2021

(2021 Revision) Chapter 3: Essential Aspects of Physical Design and Implementation of Relational Databases

Tatiana Malyuta
CUNY New York City College of Technology

Ashwin Satyanarayana
CUNY New York City College of Technology

Chapter 3. Distributed Database Design

Today's amazing growth of databases is caused by the needs of business and is possible because of continuous technological advances. Under these circumstances, good performance and scalability of growing databases become very important, as well as reliability and constant availability of data. Distributed database design can significantly improve these database features.

> **The distributed database is a collection of physically independent and logically related databases. Each database of the distributed database is capable of processing data independently; however, some applications process data from multiple databases.**

We will discuss possible distributed solutions for the Manufacturing Company case described in Appendix 1 with use cases. The relational model of the database includes three relations: Department, Title, and Employee. The diagram in Figure 1 shows the dependencies between the relations: Employee references Department and Title, or in other words, Employee is the child relation, and Department and Title are the parent relations.

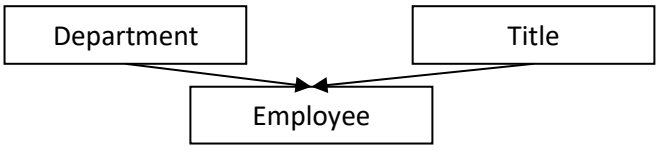

Figure 1. Diagram of dependencies between relations.

Making decisions about data distribution requires an understanding of how the different database applications work with the data. For the Manufacturing Company case, departments are located in three cities: Boston, New York, and Cleveland. Each of the company's employees is assigned to one department. An application in each city is responsible for the support and maintenance of data about the local departments and the employees assigned to these departments. The New York office, in addition to supporting local data, is in charge of some reports for all departments and employees of the company.

Each office has a database server; the servers are connected via the network as shown in Figure 2.

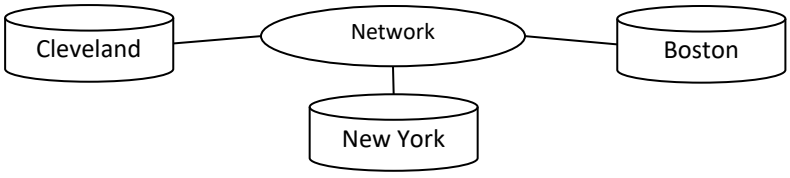

Figure 2. General architecture of the distributed database.

For our future discussions and analyses, we will use the following sample data of the relations:

Title

titleCode	titleDescription	salary
T1	Accountant	10000
T2	Analyst	20000
T3	Programmer	30000
T4	DBA	40000
T5	Manager	50000

Department

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
002	Budget	New York	Business
003	Marketing	Boston	Marketing
004	Database Support	Cleveland	IT
005	Purchasing	New York	Business

Employee

ID	empName	emplType	deptCode	titleCode
1	John	Full-time	002	T1
2	Adam	Consultant	001	T3
3	Mary	Part-time	004	T4
4	Peter	Full-time	003	T2
5	Scott	Consultant	002	T1
6	Susan	Full-time	005	T5
7	Alex	Part-time	004	T2

The distributed solution, which is more expensive and more complex than the centralized one, is justified by the following expectations:

- Improving performance of local applications in each city and the global New York application
- Achieving better reliability and availability of the data
- Ensuring that the database is scalable and will be able to function with an increased number of users and larger amounts of data.

Prerequisites of the Distributed Design

Distribution of data is one of the considerations in performing the physical design of the database. Decisions related to physical design are based on the logical data model and user requirements that specify:

- The structure of the company and the architecture of the company's network
- How data will be used by database applications, including what data is needed by each application, the amounts of processed data, and the frequencies of applