, access, disaster recovery, and security/privacy. In this design alternative, there is only one copy of each fragment in the system (nonreplicated fragments). We will explain how each of the above fragmentation schemes works in Section 2.2.

***2.1.2.4 Partially Replicated*** In this distribution alternative, the designer will make copies of some of the tables (or fragments) in the database and store these copies at different sites. This is based on the belief that the frequency of accessing database tables is not uniform. For example, perhaps Fragment 1 of the EMP table might be accessed more frequently than Fragment 2 of the table. To satisfy this requirement, the designer may decide to store only one copy of Fragment 2, but more than one copy of Fragment 1 in the system. Again, the number of Fragment 2 copies needed depends on how frequently these access queries run and where these access queries are generated.

***2.1.2.5 Mixed Distribution*** In this design alternative, we fragment the database as desired, either horizontally or vertically, and then partially replicate some of the fragments.

## 2.2 FRAGMENTATION

As outlined earlier, fragmentation requires a table to be divided into a set of smaller tables called fragments. Fragmentation can be **horizontal, vertical**, or **hybrid** (a mix of horizontal and vertical). Horizontal fragmentation can further be classified into two classes: **primary horizontal fragmentation (PHF)** and **derived horizontal fragmentation (DHF)**. When thinking about fragmentation, designers need to decide on the degree of granularity for each fragment. In other words, how many of the table columns and/or rows should be in a fragment? The range of options is vast. At one end, we can have all the rows and all the columns of the table in one fragment. This obviously gives us a nonfragmented table; the grain is too coarse if we were planning to have at least one fragment. At the other end, we can put each data item (a single column value for a single row) in a separate fragment. This grain obviously is too fine: it would be hard to manage and would add too much overhead to processing queries. The answer should be somewhere in between these two extremes. As we will explain later, the optimal solution depends on the type and frequency of queries that applications run against the table. In the rest of this section, we explore each fragmentation type and formalize the fragmentation process.

### 2.2.1 Vertical Fragmentation

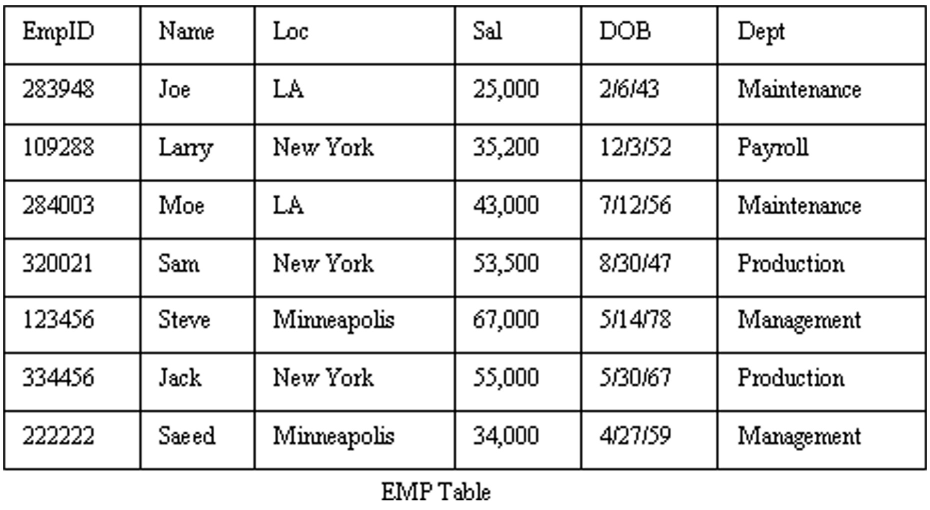
Vertical fragmentation (VF) will group the columns of a table into fragments. VF must be done in such a way that the original table can be reconstructed from the fragments. This fragmentation requirement is called “reconstructiveness.” This requirement is used to reconstruct the original table when needed. As a result, each VF fragment must contain the primary key column(s) of the table. Because each fragment contains a subset of the total set of columns in the table, VF can be used to enforce security and/or privacy of data. To create a vertical fragment from a table, a select statement is used in which “Column_list” is a list of columns from R that includes the primary key.

```
Select Column_list from R;
```

**Example 2.1** Consider the EMP table shown in Figure 2.3. Let’s assume that for security reasons the salary information for employees needs to be maintained in the company headquarters’ server, which is located in Minneapolis.

To achieve this, the designer will fragment the table vertically into two fragments as follows:

```
Create table EMP_SAL as
     Select EmpID, Sal
     From EMP;
```



| EmpID | Name | Loc | Sal | DOB | Dept |
|---|---|---|---|---|---|
| 283948 | Joe | LA | 25,000 | 2/6/43 | Maintenance |
| 109288 | Larry | New York | 35,200 | 12/3/52 | Payroll |
| 284003 | Moe | LA | 43,000 | 7/12/56 | Maintenance |
| 320021 | Sam | New York | 53,500 | 8/30/47 | Production |
| 123456 | Steve | Minneapolis | 67,000 | 5/14/78 | Management |
| 334456 | Jack | New York | 55,000 | 5/30/67 | Production |
| 222222 | Saeed | Minneapolis | 34,000 | 4/27/59 | Management |

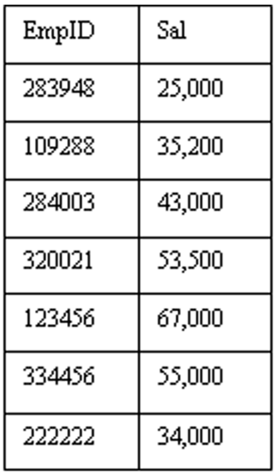
EMP Table

**Figure 2.3** The nonfragmented version of the EMP table.

```
Create table EMP_NON_SAL as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP;
```

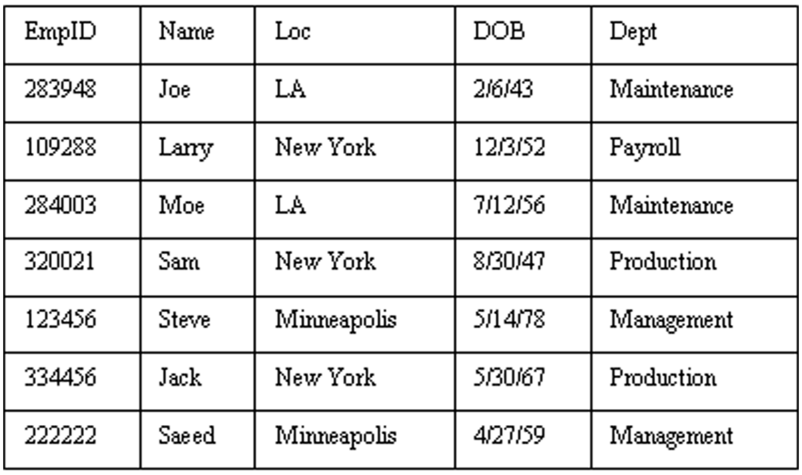
EMP_SAL contains the salary information for all employees while EMP_NON_SAL contains the nonsensitive information. These statements generate the vertical fragments shown in Figure 2.4a, 2.4b from the EMP table.

After fragmentation, the EMP table will not be stored physically anywhere. But, to provide for fragmentation transparency—not requiring the users to know that the EMP table is fragmented—we have to be able to reconstruct the EMP table from its VF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following join statement anywhere the EMP table is required:



| EmpID | Sal |
|---|---|
| 283948 | 25,000 |
| 109288 | 35,200 |
| 284003 | 43,000 |
| 320021 | 53,500 |
| 123456 | 67,000 |
| 334456 | 55,000 |
| 222222 | 34,000 |

**(a)** EMP_Sal Fragment



| EmpID | Name | Loc | DOB | Dept |
|---|---|---|---|---|
| 283948 | Joe | LA | 2/6/43 | Maintenance |
| 109288 | Larry | New York | 12/3/52 | Payroll |
| 284003 | Moe | LA | 7/12/56 | Maintenance |
| 320021 | Sam | New York | 8/30/47 | Production |
| 123456 | Steve | Minneapolis | 5/14/78 | Management |
| 334456 | Jack | New York | 5/30/67 | Production |
| 222222 | Saeed | Minneapolis | 4/27/59 | Management |

**(b)** EMP_NON_Sal Fragment

**Figure 2.4** The vertical fragments of the EMP table.

```
Select EMP_SAL.EmpID, Sal, Name, Loc, DOB, Dept
From EMP_SAL, EMP_NON_SAL
Where EMP_SAL.EmpID = EMP_NON_SAL.EmpID;
```

**Note:** This join statement can be used in defining a view called "EMP" and/or can be used as an in-line view in any select statement that uses the virtual (physically nonexisting) table "EMP."

### 2.2.2 Horizontal Fragmentation

Horizontal fragmentation (HF) can be applied to a base table or to a fragment of a table. Note that a fragment of a table is itself a table. Therefore, in the following discussion when we use the term table, we might refer to a base table or a fragment of the table. HF will group the rows of a table based on the values of one or more columns. Similar to vertical fragmentation, horizontal fragmentation must be done in such a way that the base table can be reconstructed (reconstructiveness). Because each fragment contains a subset of the rows in the table, HF can be used to enforce security and/or privacy of data. Every horizontal fragment must have all columns of the original base table. To create a horizontal fragment from a table, a select statement is used. For example, the following statement selects the row from R satisfying condition C:

```
Select * from R where C;
```

As mentioned earlier, there are two approaches to horizontal fragmentation. One is called primary horizontal fragmentation (PHF) and the other is called derived horizontal fragmentation (DHF).

***2.2.2.1 Primary Horizontal Fragmentation*** Primary horizontal fragmentation (PHF) partitions a table horizontally based on the values of one or more columns of the table. Example 2.2 discusses the creation of three PHF fragments from the EMP table based on the values of the Loc column.

**Example 2.2** Consider the EMP table shown in Figure 2.3. Suppose we have three branch offices, with each employee working at only one office. For ease of use, we decide that information for a given employee should be stored in the DBMS server at the branch office where that employee works. Therefore, the EMP table needs to be fragmented horizontally into three fragments based on the value of the Loc column as shown below:

```
Create table MPLS_EMPS as
     Select *
     From EMP
     Where Loc = 'Minneapolis';

Create table LA_EMPS as
     Select *
     From EMP
     Where Loc = 'LA';
```

```
Create table NY_EMPS as
     Select *
     From EMP
     Where Loc = 'New York';
```

This design generates three fragments, shown in Figure 2.5a,b,c. Each fragment can be stored in its corresponding city's server.

Again, after fragmentation, the EMP table will not be physically stored anywhere. To provide for horizontal fragmentation transparency, we have to be able to reconstruct the EMP table from its HF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following union statement anywhere the EMP table is required:

```
(Select * from MPLS_EMPS
Union
Select * from LA_EMPS)
     Union
     Select * from NY_EMPS;
```

| EmpID | Name | Loc | Sal | DOB | Dept |
|---|---|---|---|---|---|
| 123456 | Steve | Minneapolis | 67,000 | 5/14/78 | Management |
| 222222 | Saeed | Minneapolis | 34,000 | 4/27/59 | Management |

**(a)** MPLS_EMPS fragment

| EmpID | Name | Loc | Sal | DOB | Dept |
|---|---|---|---|---|---|
| 283948 | Joe | LA | 25,000 | 2/6/43 | Maintenance |
| 284003 | Moe | LA | 43,000 | 7/12/56 | Maintenance |

**(b)** LA_EMPS fragment

| EmpID | Name | Loc | Sal | DOB | Dept |
|---|---|---|---|---|---|
| 109288 | Larry | New York | 35,200 | 12/3/52 | Payroll |
| 320021 | Sam | New York | 53,500 | 8/30/47 | Production |
| 334456 | Jack | New York | 55,000 | 5/30/67 | Production |

**(c)** NY_EMPS fragment

**Figure 2.5** The horizontal fragments of the EMP table with fragments based on Loc.

***2.2.2.2 Derived Horizontal Fragmentation*** Instead of using PHF, a designer may decide to fragment a table according to the way that another table is fragmented. This type of fragmentation is called derived horizontal fragmentation (DHF). DHF is usually used for two tables that are naturally (and frequently) joined. Therefore, storing corresponding fragments from the two tables at the same site will speed up the join across the two tables. As a result, an implied requirement of this fragmentation design is the presence of a join column across the two tables.

**Example 2.3** Figure 2.6a shows table "DEPT(Dno, Dname, Budget, Loc)," where Dno is the primary key of the table. Let's assume that DEPT is fragmented based on the department's city. Applying PHF to the DEPT table generates three horizontal fragments, one for each of the cities in the database, as depicted in Figure 2.6b,c,d.

Now, let's consider the table "PROJ," as depicted in Figure 2.7a. We can partition the PROJ table based on the values of Dno column in the DEPT table's fragments with the following SQL statements. These statements will produce the derived fragments from the PROJ table as shown in Figure 2.7b,c. Note that there are no rows in PROJ3, since department "D4" does not manage any project.

| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D1 | Management | 750,000 | Minneapolis |
| D2 | Payroll | 500,000 | New York |
| D3 | Production | 400,000 | New York |
| D4 | Maintenance | 300,000 | LA |

**(a)** DEPT Table

| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D1 | Management | 750,000 | Minneapolis |

**(b)** MPLS_DEPTS Fragment

| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D2 | Payroll | 500,000 | New York |
| D3 | Production | 400,000 | New York |

**(c)** NY_DEPTS Fragment

| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D4 | Maintenance | 300,000 | LA |

**(d)** LA_DEPTS Fragment

**Figure 2.6** The fragments of the DEPT table with fragments based on Loc.

| Pno | Pname | Budget | Dno |
|---|---|---|---|
| P1 | Database Design | 135,000 | D2 |
| P2 | Maintenance | 310,000 | D3 |
| P3 | CAD/CAM | 500,000 | D2 |
| P4 | Architecture | 300,000 | D1 |
| P5 | Documentation | 450,000 | D1 |

**(a)** PROJ Table

| Pno | Pname | Budget | Dno |
|---|---|---|---|
| P4 | Architecture | 300,000 | D1 |
| P5 | Documentation | 450,000 | D1 |

**(b)** PROJ1

| Pno | Pname | Budget | Dno |
|---|---|---|---|
| P1 | Database Design | 135,000 | D2 |
| P3 | CAD/CAM | 500,000 | D2 |
| P2 | Maintenance | 310,000 | D3 |

**(c)** PROJ2

**Figure 2.7** The PROJ table and its component DHF fragments.

```
Create table PROJ1 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, MPLS_DEPTS
     Where PROJ.Dno = MPLS_DEPTS.Dno;

Create table PROJ2 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, NY_DEPTS
     Where PROJ.Dno = NY_DEPTS.Dno;

Create table PROJ3 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, LA_DEPTS
     Where PROJ.Dno = LA_DEPTS.Dno;
```

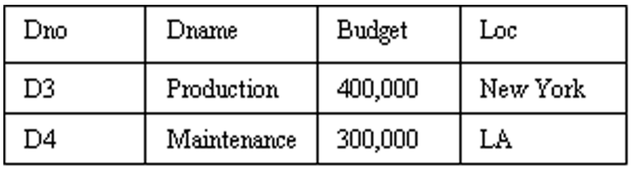
It should be rather obvious that all the rows in PROJ1 have corresponding rows in the MPLS_DEPTS fragment, and similarly, all the rows in PROJ2 have corresponding rows in the NY_DEPTS fragment. Storing a derived fragment at the same database server where the deriving fragment is, will result in better performance since any join across the two tables' fragments will result in a 100% hit ratio (all rows in one fragment have matching rows in the other).

**Example 2.4** For this example, assume that sometimes we want to find those projects that are managed by the departments that have a budget of less than or equal to 500,000 (department budget, not project budget) and at other times we want to find those projects that are managed by the departments that have a budget of more than 500,000. In order to achieve this, we fragment DEPT based on the budget of the department. All departments with a budget of less than or equal to 500,000 are stored in DEPT4 and other departments are stored in the DEPT5 fragment. Figures 2.8a and 2.8b show DEPT4 and DEPT5, respectively.

To easily answer the type of questions that we have outlined in this example, we should create two derived horizontal fragments of the PROJ table based on DEPT4 and DEPT5 as shown below.

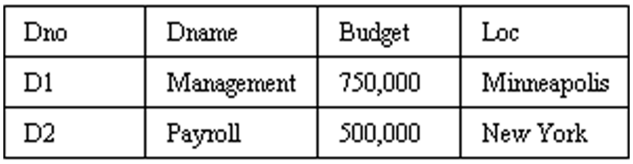
```
Create table PROJ5 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, DEPT4
     Where PROJ.Dno = DEPT4.Dno;

Create table PROJ6 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, DEPT5
     Where PROJ.Dno = DEPT5.Dno;
```

Figure 2.9 shows the fragmentation of the PROJ table based on these SQL statements.



| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D3 | Production | 400,000 | New York |
| D4 | Maintenance | 300,000 | LA |

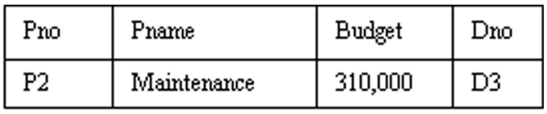
(a) DEPT4



| Dno | Dname | Budget | Loc |
|---|---|---|---|
| D1 | Management | 750,000 | Minneapolis |
| D2 | Payroll | 500,000 | New York |

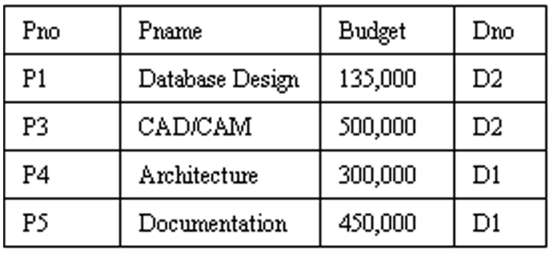
(b) DEPT5

**Figure 2.8** The DEPT table fragmented based on Budget column values.



| Pno | Pname | Budget | Dno |
|---|---|---|---|
| P2 | Maintenance | 310,000 | D3 |

**(a)** PROJ5



| Pno | Pname | Budget | Dno |
|---|---|---|---|
| P1 | Database Design | 135,000 | D2 |
| P3 | CAD/CAM | 500,000 | D2 |
| P4 | Architecture | 300,000 | D1 |
| P5 | Documentation | 450,000 | D1 |

**(b)** PROJ6

**Figure 2.9** The derived fragmentation of PROJ table based on the fragmented DEPT table.

### 2.2.3 Hybrid Fragmentation

Hybrid fragmentation (HyF) uses a combination of horizontal and vertical fragmentation to generate the fragments we need. There are two approaches to doing this. In the first approach, we generate a set of horizontal fragments and then vertically fragment one of more of these horizontal fragments. In the second approach, we generate a set of vertical fragments and then horizontally fragment one or more of these vertical fragments. Either way, the final fragments produced are the same. This fragmentation approach provides for the most flexibility for the designers but at the same time it is the most expensive approach with respect to reconstruction of the original table.

**Example 2.5** Let’s assume that employee salary information needs to be maintained in a separate fragment from the nonsalary information as discussed above. A vertical fragmentation plan will generate the EMP_SAL and EMP_NON_SAL vertical fragments as explained in Example 2.1. The nonsalary information needs to be fragmented into horizontal fragments, where each fragment contains only the rows that match the city where the employees work. We can achieve this by applying horizontal fragmentation to the EMP_NON_SAL fragment of the EMP table. The following three SQL statements show how this is achieved.

```
Create table NON_SAL_MPLS_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = ‘Minneapolis’;

Create table NON_SAL_LA_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = ‘LA’;
```

```
Create table NON_SAL_NY_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = 'New York';
```

The final distributed database is depicted in Figure 2.10.

**Observation:** The temporary EMP_NON_SAL fragment is not physically stored anywhere in the system after it has been horizontally fragmented. As a result, one can bypass generating this fragment by using the following set of SQL statements to generate the required fragments directly from the EMP table.

| EmpID | Sal |
|---|---|
| 283948 | 25,000 |
| 109288 | 35,200 |
| 284003 | 43,000 |
| 320021 | 53,500 |
| 123456 | 67,000 |
| 334456 | 55,000 |
| 222222 | 34,000 |

EMP_Sal

| EmpID | Name | Loc | DOB | Dept |
|---|---|---|---|---|
| 123456 | Steve | Minneapolis | 5/14/78 | Management |
| 222222 | Saeed | Minneapolis | 4/27/59 | Management |

NON_Sal_MPLS_EMPS

| EmpID | Name | Loc | DOB | Dept |
|---|---|---|---|---|
| 283948 | Joe | LA | 2/6/43 | Maintenance |
| 284003 | Moe | LA | 7/12/56 | Maintenance |

NON_Sal_LA_EMPS

| EmpID | Name | Loc | DOB | Dept |
|---|---|---|---|---|
| 109288 | Larry | New York | 12/3/52 | Payroll |
| 320021 | Sam | New York | 8/30/47 | Production |
| 334456 | Jack | New York | 5/30/67 | Production |

NON_Sal_NY_EMPS

**Figure 2.10** The fragments of the EMP table.

```
Create table NON_SAL_MPLS_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = ‘Minneapolis’;

Create table NON_SAL_LA_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = ‘LA’;

Create table NON_SAL_NY_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = ‘New York’;
```

### 2.2.4 Vertical Fragmentation Generation Guidelines

There are two approaches to vertical fragmentation design—grouping and splitting—proposed in the literature [Hoffer75] [Hammer79] [Sacca85]. In the remainder of this section, we will first provide an overview of these two options and then present more detail for the splitting option.

***2.2.4.1 Grouping*** Grouping is an approach that starts by creating as many vertical fragments as possible and then incrementally reducing the number of fragments by merging the fragments together. Initially, we create one fragment per nonkey column, placing the nonkey column and the primary key of the table into each vertical fragment. This first step creates as many vertical fragments as the number of nonkey columns in the table. Most of the time, this degree of fragmentation is too fine and impractical. The grouping approach uses joins across the primary key, to group some of these fragments together, and we continue this process until the desired design is achieved. Aside from needing to fulfill the requirements for one or more application, there are very few restrictions placed upon the groups (fragments) we create in this approach. For example, the same nonkey column can participate in more than one group—that is, groups can have overlapping (nonkey) columns. If this “overlap” does occur, obviously, it will add to the overhead of replication control in a distributed DBMS system. As a result, grouping is not usually considered a valid approach for vertical fragmentation design. For more details on grouping see [Hammer79] and [Sacca85]. Hammer and Niamir introduced grouping for centralized DBMSs and Sacca and Wiederhold discussed grouping for distributed DBMSs.

***2.2.4.2 Splitting*** Splitting is essentially the opposite of grouping. In this approach, a table is fragmented by placing each nonkey column in one (and only one) fragment, focusing on identifying a set of required columns for each vertical fragment. As such, there is no overlap of nonprimary key columns in the vertical fragments that are created using splitting. Hoffer and Severance [Hoffer75] first introduced splitting for centralized systems, while Navathe and colleagues [Navathe84] introduced splitting for distributed systems. There is general consensus that finding an ideal vertical fragmentation design—one that satisfies the requirements for a large set of applications—is not feasible. In an ideal vertical fragment design, each application would only need to access the columns in one vertical fragment. If certain sets of columns are always processed together by the application, the process used to create this design is trivial. But real-life applications do not always behave as we wish. Hence, for a database that contains many tables with many columns, we need to develop a systematic approach for defining our vertical fragmentation.

As pointed out by Hammer and Niamir [Hammer79], there is a direct correlation between the number of columns in a table and the number of the possible vertical fragmentation options. A table with $m$ columns can be vertically partitioned into $B(m)$ different alternatives, where $B(m)$ is the $m$th Bell number. For large $m$, $B(m)$ approaches $m^m$. For example, if a table has 15 columns, then the number of possible vertical fragments is $10^9$ and the number of vertical fragments for a table with 30 columns is $10^{23}$. Obviously, evaluating $10^9$ (or 1,000,000,000) alternatives for a 15-column table is not practical. Instead of evaluating all possible vertical fragments, designers can use the metric **affinity** or **closeness** of columns to each other to decide whether or not a group of columns should be put in the same fragment. The affinity of columns expresses the extent to which they are used together in processing. By combining the access frequency of applications to columns of a table with the usage pattern of these applications, one can create the affinity matrix that forms the basis for vertically fragmenting the table.

*Splitting in Distributed Systems* In this section, we will outline the proposal by Navathe and colleagues [Navathe84] in conjunction with an example. For further details about the approach, the reader should see the referenced publication.

**Example 2.6** Consider applications "AP1", "AP2", "AP3", and "AP4" as shown. These applications work on the table "T" defined as "T($\underline{C}$, C1, C2, C3, C4)," where C is the primary key column of the table.

```
AP1: Select C1 from T where C4 = 100;
AP2: Select C4 from T;
AP3: Update T set C3 = 15 where C2 = 50;
AP4: Update T set C1 = 5 where C3 = 10;
```

Usage Matrix For a single site system, these applications are local and will have the usage matrix depicted in Figure 2.11. As can be seen, the usage matrix is a two-dimensional matrix that indicates whether or not an attribute (column) is used by an application. The cell in position (APi, Cj) is set to 1 if application "APi" accesses column "Cj," otherwise it is set to 0.

**Observation 1:** The **usage matrix** only indicates if a column is used by an application. However, the matrix does not show how many times an application accesses the table columns during a given time period. The **access frequency** is a term that represents how many times an application runs in a given period of time. The period of this measurement can be an hour, a day, a week, a month, and so on—as decided by the designer. Figure 2.12 depicts the expansion of the usage matrix to include the access frequency of each application.

| | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| AP1 | 1 | 0 | 0 | 1 |
| AP2 | 0 | 0 | 0 | 1 |
| AP3 | 0 | 1 | 1 | 0 |
| AP4 | 1 | 0 | 1 | 0 |

**Figure 2.11** Single site usage matrix for Example 2.6.

| | C1 | C2 | C3 | C4 | Access Frequency |
|---|---|---|---|---|---|
| AP1 | 1 | 0 | 0 | 1 | 3 |
| AP2 | 0 | 0 | 0 | 1 | 7 |
| AP3 | 0 | 1 | 1 | 0 | 4 |
| AP4 | 1 | 0 | 1 | 0 | 3 |

**Figure 2.12** Expansion of usage matrix.

**Observation 2:** Neither the usage matrix nor the access frequencies have any indication of distribution. In a distributed system, however, an application can have different frequencies at different sites. For example, in a four-site system, AP2 might run four times at S2 and three times at S3. It is also possible that AP2 might run seven times at site S2 and zero times everywhere else. In both cases, the frequency would still be shown as seven in the usage matrix. Also, each time the application runs at a site it might make more than one access to the table (and its columns). For example, suppose we had another application, AP5, defined as follows:

```
Begin AP5
     Select C1 from T where C4 = 100;
     Select C4 from T;
End AP1;
```

In this case, AP5 makes two references to T each time it runs. As a result, the actual access frequency for AP5 is calculated as "ACC(Pi) * REF (Pi)," where ACC(Pi) is the number of times the application runs and REF (Pi) is the number of accesses Pi makes to T every time it runs. To simplify the discussion, we assume "REF(Pi) = 1" for all processes, which results in "ACC(Pi) * REF(Pi) = ACC(Pi)."

If we include the access frequencies of the applications at each site for our original example (without AP5), this makes the usage matrix a three-dimensional matrix, where in the third axis we maintain the frequency of each application for each site. Since such

| | C1 | C2 | C3 | C4 | Access Frequency | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | S1 | S2 | S3 | S4 |
| AP1 | 1 | 0 | 0 | 1 | 1 | 0 | 2 | 0 |
| AP2 | 0 | 0 | 0 | 1 | 0 | 4 | 3 | 0 |
| AP3 | 0 | 1 | 1 | 0 | 0 | 0 | 4 | 0 |
| AP4 | 1 | 0 | 1 | 0 | 3 | 0 | 0 | 0 |

**Figure 2.13** Access frequencies for a distributed system.

representation is difficult to display on two-dimensional paper, we flatten the matrix as shown in Figure 2.13.

By adding access frequencies for each application across all sites, we can get the affinity or closeness that each column has with the other columns referenced by the same application. This matrix is called the **process column affinity matrix**, and it is shown for this example in Figure 2.14.

The first row in this matrix shows the affinity of C1 to C4 (or C4 to C1) for AP1 being 3. Similarly, the third row indicates the affinity of C2 to C3 (or C3 to C2) for AP3 being 4; C1 to C3 (or C3 to C1) for AP4 being 3; and so on. Note that since AP2 only uses C4, for AP2 there is no affinity between C4 and other columns. In order to capture these affinities, we can remove the application names from the matrix and create a two-dimensional matrix as shown in Figure 2.15. This matrix shows the affinity of each column of the table with other column(s) regardless of the applications that use them. We call this the **affinity matrix**.

Notice in Figure 2.15 that the diagonal cells—cells in position "(i,i)" for "i = 1 to 4"—have value 0. Since the diagonal cells do not store any values, we can calculate the affinity of column Ci with respect to all the other columns and store this value in "Cell(i,i)" in the matrix. C1 has an affinity with C3 weighted at 3 and an affinity with C4 weighted at 3. Therefore, the affinity of C1 across all applications at all sites is found by summing its affinity weights with all the other columns, "$3 + 3 = 6$." We will insert this total affinity value in "Cell(1,1)." Similarly, C3 has two affinities—one with C1 weighted at 3 and one with C2 weighted at 4. C3 has the total affinity value of

| | C1 | C2 | C3 | C4 | Affinity |
|---|---|---|---|---|---|
| AP1 | 1 | 0 | 0 | 1 | 3 |
| AP2 | 0 | 0 | 0 | 1 | 7 |
| AP3 | 0 | 1 | 1 | 0 | 4 |
| AP4 | 1 | 0 | 1 | 0 | 3 |

**Figure 2.14** Process column affinity matrix.

| | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| C1 | 0 | 0 | 3 | 3 |
| C2 | 0 | 0 | 4 | 0 |
| C3 | 3 | 4 | 0 | 0 |
| C4 | 3 | 0 | 0 | 0 |

**Figure 2.15** The affinity matrix.

| | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| C1 | 6 | 0 | 3 | 3 |
| C2 | 0 | 4 | 4 | 0 |
| C3 | 3 | 4 | 7 | 0 |
| C4 | 3 | 0 | 0 | 3 |

**Figure 2.16** Affinity matrix with diagonal values calculated.

"$3 + 4 = 7$" for all applications across all sites, which we insert in "Cell(3,3)." Adding up all affinities of each column (summing across the rows), we can calculate the total affinity for each column, storing them in the diagonal cells. Figure 2.16 shows the results. In the next section, we will discuss an algorithm that generates our vertical fragmentation design based on the affinity information contained in this matrix.

The Bond Energy Algorithm The information in the affinity matrix shown in Figure 2.16 can be used to cluster the columns of T together. McCormick and colleagues suggest using the **Bond Energy Algorithm (BEA)** [McCormick72] for this. This algorithm takes the affinity matrix as an input parameter and generates a new matrix called the **clustered affinity matrix** as its output. The clustered affinity matrix is a matrix that reorders the columns and rows of the affinity matrix so that the columns with the greatest affinity for each other are "grouped together" in the same cluster—which is then used as the basis for splitting our table into vertical fragments. Details of this algorithm are outside the scope of this chapter. In what follows, we will only discuss the steps in the algorithm for our example.

*Step 1: Placement of the First Two Columns*. In this step, we place the first two columns of the affinity matrix into a new matrix (the clustered affinity matrix). In our example, the first two columns are C1 and C2. According to McCormick and colleagues, the bond energy of any two columns such as "X1" and "X2"—shown as "Bond(X1, X2)"—is the row-wise sum of the product of the affinity values for X1 and X2, taken from the affinity matrix. Figure 2.17 shows this bond energy calculation for C1 and C2 of our example. In this figure, we created a new matrix

| | C1 | C2 | product |
|---|---|---|---|
| C1 | 6 | 0 | 0 |
| C2 | 0 | 4 | 0 |
| C3 | 3 | 4 | 12 |
| C4 | 3 | 0 | 0 |
| Bond | | | 12 |

**Figure 2.17** The Bond calculation for columns C1 and C2 of Example 2.6.

by first copying the first two columns from the affinity matrix. Next, we augmented this new matrix by adding a new “product” column, where we show the result of the product (multiplication) of the two values in the same row, to the left of the column. For example, the product for row “C3” in this figure is 12, because “$3 * 4 = 12$.” Finally, we augmented it again by including the “Bond” row, which simply stores the sum of all the product column’s values.

*Step 2: Placement of Remaining Columns*. In this step, we place each of the remaining columns (C3 and C4 in our example), one at a time, into the new clustered affinity matrix—which we have not actually shown yet, but since it only contains C1 and C2 we can understand what it looks like without presenting it in a separate figure. First, we will place C3, and then later C4, into the new matrix. The purpose of this step is to determine where each “new” column should be placed. In our example, first we need to decide “where” to add the C3 column from the affinity matrix to the matrix that already contains C1 and C2. For C3, there are three options—to place the C3 affinity information to the left side of C1 (making it the leftmost column in our new matrix), to place the C3 information in between C1 and C2, or to place the C3 affinity information on the right side of C2 (making it the rightmost column in the new table). We then need to calculate the contribution values of all these possible placements of the C3 affinity information, using the formula given in Equation 2.1. Once all contribution values are calculated, we choose the ordering that provides the highest contribution to the affinity measure.

### Bond Contribution Calculation

$$\texttt{Cont(X1, X2, X3)} = 2 * \texttt{Bond(X1, X2)} + 2 * \texttt{Bond(X2, X3)} - 2 * \texttt{Bond(X1, X3)} \qquad (2.1)$$

In general, when adding an affinity matrix column to the clustered affinity matrix, we must consider the two boundary conditions—leftmost and rightmost positions for the new column. The “X2” in the contribution formula is replaced with the column being considered, while the “X1” is the column to its left and “X3” is the column to its right. However, when we consider placing the new column in the leftmost position, there are no columns to the left of it—therefore, we use a pseudo-column named “C0” to represent this nonexistent column-to-the-left. Similarly, when we consider placing the new column in the rightmost position, we use another pseudo-column named "Cn" to represent the nonexistent column to its right. In our calculations, both C0 and Cn have an affinity of zero.

Therefore, when we consider adding C3 to the clustered affinity matrix for our example, there are three ordering options: "(C3, C1, C2)", "(C1, C3, C2)", or "(C1, C2, C3)." To determine which ordering yields the highest value for our affinity measure, we must calculate the contribution of C3 being added to the left of C1, between C1 and C2 and to the right of C2. Therefore, we need to calculate "Cont(C0, C3, C1)", "Cont(C1, C3, C2)," and "Cont(C2, C3, Cn)" and compare the results.

To calculate the contribution of "(C0, C3, C1)," we need to figure out the bonds for "(C0,C3)," "(C3, C1)," and "(C0, C1)." Once we have determined these bonds, we can use Equation 2.1 to calculate "Cont(C0, C3, C1) = 2∗ Bond(C0, C3) +2∗

| | C0 | C3 | product |
|---|---|---|---|
| C1 | 0 | 3 | 0 |
| C2 | 0 | 4 | 0 |
| C3 | 0 | 7 | 0 |
| C4 | 0 | 0 | 0 |
| Bond | | | 0 |

(a)

| | C3 | C1 | product |
|---|---|---|---|
| C1 | 3 | 6 | 18 |
| C2 | 4 | 0 | 0 |
| C3 | 7 | 3 | 21 |
| C4 | 0 | 3 | 0 |
| Bond | | | 39 |

(b)

| | C0 | C1 | product |
|---|---|---|---|
| C1 | 0 | 6 | 0 |
| C2 | 0 | 0 | 0 |
| C3 | 0 | 3 | 0 |
| C4 | 0 | 3 | 0 |
| Bond | | | 0 |

(c)

**Figure 2.18** Bonds for (C0, C3, C1) ordering.

Bond(C3, C1) −2∗ Bond(C0, C1).” Figures 2.18a, 2.18b, and 2.18c depict these bond calculations. As these figures demonstrate, the Bond function will always return zero when either C0 or Cn is passed to it as a parameter, because C0 and Cn always have an affinity of zero.

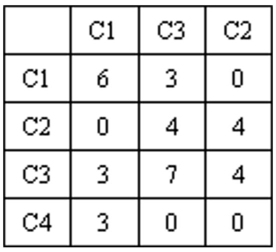
By replacing the Bond functions with their values in the contribution formula, we calculate the contribution of this ordering as “Cont(C0, C3, C1) $= 2 * 0 + 2 * 39 - 2 * 0 = 78$.” Similarly, we can calculate the contributions of orderings “(C1, C3, C2)” and “(C1, C2, C3)” as

```
For ordering (C1, C3, C2):
Bond(C1, C3) = 39
Bond(C3, C2) = 44
Bond(C1, C2) = 12
Cont(C1, C3, C2) = 2*39 + 2*44 - 2*12 = 142

For ordering (C2, C3, Cn):
Bond(C2, C3) = 44
Bond(C3, Cn) = 0
Bond(C2, Cn) = 0
Cont(C1, C2, C3) = 2*44 + 2*0 - 2*0 = 88
```

Based on these calculations, the ordering “(C1, C3, C2)” provides the highest contribution. Using this ordering, Figure 2.19 shows the current definition of the new clustered affinity matrix. Since this matrix does not have C4 in it yet, it is called the partial clustered affinity matrix.

After adding C3 to the matrix, we are ready to add the affinity information for C4 as the final column. Adding C4 to the matrix requires calculating contributions of the orderings “(C0, C4, C1),” “(C1, C4, C3),” “(C3, C4, C2),” and “(C2, C4, Cn).” The following are the Bonds and contributions of these ordering options:

```
For ordering (C0, C4, C1):
Bond(C0, C4) = 0
Bond (C4, C1) = 27
Bond(C0, C1) = 0
Cont(C0, C4, C1) = 2*0 + 2*27 – 2 *0 = 54
```



|  | C1 | C3 | C2 |
|---|---|---|---|
| C1 | 6 | 3 | 0 |
| C2 | 0 | 4 | 4 |
| C3 | 3 | 7 | 4 |
| C4 | 3 | 0 | 0 |

**Figure 2.19** The partial clustered affinity matrix.

```
For ordering (C1, C4, C3):
Bond(C1, C4) = 27
Bond(C4, C3) = 9
Bond(C1, C3) = 39
Cont(C1, C4, C3) = 2*27 + 2*9 – 2*39 = –6

For ordering (C3, C4, C2):
Bond(C3, C4) = 9
Bond(C4, C2) = 0
Bond(C3, C2) = 44
Cont((C3, C4, C2) = 2*9 + 2*0 – 2*44 = –70

For ordering (C2, C4, Cn):
Bond(C2, C4) = 0
Bond(C4, Cn) = 0
Bond(C2, Cn) = 0
Cont((C1, C4, C2) = 2*0 + 2*0 - 2*0 = 0
```

From these contribution calculations, the ordering "(C0, C4, C1)" yields the highest contribution and therefore is chosen. Figure 2.20 depicts the final ordering of affinity information for the columns of table "T" within the clustered affinity matrix.

Once the columns have been put in the right order, we order the rows similarly, ending up with the final clustered affinity matrix as shown in Figure 2.21.

|  | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C1 | 3 | 6 | 3 | 0 |
| C2 | 0 | 0 | 4 | 4 |
| C3 | 0 | 3 | 7 | 4 |
| C4 | 3 | 3 | 0 | 0 |

**Figure 2.20** The clustered affinity matrix after rearranging the columns.

|  | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

**Figure 2.21** The clustered affinity matrix after rearranging the rows and the columns.

*Step 3: Partitioning the Table into Vertical Fragments*. Step 3 is the last step in the process. In this step, we need to split the clustered affinity matrix into a top portion and a bottom portion as shown in Figure 2.22.

Among all applications that access this table, some access only the columns in the top corner (TC), some access only the columns in the Bottom Corner (BC), and the rest access columns in both corners (BOC). We consider ($n - 1$) possible locations of the X point along the diagonal, where $n$ is the size of the matrix (also the number of nonkey columns of the table). A nonoverlapping partition is obtained by selecting X such that the goal function "Z" as defined below is maximized.

Goal Function: Z = TCW * BCW – $\text{BOCW}^2$

In Z, TCW is the total number of accesses by applications to columns in TC (**only** the columns in the top corner), BCW is the total number of accesses by applications to columns in BC (**only** the columns in the bottom corner), and BOCW is the total number of accesses by applications to columns in both quadrants (must access **at least one** column from TC and **at least one** column from BC).The partitioning that corresponds to the maximal value of Z is accepted if Z is positive and is rejected otherwise. This formula is based on the belief that "good" partitioning strategies increase the values of TCW and BCW while decreasing the value of BOCW. In order words, the approach tries to maximize the product TCW * BCW. This also results in the selection of values for TCW and BCW that are as close to equal as possible. Thus, the function will produce fragments that are "balanced." To find the splitting that yields the best partitioning of columns, we have to split the columns into a "one-column BC" and "$n - 1$ column TC" first. We calculate the value of Z for this starting point and then repeatedly add columns from TC to BC until TC is left with only one column. From these calculations, we choose the splitting that has the highest Z value. Recall the affinity matrix for our example (shown here again in

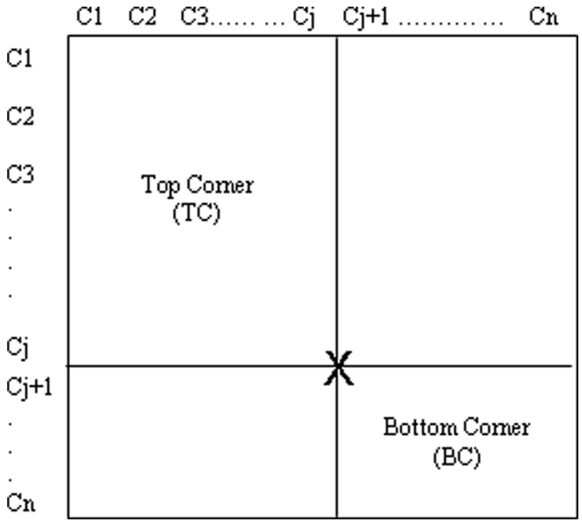


**Figure 2.22** Partitioning of the clustered affinity matrix.

Figure 2.23a). As shown in Figure 2.23b, we begin partitioning the table by putting C4, C1, and C3 in TC and C2 in BC. In this figure, the "X icon" indicates the upper left corner of BC and the bottom right corner of TC.

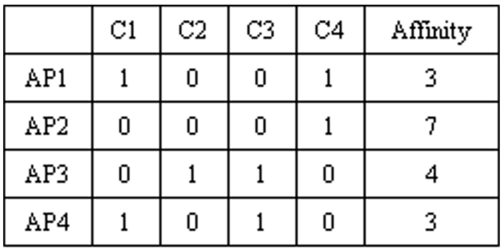
The TC calculation will include the affinity value for all the applications that access one of the TC columns (C4, C1, or C3) but do not access any BC columns (C2). In this case, applications AP1, AP2, and AP4 access TC-only columns, AP3 accesses both TC and BC columns, and no application is BC-only. We use the function "AFF(a)" to represent the affinity for application "a." In other words, in our example, this is a short-hand notation for looking at the value for the affinity column in Figure 2.23a for the row corresponding to application "a." For this splitting option in our example, we calculate Z as follows:

`TCW = AFF(AP1) + AFF(AP2) + AFF(AP4) = 3 + 7 + 3 = 13`
`BCW = none = 0`
`BOCW = AFF(AP3) = 4`
`Z = 13*0 –` $4^2$ `= –16`

In the next step, the splitting point is going to move to the middle of the matrix, leaving the TC and BC each with two columns as shown in Figure 2.24. Now, AP1 and AP2 remain the same, but AP3 uses BC-only columns, and AP4 uses columns that are from both-corners.

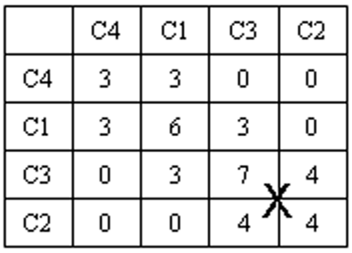
For this splitting option, the value of Z is calculated as follows:

```
TCW = AFF(AP1) + AFF(AP2) = 3 + 7 = 10
BCW = AFF(AP3) = 4
```



| | C1 | C2 | C3 | C4 | Affinity |
|---|---|---|---|---|---|
| AP1 | 1 | 0 | 0 | 1 | 3 |
| AP2 | 0 | 0 | 0 | 1 | 7 |
| AP3 | 0 | 1 | 1 | 0 | 4 |
| AP4 | 1 | 0 | 1 | 0 | 3 |

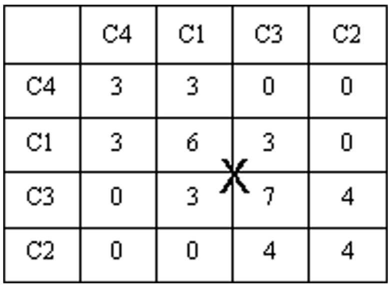
(a)



| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

(b)

**Figure 2.23** The clustered affinity matrix partitioning starting point.



| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

**Figure 2.24** Matrix for second split point.

```
BOCW = AFF(AP4) = 3
```
Z = 4*10 – $3^2$ = 40 – 9 = 31

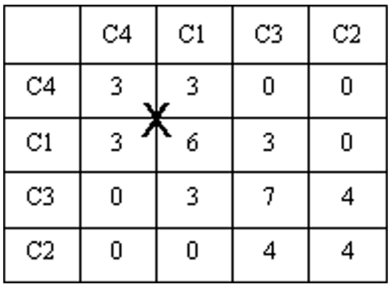
In the next step, the splitting point is going to move to the upper left corner as shown in Figure 2.25. Now, AP2 is still using only top-corner columns, but AP3 and AP4 are using bottom-only columns and AP1 is using columns from both-corners.

For this splitting option we have the following:

```
TCW = AFF(AP2) = 7
BCW = AFF(AP3) + AFF(AP4) = 4 + 3 = 7
BOCW = AFF(AP1) = 3
```
Z = 7*7 – $3^2$ = 49 – 9 = 40

By now, it should be obvious that this approach creates nonoverlapping partitions by moving along the matrix diagonal. The proposed algorithm has the disadvantage of not being able to carve out an embedded or inner block of columns as a partition. To be able to do this, the approach has to place { C1, C3 } in one partition and { C4, C2 } in another partition as shown in Figure 2.26.

We can overcome this disadvantage by adding a SHIFT operation. When the SHIFT operation is utilized, it moves the topmost row of the matrix to the bottom and then it moves the leftmost column of the matrix to the extreme right. Figure 2.27a depicts the original matrix, along with an arrow indicating where the topmost row is supposed to be moved as part of the SHIFT. Figure 2.27b shows the

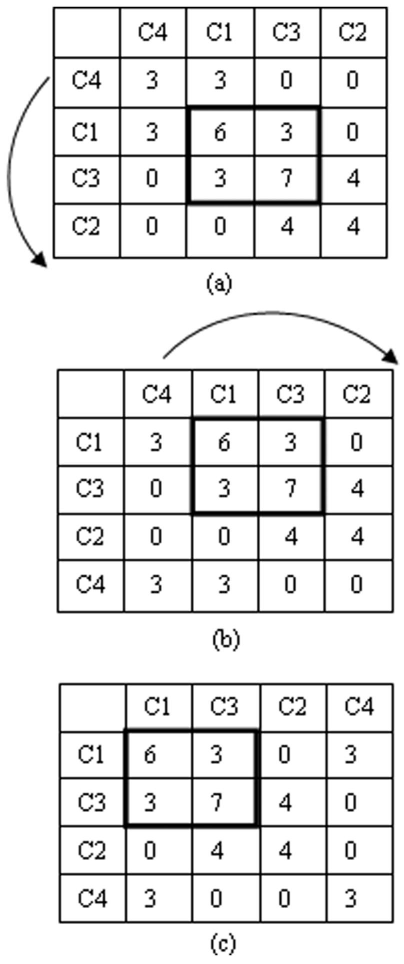


| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

**Figure 2.25** Matrix for the third split point.

| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

**Figure 2.26** Matrix for inner block of columns.



(a)

| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C4 | 3 | 3 | 0 | 0 |
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |

(b)

| | C4 | C1 | C3 | C2 |
|---|---|---|---|---|
| C1 | 3 | 6 | 3 | 0 |
| C3 | 0 | 3 | 7 | 4 |
| C2 | 0 | 0 | 4 | 4 |
| C4 | 3 | 3 | 0 | 0 |

(c)

| | C1 | C3 | C2 | C4 |
|---|---|---|---|---|
| C1 | 6 | 3 | 0 | 3 |
| C3 | 3 | 7 | 4 | 0 |
| C2 | 0 | 4 | 4 | 0 |
| C4 | 3 | 0 | 0 | 3 |

**Figure 2.27** Matrix with the final split after the SHIFT operation.

matrix after the rotation of the topmost row, with another arrow indicating where the leftmost columns are supposed to be moved as part of the same SHIFT operation. Figure 2.27c shows the matrix after the rightmost column is rotated. The SHIFT process is repeated a total of *n* times, so that every diagonal block gets the opportunity of being brought to the upper left corner in the matrix. For our example, we only need to utilize SHIFT once since all other combinations have already been covered.

After the SHIFT, AP4 is using TC-only columns, AP2 is using BC-only columns, and AP1 and AP3 are using columns from both-corners. For this splitting option we have the following:

```
TCW = AFF(AP4) = 3
BCW = AFF(AP2) = 7
BOCW = AFF(AP1) + AFF(AP3) = 3 + 4 = 7
```
$Z = 3*7 - 7^2 = 21 - 49 = -28$

Among all the positive Z values, the maximum value is 40. This value corresponds to the option creating two partitions "(C4)" and "(C1, C2, C3)." As mentioned before, we need to include the primary key column of the T table in every vertical fragment. As a result, the two vertical fragments will be defined as "VF1($\underline{C}$, C4)" and "VF2($\underline{C}$, C1, C2, C3)."

**Final Note:** We have only discussed the BEA for splitting the table into nonoverlapping vertical fragments (the primary key is not considered an overlap). Readers are referred to [Navathe84] and [Hammer79] for information about overlapping vertical fragments.

### 2.2.5 Vertical Fragmentation Correctness Rules

Since the original table is not physically stored in a DDBE, the original table must be reconstructible from its vertical fragments using a combination of some SQL statements (joins in this case).

As a result, the following requirements must be satisfied when fragmenting a table vertically:

- Completeness—every data item of the table is in, at least, one of the vertical fragments.
- Reconstructiveness—the original table can be reconstructed from the vertical fragments.
- Shared primary key—reconstruction requires that the primary key of the table be replicated in all vertical fragments (to perform joins).

### 2.2.6 Horizontal Fragmentation Generation Guidelines

As explained earlier, applying horizontal fragmentation to a table creates a set of fragments that contain disjoint rows of the table (horizontal disjointness). Horizontal fragmentation is useful because it can group together the rows of a table that satisfy the predicates of frequently run queries. As such, all rows of a given fragment should be in the result set of a query that runs frequently. To take advantage of this, the designer of a distributed database system should store each such horizontal fragment at the site where these types of queries run. This raises the question, "How do we decide which condition or conditions to use when we horizontally fragment a table?" The answer is formalized in [Özsu99] and also in [Bobak96]. We will summarize their discussion in this section and apply their formulation to an example in order to explain the general approach.

To fragment a table horizontally, we use one or more predicates (conditions). There are different types of predicates to choose from when fragmenting the table. For example, we can use **simple predicates**, such as "Sal > 100000" or "DNO = 1." In general, a simple predicate, P, follows the format "Column_Name **comparative_operator** Value." The comparative operator is one of the operators in the set {=, <, >, >=, <=, <>}. We can also use a **minterm predicate**, M, which is defined as a **conjunctive normal form** of simple predicates. For example, the predicates { Salary > 30,000 ^ Location = "LA" } and { Salary > 30,000 ^ Location<> "LA" } are minterm predicates. Here, the " ^ " operator is read as "AND."

The set of all simple predicates used by all applications that query a given table is shown as "Pr = {p1, p2, . . . , pn}." The set of all minterm predicates used by all applications that query the table is shown as " M = {m1, m2, . . . , mk}." Applying the minterm predicates M to the table generates k minterm horizontal fragments denoted by the set "F = {F1, F2, . . . , Fk}." For this fragmentation design, all rows in Fi, for "i = 1..k," satisfy mi.

***2.2.6.1 Minimality and Completeness of Horizontal Fragmentation*** It should be obvious that the more fragments that exist in a system, the more time the system has to spend in reconstructing the table. As a result, it is important to have a minimal set of horizontal fragments. To generate a minimal set of fragments, the following rules are applied.

*Rule 1.* The rows of a table (or a fragment) should be partitioned into at least two horizontal fragments, if the rows are accessed differently by at least one application.

When the successive application of Rule 1 is no longer required, the designer has generated a minimal and complete set of horizontal fragments. Completeness is, therefore, defined as follows.

*Rule 2.* A set of simple predicates, Pr, for a table is complete if and only if, for any two rows within any minterm fragment defined on Pr, the rows have the same probability of being accessed by any application.

**Example 2.7** Suppose application "AP1" queries the table "EMP" (see Figure 2.3), looking for those employees who work in Los Angeles (LA). The set "Pr = {p1: Loc = "LA"}" shows all the required simple predicates used by AP1. Therefore, the set "M = {m1: Loc = "LA", m2: Loc<>"LA"}" is a minimal and complete set of minterm predicates for AP1. M fragments EMP into the following two fragments:

```
Fragment F1: Create table LA_EMPS as
             Select * from EMP
             Where Loc = "LA";
```

```
Fragment F2: Create table NON_LA_EMPS as
             Select * from EMP
             Where Loc <> "LA";
```

If AP1 were to also (in addition to checking the value of Loc) exclude any employee whose salary was less than or equal to 30000, then the set of predicates would no longer be minimal or complete. This additional check would mean that the rows in F1 would be accessed by AP1 differently depending on the salary of each employee in F1. Applying the minimality rule mentioned above, we would need to further fragment the EMP table, if this were the case. The new simple predicates for AP1 would require changing Pr and M to the following:

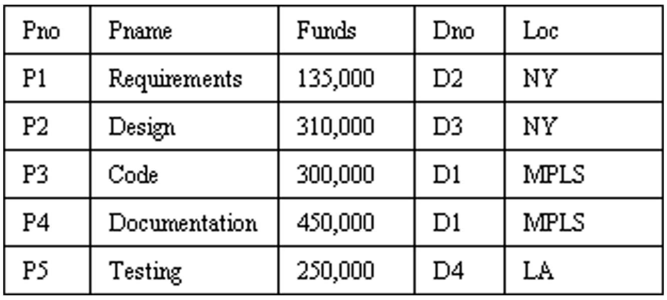
```
Pr = {p1: Loc = "LA",
      p2: salary > 30000}

M = {m1: Loc = "LA"   Sal > 30000,
     m2: Loc = "LA"   Sal <= 30000,
     m3: Loc <>"LA"   Sal > 30000,
     m4: Loc <>"LA"   Sal <= 30000}
```

**Observation 1:** Since there are two simple predicates in Pr, and since for each predicate we have to consider the predicate and its negative, M will now have four minterm predicates in it. In general, if there are $N$ simple predicates in Pr, M will have $2^N$ minterm predicates. As explained later in this section, not all minterm predicates are relevant—the irrelevant predicates must be removed from the set. Therefore, the actual number of horizontal fragments is usually fewer than $2^N$.

**Observation 2:** In forming minterm predicates, we only use conjunctive normal form (“^”) and do not use disjunctive normal form (“OR”). That is because disjunctive normal forms create coarser fragments than fragments generated by conjunctive normal form and hence are not needed.

**Example 2.8** As another example, consider table “PROJ (PNO, Pname, Funds, Dno, Loc),” where the PNO column is the primary key as shown in Figure 2.28.

Also assume two applications (“AP1” and “AP2”) query the PROJ table based on the following set of simple predicates:



| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P1 | Requirements | 135,000 | D2 | NY |
| P2 | Design | 310,000 | D3 | NY |
| P3 | Code | 300,000 | D1 | MPLS |
| P4 | Documentation | 450,000 | D1 | MPLS |
| P5 | Testing | 250,000 | D4 | LA |

**Figure 2.28** The PROJ table.

```
The AP1's simple predicates:
p1: Loc = "MPLS"
p2: Loc = "NY"
p3: Loc = "LA"

The AP2's simple predicates:
p4: Funds <= 300000
```

The two applications have a combined set of four simple predicates in Pr, defined as:

```
Pr = {Loc = "MPLS",
      Loc ="NY",
      Loc ="LA",
      Funds <= 300000}
```

Given the four simple predicates in Pr, M will have "$2^4 = 16$" different minterm predicates. The following depicts all these predicates.

```
m1 = {Loc =  "MPLS" ^  Loc =  "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m2 = {Loc =  "MPLS" ^  Loc =  "NY" ^  Loc =  "LA" ^ Funds >  300000}
m3 = {Loc =  "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m4 = {Loc =  "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds >  300000}
m5 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m6 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds >  300000}
m7 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m8 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds >  300000}
m9 = {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m10= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc =  "LA" ^ Funds >  300000}
m11= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m12= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds >  300000}
m13= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m14= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds >  300000}
m15= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m16= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds >  300000}
```

**Observation 3:** Simple predicates p1, p2, and p3 are mutually exclusive. This means that only p1 or p2 or p3 can be true and not any combination of them. For example, if Loc is set to "MPLS," then it cannot be equal to "NY" or "LA." As a result, m1, m2, m3, m4, m5, m6, m9, and m10 are invalid—we will remove them. This will leave the following eight minterm candidates:

```
m7 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m8 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds >  300000}
m11= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m12= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds >  300000}
m13= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m14= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds >  300000}
m15= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m16= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds >  300000}
```

**Observation 4:** m15 and m16 are invalid predicates, because Loc must have one of the three values mentioned (no blank values, no null values, and no other location values are allowed). After removing m15 and m16, we will have the following six minterm predicates:

```
m7 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m8 = {Loc =  "MPLS" ^  Loc <> "NY" ^  Loc <> "LA" ^ Funds >  300000}
m11= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds <= 300000}
m12= {Loc <> "MPLS" ^  Loc =  "NY" ^  Loc <> "LA" ^ Funds >  300000}
m13= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds <= 300000}
m14= {Loc <> "MPLS" ^  Loc <> "NY" ^  Loc =  "LA" ^ Funds >  300000}
```

**Observation 5:** Since Loc can only have one value at a time, we can simplify the predicates to the following:

```
m7 = {Loc = "MPLS" ^  Funds <= 300000}
m8 = {Loc = "MPLS" ^  Funds >  300000}
m11= {Loc = "NY"   ^  Funds <= 300000}
m12= {Loc = "NY"   ^  Funds >  300000}
m13= {Loc = "LA"   ^  Funds <= 300000}
m14= {Loc = "LA"   ^  Funds >  300000}
```

Applying these minterm predicates to PROJ, we can generate fragments depicted in Figure 2.29. Although there are six minterm predicates, only five of them produce useful results from PROJ. For the current state of the table, only the five fragments shown actually contain rows, the sixth fragment does not. This fragmentation is minimal since rows within each fragment are accessed the same by both applications. This fragmentation is also complete since any two rows within each fragment have the same access probability for AP1 and AP2.

### 2.2.7 Horizontal Fragmentation Correctness Rules

Whenever we use fragmentation (vertical, horizontal, or hybrid), the original table is not physically stored. Therefore, the original table must be reconstructible from its fragments using a combination of SQL statements. Vertically fragmented tables are reconstructed using join operations, horizontally fragmented tables are reconstructed using union operations, and tables fragmented using hybrid fragmentation are reconstructed using a combination of union and join operations.

Any fragmentation must satisfy the following rules as defined by Özsu [Özsu99]:

- *Rule 1: Completeness.* Decomposition of R into R1, R2, ..., Rn is complete if and only if each data item in R can also be found in some Ri.
- *Rule 2: Reconstruction.* If R is decomposed into R1, R2, ..., Rn, then there should exist some relational operator, Δ, such that "$R = \Delta_{1\leq i\leq n}$ Ri."
- *Rule 3: Disjointness.* If R is decomposed into R1, R2, ..., Rn, and di is a tuple in Rj, then di should not be in any other fragment, such as Rk, where $k \neq j$.

| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P3 | Code | 300,000 | D1 | MPLS |

(a) PROJ1: generated from applying m7

| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P4 | Documentation | 450,000 | D1 | MPLS |

(b) PROJ2: generated from applying m8

| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P1 | Requirements | 135,000 | D2 | NY |

(b) PROJ3: generated from applying m9

| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P2 | Design | 310,000 | D3 | NY |

(c) PROJ4: generated from applying m11

| Pno | Pname | Funds | Dno | Loc |
|---|---|---|---|---|
| P5 | Testing | 250,000 | D4 | LA |

(d) PROJ5: generated from applying m13

**Figure 2.29** The PROJ table fragmentation based on minterm predicates.

Rule 1 states that during fragmentation none of the data in the original table is lost. Every data item that exists in the original table is in at least one of the fragments. We use the term “at least” because vertical fragmentation always requires inclusion of the primary key column in all vertical fragments.

Rule 2 is required for reconstructiveness. This rule is required because, after fragmentation, the original table is not stored in the system anymore. Local DBMS servers will store fragments as part of their local conceptual schema. Globally, users of a distributed database system are not aware of the fragmentation that has been applied. To them, the original table exists as a whole. They will query the original table and not the fragments. Therefore, the system must be able to reconstruct the original table from its fragments on-the-fly.

Rule 3 applies to horizontal fragments. Each horizontal fragment is generated by the application of a minterm predicate to the original table. Therefore, each horizontal fragment houses only the rows that satisfy the corresponding minterm predicate. Since this is true for all horizontal fragments, rows are not shared across horizontal fragments.

### 2.2.8 Replication

During the database design process, the designer may decide to copy some of the fragments or tables to provide better accessibility and reliability. It should be obvious that the more copies of a table/fragment one creates, the easier it is to query that table/fragment. On the other hand, the more copies that exist, the more complicated (and time consuming) it is to update all the copies. That is why a designer has to know the frequency by which a table/fragment is queried versus the frequency by which it is modified—via inserts, updates, or deletes. As a rule of thumb, if it is queried more frequently than it is modified, then replication is advisable. Once we store more than one copy of a table/fragment in the distributed database system, we increase the probability of having a copy locally available to query.

Having more than one copy of a fragment in the system increases the resiliency of the system as well. That is because the probability of all copies failing at the same time is very low. In other words, we can still access one of the copies even if some of the copies have failed: that is, of course, if all copies of a fragment show the same values. Therefore, this benefit comes with the additional cost of keeping all copies identical. This cost, which could potentially be high, consists of total storage cost, cost of local processing, and communication cost. Note that the copies need to be identical only when the copies are online (in service). We will discuss the details of how copies are kept in sync as part of the replication control (in Chapter 7). We will also discuss calculating total cost of queries/updates as part of managing transactions (in Chapter 3) and query optimization (in Chapter 4).

## 2.3 DISTRIBUTION TRANSPARENCY

Although a DDBMS designer may fragment and replicate the fragments or the tables of a system, the users of such a system should not be aware of these details. This is what is known as **distribution transparency**. Distribution transparency is one of the sought after features of a distributed DBE. It is this transparency that makes the system easy to use by hiding the details of distribution from the users. There are three aspects of distribution transparency—location, fragmentation, and replication transparencies.

### 2.3.1 Location Transparency

The fact that a table (or a fragment of table) is stored at a remote site in a distributed system should be hidden from the user. When a table or fragment is stored remotely, the user should not need to know which site it is located at, or even be aware that it is not located locally. This provides for location transparency, which enables the user to query any table (or any fragment) as if it were stored locally.

### 2.3.2 Fragmentation Transparency

The fact that a table is fragmented should be hidden from the user. This provides for fragmentation transparency, which enables the user to query any table as if it were intact and physically stored. This is somewhat analogous to the way that users of a SQL view are often unaware that they are not using an actual table (many views are actually defined as several union and join operations working across several different tables).

### 2.3.3 Replication Transparency

The fact that there might be more than one copy of a table stored in the system should be hidden from the user. This provides for replication transparency, which enables the user to query any table as if there were only one copy of it.

### 2.3.4 Location, Fragmentation, and Replication Transparencies

The fact that a DDBE designer may fragment a table, make copies of the fragments, and store these copies at remote sites should be hidden from the user. This provides for complete distribution transparency, which enables the user to query the table as if it were physically stored at the local site without being fragmented or replicated.

## 2.4 IMPACT OF DISTRIBUTION ON USER QUERIES

Developers of a distributed DBMS try to provide for location, fragmentation, and replication transparencies to their users. This is an attempt to make the system easier to use. It is obvious that in order to provide for these transparencies, a DDBMS must store distribution information in its global data dictionary and use this information in processing the users' requests. It is expensive to give users complete distribution transparency. In such a system, although the users query the tables as if they were stored locally, in reality their queries must be processed by one or more database servers across the network. Coordinating the work of these servers is time consuming and hard to do. In Chapter 1, we discussed the issues related to distributed query execution. We will also address the performance impact of providing complete distribution transparency in Chapter 4. In the rest of this section, we will outline the impact of distribution on a user's queries.

**Example 2.9** Let's assume that our database contains an employee table as defined below:

```
EMP (Eno, Ename, Sal, Tax, Mgr, Dno)
```

The column "Eno" is the primary key of this table. The column "Dno" is a foreign key that tracks the department number of the department in which an employee works. Suppose we have horizontally fragmented EMP into EMP1 and EMP2, where EMP1 contains only employees who work in a department with a department number less than or equal to 10, while EMP2 contains only employees who work in the departments with a department number greater than 10. Assume EMP2 has been replicated and there are two copies of it. Also assume that the company owns three database servers—one server is in Minneapolis (Site 1), one server is in St. Paul (Site 2), and the third server is located in St. Cloud (Site 3). The company decides to store EMP1 at Site 1, one copy of EMP2 at Site 2, and the other copy of EMP2 at Site 3. To see the impact of this database design on a user's queries, let's discuss a very simple query that finds salary information for "Jones," who is an employee with employee number 100. If the system were a centralized DBMS, a user would run the following SQL statement to find Jones' information:

```
Select * from EMP where Eno = 100;
```

In this example, our system is a distributed system and the table has been fragmented and replicated. How would a user write the query in this distributed system? The answer depends on whether or not the system provides for location, fragmentation, and replication transparencies. As mentioned in Chapter 1, the global data dictionary (GDD) stores distribution information for the environment. The impact of data distribution on user queries depends on how much of the distribution information is stored in the GDD and how many types of transparency are provided to the user.

Let's consider three different scenarios, with different degrees of transparency being provided to the user. For the first case, think of a GDD implementation that does not store any distribution information—the GDD is basically nonexistent. In this case, the design, implementation, and administration of the GDD would be trivial, but the GDD would provide absolutely no transparency to the user. This is clearly the simplest GDD implementation possible, but from the user's point of view, it is the hardest to use. The harder the system is to use, the less likely it will be used in the real world! Suppose we had the other extreme situation instead. This other case would be a very powerful GDD providing location, replication, and fragmentation transparencies to the user. While this would be nice for the user and it is the easiest for the user to use, this other extreme would obviously require a more complicated design and implementation, and would also be less trivial to administer. The more complicated a system is to implement and administer, the less likely it is to become ubiquitous. Somewhere between these two extremes, we are likely to find the right balance of transparency provided versus complexity required. In Sections 2.4.1, 2.4.2, and 2.4.3, we will attempt to illustrate the types of trade-offs that need to be considered as we move from case to case between these extremes.

### 2.4.1 No GDD—No Transparency

If the GDD does not contain any information about data distribution, then the users will need to be aware of where data is located, how it is fragmented and replicated, and where these fragments and replicas are stored. This means that the users need to know what information is stored in each of the local systems and then they need to incorporate this information into their queries. Figure 2.30 depicts the contents of the local data dictionaries at each of the three sites and the GDD.

Since the GDD does not store any distribution information, the users need to know this information. Figure 2.31 shows the program that a user writes to retrieve salary for Jones. **Note:** We have used the notation "EMP1@Site1" to indicate the need to run the SQL command against the EMP1 fragment at Site 1. Obviously, this is not a valid SQL statement and will **not** be parsed correctly by any commercial DBMS. We are assuming, however, that our DDBMS has a parser that understands this notation, translates it into the correct syntax, and actually sends the correct SQL statements to the right database servers. The notation /*. . . */ represents a comment in SQL.

In this program, the user assumes Jones is in EMP1 and queries EMP1 looking for Jones. The user had to "hard-code" the location for the table indicating which site contained the EMP1 table. Since there is only one copy of EMP1 in the system, the user specified Site 1. If the employee is not found there, then the employee might be in EMP2. We have two copies of EMP2 (at Site 2 and Site 3). There is no need to look in both of them since these are copies of the same fragment. The user has to decide which site to query for this fragment. In our example, the user has chosen Site 3. For this simple query, the user needed to write a rather long program,

```
GDD:Contains no distribution information

Site 1 Schema:
EMP1 (         Eno                          Integer,
               Ename                        Char (20),
               Sal                          Number(10,2),
               Tax                          Number(5,2),
               Mgr                          Integer,
               Dno                          Integer
                            );
Site 2 Schema:
EMP2 (         Eno                          Integer,
               Ename                        Char (20),
               Sal                          Number(10,2),
               Tax                          Number(5,2),
               Mgr                          Integer,
               Dno                          Integer
                            );
Site 3 Schema:
EMP2 (         Eno                          Integer,
               Ename                        Char (20),
               Sal                          Number(10,2),
               Tax                          Number(5,2),
               Mgr                          Integer,
               Dno                          Integer
                            );
```

**Figure 2.30** The GDD and local schemas for Example 2.9.

```
If (select count (Sal) from EMP1@Site1 where Eno = 100) = 1
Then /* Jones is in fragment EMP1*/
              Select Sal
              From EMP1@Site1
              Where Eno =100
Else /* Jones is not in EMP1 - check EMP2 at Site 3 */
              If (select count (Sal)
                  from EMP2@Site3 where Eno = 100) = 1
              Then /* Jones is in fragment EMP2 */
                            Select Sal
                            From EMP2@Site3
                            Where Eno = 100
              Else /* Jones is not in the database */
                            Output "No such employee"
              End if;
End if;
```

**Figure 2.31** DDBMS with no transparency.

incorporating fragmentation, replication, and location information directly in the query. That is because distribution information is not stored in the GDD. In other words, if there is no GDD, then there is no distribution transparency provided to the user. If any of these details were to change (the fragmentation design, the number of fragments, the number of copies, or the locations for any of the tables/fragments/copies) then this program might stop working! This program would need to be modified, manually, to reflect the new information before it would work correctly again.

### 2.4.2 GDD Containing Location Information—Location Transparency

Now let's assume that the GDD contains location details for the tables/fragments in the environment, but no details about how the fragments relate to each other or to the EMP table as a whole. For example, the GDD could store location information in a special table called "Location_Table" as depicted in Figure 2.32. Assume we have the same local schemas for this case as we did for the pervious case (see Figure 2.30).

Since the GDD has location information for all the fragments, the system does provide for location transparency. As a result, the user does not need to specify location information in the SQL statements. Therefore, the user queries will be a bit simpler than they were in the previous case. Figure 2.33 shows the query we would write for

| Table_Name | Site_Name |
|---|---|
| EMP1 | Site1 |
| EMP2 | Site2 |
| EMP2 | Site3 |

**Figure 2.32** The Location_Table of the GDD.

```
If (select count (Sal) from EMP1 where Eno = 100) = 1
Then /* Jones is in fragment EMP1*/
            Select Sal
            From EMP1
            Where Eno =100
Else /* Jones is not in EMP1 - check EMP2 */
            If (select count (Sal)
                from EMP2 where Eno = 100) = 1
            Then /* Jones is in fragment EMP2 */
                        Select Sal
                        From EMP2
                        Where Eno = 100
            Else /* Jones is not in the database */
                        Output "No such employee"
            End if;
End if;
```

**Figure 2.33** DDBMS with location transparency.

this environment in order to answer the same question we considered in the previous section.

As you can see from this program, all of the location information for the fragments has been removed from the select statements. In this case, the DDBMS looks up the location of EMP1 and EMP2 in the Location_Table in the GDD and sends the SQL statement to the desired site automatically. Although these select statements are less complicated to create than the ones given before, the user still needs to know that there are two fragments of the EMP table and that EMP2 has been replicated. The only benefit is that the user does not need to know where these fragments are stored. If the GDD stores additional details about each fragment (such as the fragmentation type for each table, how each fragment is generated, how it fits into the overall fragmentation plan, and the replication information for each fragment), then the user does not need to specify any of these details in the program.

### 2.4.3 Fragmentation, Location, and Replication Transparencies

To provide for location, fragmentation, and replication transparencies, the GDD has to store fragmentation, location, and replication information. This GDD implementation is the most complicated and difficult. Conversely, this environment is the easiest for the user to use. The GDD needs to contain additional tables to store the necessary information. For example, in addition to the Location_Table shown in Figure 2.32, we can create a table named "Fragmentation_Table." This new table contains information about real tables, real fragments, and "virtual tables." A virtual table refers to a table that is reconstructed from a set of fragments—recall that this table does not physically exist in the DDB! A simple GDD for this scenario is depicted in Figure 2.34. The local schemas for this example are the same ones we used in the previous sections (see Figure 2.30). In this case, the DDBMS provides the user with the illusion that the EMP table exists as a single physical table. As such, the DDBMS needs to store the definition of a virtual EMP table for users to access. Since the system is providing complete distribution transparency, the users query the system as if the system were a centralized system—just like a traditional, nondistributed system. Therefore, getting the salary for Jones can easily be achieved by the following SQL statement.

```
Select Sal
From EMP
Where Eno = 100;
```

Since the EMP table is a virtual table, when processing this query, the system generates all the proper subqueries (defined against the physical fragments) and then runs them at the sites where the fragments are stored.

## 2.5 A MORE COMPLEX EXAMPLE

Let's consider a more complicated example, in which we will apply a series of horizontal and vertical (hybrid) fragmentation to the table "EMP(eno, name, sal, tax, mgr, dno)." This fragmentation approach fragments the EMP table into four fragments called "EMP1," "EMP2," "EMP3," and "EMP4" as shown in Figure 2.35.

```
EMP (    Eno          Integer,
         Ename        Char (20),
         Sal          Number(10,2),
         Tax          Number(5,2),
         Mgr          Integer,
         Dno          Integer
                      );
```

EMP Virtual Table

| Table_Name | Site_Name |
|---|---|
| EMP1 | Site1 |
| EMP2 | Site2 |
| EMP2 | Site3 |

Location_Table

| Table_Name | Fragment_Type | Virtual/Physical | Condition |
|---|---|---|---|
| EMP | | Virtual | |
| EMP1 | Horizontal | Physical | Dno <= 10 |
| EMP2 | Horizontal | Physical | Dno > 10 |

Fragmentation_Table

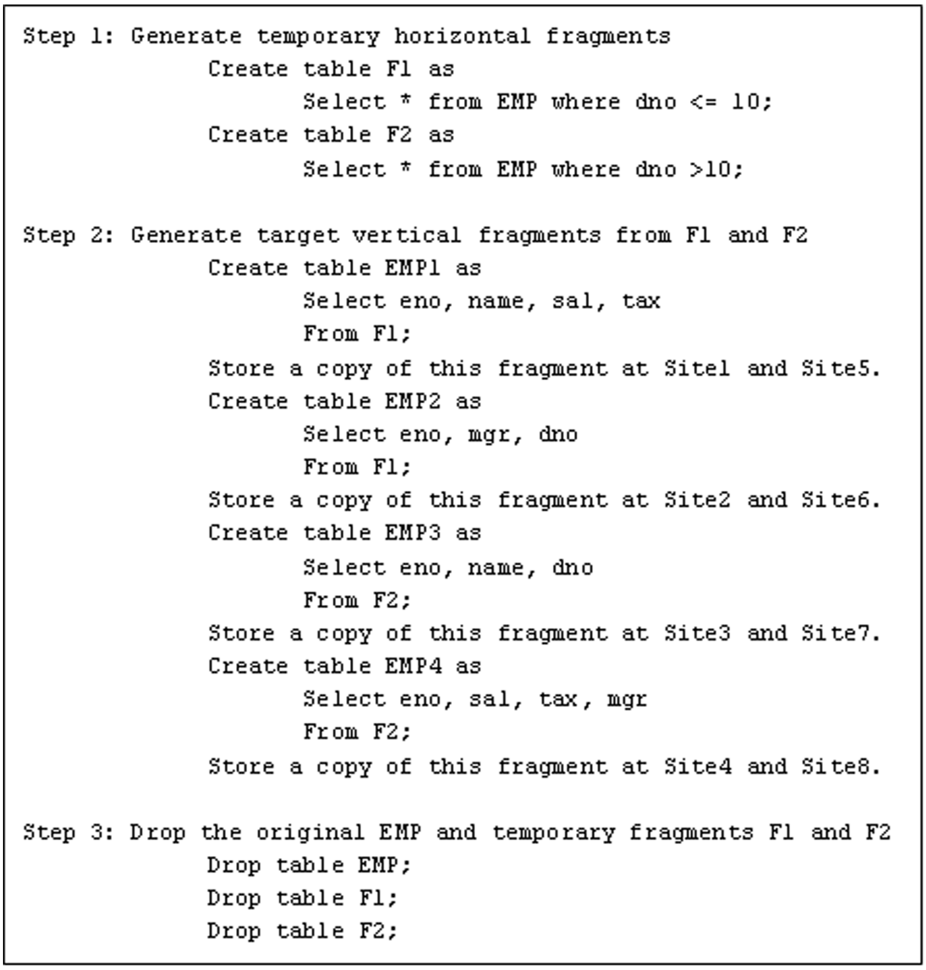
**Figure 2.34** GDD for Example 2.9.

Obviously EMP1 and EMP2 both contain information about those employees who work in the departments numbered less than or equal to 10. Similarly, EMP3 and EMP4 both contain information about those employees who work in the departments numbered greater than 10. Figure 2.36 depicts the distribution of the EMP table and its fragments.

To show the effect of this fragmentation on our queries and commands, let's assume that "Smith" is an employee whose employee number is 100 (eno = 100). Smith currently works in department number 3 (dno = 3). Therefore, Smith's employee information is stored in EMP1 and EMP2. Suppose we need to move Smith to department number 15 (dno = 15). How do we achieve this? The answer, as explained in the previous case, depends on what is stored in the GDD. To illustrate the differences between the transparency alternatives, we will consider how the same operation is performed against the same schema, in three different scenarios. Each scenario will have a GDD that provides a different level of transparency.

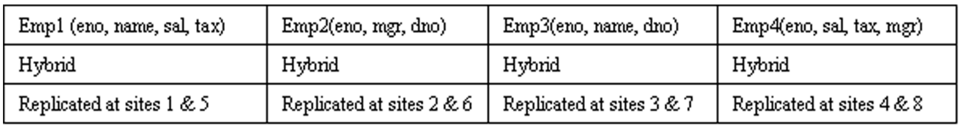
Again, we will consider the following three levels of transparency:

- The system provides for fragmentation, location, and replication transparency.



```
Step 1: Generate temporary horizontal fragments
            Create table F1 as
                  Select * from EMP where dno <= 10;
            Create table F2 as
                  Select * from EMP where dno >10;

Step 2: Generate target vertical fragments from F1 and F2
            Create table EMP1 as
                  Select eno, name, sal, tax
                  From F1;
            Store a copy of this fragment at Site1 and Site5.
            Create table EMP2 as
                  Select eno, mgr, dno
                  From F1;
            Store a copy of this fragment at Site2 and Site6.
            Create table EMP3 as
                  Select eno, name, dno
                  From F2;
            Store a copy of this fragment at Site3 and Site7.
            Create table EMP4 as
                  Select eno, sal, tax, mgr
                  From F2;
            Store a copy of this fragment at Site4 and Site8.

Step 3: Drop the original EMP and temporary fragments F1 and F2
            Drop table EMP;
            Drop table F1;
            Drop table F2;
```

**Figure 2.35** The EMP table fragmentation steps.



| Emp1 (eno, name, sal, tax) | Emp2(eno, mgr, dno) | Emp3(eno, name, dno) | Emp4(eno, sal, tax, mgr) |
|---|---|---|---|
| Hybrid | Hybrid | Hybrid | Hybrid |
| Replicated at sites 1 & 5 | Replicated at sites 2 & 6 | Replicated at sites 3 & 7 | Replicated at sites 4 & 8 |

**Figure 2.36** The EMP table distribution.

- The system provides for location and replication transparencies, but no fragmentation transparency.
- The system does not provide for any transparency.

### 2.5.1 Location, Fragmentation, and Replication Transparencies

Because the system provides for location, fragmentation, and replication transparencies, we don't need to worry about any of the distribution details. As a result, our query can be written as if it were running in a centralized system. Because Smith needs to be moved from department 3 to department 15, we have to make a simple update to the database and indicate the new department number. We can do this move by using a single, simple SQL statement as shown:

```
Update Emp
Set dno = 15
Where eno = 100;
```

We do not need to add any complex logic to ensure the integrity of the database. The DDBMS knows that EMP is a virtual table. It looks up the fragmentation, location, and replication details in the GDD. The DDBMS uses this information to translate our update statement into the necessary SQL statements that need to run on the individual local systems. The DDBMS then executes the appropriate statements at the appropriate sites to make the requested updates to the necessary tables/fragments/copies.

### 2.5.2 Location and Replication Transparencies

In this case, the system provides for location and replication transparencies, but it does not provide for fragmentation transparency. Like the previous scenario, we do not need to include any location or replication details in our SQL. However, we do need to know the fragmentation details and we need to embed these details in our SQL. Unlike the previous case, our program is hard-coded to work for the old value of dno. In other words, because the old value of dno was less than or equal to 10, we cannot achieve this move in a single statement. First, we need to fetch the column values for EMP1 and EMP2 and store them into some program variables. Next, we need to delete the old values from the EMP1 and EMP2 fragments. Finally, we insert the values of our variables into the EMP3 and EMP4 fragments.

```
select name, sal, tax into $name, $sal, $tax
from Emp1
where eno = 100;

select mgr into $mgr
from Emp2
where eno = 100;

delete Emp1 where eno = 100;
delete Emp2 where eno = 100;

insert into Emp3 Values (100, $name, 15);
insert into Emp4 Values(100, $sal, $tax, $mgr);
```

In this case, we need to make sure that Smith's information is collected from the old fragments first to ensure that we do not lose them before we insert them in the new fragments. This means that we need to use some program variables to hold the selected information. Program variables $name, $sal, $tax, and $mgr are used to collect name, sal, tax, and mgr information from EMP1 and EMP2. After this information is collected, we delete Smith's information from EMP1 and EMP2 and insert Smith's information into EMP3 and EMP4. It should be obvious that the user would need to explicitly add transactions and/or locks to this program to ensure the integrity of the system during this move. We will not discuss locking and commitment/rollback of the transactions here (see Chapter 6 for details).

**Note 1:** This program only works when the old department number for Smith is less than or equal to 10 and the new department number is greater than 10. It should be obvious that moving Smith from department, say, 15, to department 3 would require a different program with the deletion of Smith's information from EMP3 and EMP4 and insertion of Smith's information back into EMP1 and EMP2. If, on the other hand, Smith is currently in department 12, then the move to department 15 only requires updating Dno in EMP3 and EMP4. Remember, any programs in this system must embed (hard-code) the fragmentation details defined in Figure 2.36 into the SQL statements. If any of the fragmentation details change, our program would be broken until we manually fixed it by hard-coding the new details.

**Note 2:** We have eliminated error checking in an attempt to not clutter the code. We should **always** implement error checking and validation checks in the real code. For example, in this case we should check whether or not Smith is an employee in the database, check the current department number, and other details before we perform the actual update. We will leave implementing the code of a general case as an exercise.

### 2.5.3 No Transparencies

This is the most complex case from the query-writer's perspective. In this case, the distributed database system does not provide for any of the fragmentation, location, or distribution transparencies. Since these transparencies are not provided by the system, we need to be aware of the location, fragmentation, and replication information. In Section 2.5.2, we analyzed the case of location and replication transparencies. Now that we do not have these transparencies, we have to manually hard-code in our program all the details that those transparences would normally provide. This makes the program very hard to write. The code snippet below shows the SQL-like program for this case. (Note that in the code, we have used the notation "@site X" to indicate that the SQL statement needs to run at Site X.)

```
select name, sal, tax into $name, $sal, $tax
from Emp1
where eno = 100 @site1;

select mgr into $mgr
from Emp2
where eno = 100 @site2;

delete Emp1 where eno = 100 @site 1;
delete Emp1 where eno = 100 @site 5;
delete Emp2 where eno = 100 @site 2;
delete Emp2 where eno = 100 @site 6;

insert into Emp3 values (100, $name, 15) @site 3;
insert into Emp3 values(100, $name, 15) @site 7;
insert into Emp4 values(100, $sal, $tax, $mgr) @site 4;
insert into Emp4 values (100, $sal, $tax, $mgr) @site 8;
```

This program resembles the program we presented in Section 2.5.2. However, since there are two copies of each fragment, we need to explicitly insert Smith's information into each copy of the new fragment and also explicitly delete Smith's information from each copy of the old fragment. Once again, we should really explicitly add transactions or locks to this program to ensure the integrity of the system. Notes 1 and 2 from Section 2.5.2 apply here as well.

## 2.6 SUMMARY

Distribution design in a distributed database management system is a top–down process. Designers start with the data access requirements of all users of the distributed system and design the contents of the distributed database as well as the global data dictionary to satisfy these requirements. Once the GDD contents are determined, the designers decide how to fragment, replicate, and allocate data to individual database servers across the network. In this chapter, we outlined the necessary steps for such a design.

We outlined the steps for creating horizontal fragments and vertical fragments of a table based on a set of requirements. We examined the rules for correct fragmentation design (such as completeness, reconstructiveness, and disjointness). We briefly discussed the impact of distribution on complexity of user programs to access and update distributed information.

## 2.7 GLOSSARY

**Access Frequency** A measurement reflecting how often an application accesses the columns of a table during a defined period of time, for example, "fifteen accesses per day."

**Affinity** A measure of closeness between the columns of a table as it relates to a given application or a set of applications.

**Affinity Matrix** A matrix that contains the affinity of all columns of a table for all applications that refer to them.

**Bond Energy Algorithm (BEA)** The algorithm that calculates a metric that indicates the benefits of having two or more columns to be put in the same vertical partition.

**Bottom–Up** A methodology that creates a complex system by integrating system components.

**Clustered Affinity Matrix** A matrix that is used for vertical partitioning of columns of a table based on the Bond Energy Algorithm.

**Derived Horizontal Fragmentation** A horizontal fragmentation approach that fragments rows of a table based on the values of columns of another table.

**Distribution Transparency** A type of transparency that hides the distribution details of database tables or fragments from the end users.

**Fragmentation Transparency** A type of transparency that hides fragmentation details of database tables or fragments from the end users.

**Fragment** A partition of a table.

**Global Conceptual Model** A repository of information such as location, fragmentation, replication, and distribution for a distributed database system.

**Global Data Dictionary (GDD)** The portion of global conceptual schema that contains dictionary information such as table name, column names, view names, and so on.

**Horizontal Fragmentation** The act of partitioning (grouping) rows of a table into a smaller set of tables.

**Hybrid Fragmentation** The application of horizontal fragmentation to vertical fragments of a table, or the application of vertical fragmentation to horizontal fragments of a table.

**Local Conceptual Model** The conceptual schema that users of a local DBMS employ.

**Location Transparency** A transparency that hides location of database tables or fragments from the end users.

**Minterm Predicate** A predicate that contains two or more simple predicates.

**Partitions** Either vertical or horizontal fragments of a table.

**Primary Horizontal Fragmentation** A horizontal fragmentation of a table based on the value of one of its columns.

**Replication Transparency** A transparency that hides the fact that there might be more than one copy of a database table or fragments from the end users.

**Schema Integration** The act of integrating local conceptual schemas of the component database systems into a cohesive global conceptual schema.

**Simple Predicate** A condition that compares a column of a table against a given value.

**Top–Down** A database design methodology that starts with the requirements of a distributed system (the global conceptual schema) and creates its local conceptual schemas.

**Unified Schema** An integrated, nonredundant, and consistent schema for a set of local database systems.

**Usage Matrix** A matrix that indicates the frequency of usage of columns of a table by a set of applications.

**Vertical Fragmentation** The act of partitioning the columns of a table into a set of smaller tables.

## REFERENCES

[Bobak96] Bobak, A., *Distributed and Multi-Database Systems*, Artech House, Boston, MA, 1996.

[Ceri84] Ceri, S., and Pelagatti, G., *Distributed Databases—Principles and Systems*, McGraw-Hill, New York, 1984.

[Ceri87] Ceri, S., Pernici, B., and Wiederhold, G., "Distributed Database Design Methodologies," *Proceedings of IEEE*, Vol. 75, No. 5, pp. 533–546, May 1987.

[Eswaren74] Eswaren, K., "Placement of Records in a File and File Allocation in a Computer Network," in *Proceedings of Information Processing Conference*, Stockholm, pp. 304–307, 1974.

[Hammer79] Hammer, M., and Niamir, B., "A Heuristic Approach to Attribute Partitioning," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 93–101, Boston, May 1979.

[Hoffer75] Hoffer, H., and Severance, D., "The Use of Cluster Analysis in Physical Database Design," in *Proceedings of the First International Conference on Very Large Databases*, Framingham, MA, pp. 69–86, September 1975.

[McCormick72] McCormick, W., Schweitzer, P., and White T., "Problem Decomposition and Data Reorganization by a Clustering Technique," *Operations Research*, Vol. 20, No. 5, pp. 993–1009, 1972.

[Navathe84] Navathe, S., Ceri, S., Wiederhold, G., and Dou, J., "Vertical Partitioning Algorithms for Database Design," *ACM Transactions on Database Systems*, Vol. 9, No. 4, pp. 680–710, December 1984.

[Özsu99] Özsu, M., and Valduriez, P., *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ, 1999.

[Sacca85] Sacca, D., and Wiederhold, G., "Database Partitioning in a Cluster of Processors," *ACM Transactions on Database Systems*, Vol. 10, No. 1, pp. 29–56, October 1985.

## EXERCISES

Provide short (but complete) answers to the following questions.

**2.1** Assume EMP has been fragmented as indicated in Figure 2.36. Also, assume the system **does not** provide for any transparencies. Write a SQL-like program that deletes the employee indicated by $eno from the database. Make sure you perform all the necessary error checking.

**2.2** Answer true or false for the statements in Figure 2.37.

**2.3** For the EMP table fragmented in Example 2.5, write a **single** SQL statement that reconstructs the original EMP table from its fragments.

**2.4** An Employee table has the following relational scheme: "Employee (name, sal, loc, mgr)," where name is the primary key. The table has been horizontally fragmented into SP and MPLS fragments. SP has employees who work in St. Paul and MPLS contains all employees who work in Minneapolis. Each fragment is stored in the city where the employees are located. Assume transactions only enter the system in NY and there are no employees in NY. Write down the local schemas and global schema and indicate in which cities these schemas are located for the following three cases:

**(A)** The system does not provide for any transparencies.

**(B)** The system provides for location and replication transparencies.

**(C)** The system provides for location, replication, and fragmentation transparencies.

**2.5** Consider table "EMP(EmpID, Name, Sal, Loc, Dept)." There are four applications running against this table as shown below. Design an optimal vertical

| # | Statement | True/ False |
|---|---|---|
| 1 | In SQL, vertical fragments are created using the project statement. | |
| 2 | The external schemas define the database as the end users see it. | |
| 3 | Tuple is another word for a row in a relational database. | |
| 4 | Federated databases have three levels of schemas. | |
| 5 | Federated databases are formed from the Bottom–Up. | |
| 6 | Referential integrity enforced by a DDBMS does not span sites. | |
| 7 | Horizontal fragments need to be disjoint. | |
| 8 | Distributed DBMSs are formed Top–Down. | |
| 9 | Global data dictionary may be copied at some sites in a distributed DBMS. | |
| 10 | There are as many physical fragments as there are minterm predicates of a minimal and complete distribution design. | |

**Figure 2.37** True or false questions.

fragmentation strategy that satisfies the needs of these applications. Show steps for arriving at the answer.

A1: "Select EmpID, Sal From EMP;"
A2: "Select EmpID, Name, Loc, Dept From EMP Where Dept = 'Eng';"
A3: "Select EmpID, Name, Loc, Dept From EMP Where Loc = 'STP';"
A4: "Select EmpID, Name, Loc, Dept from EMP Where Loc = 'MPLS';"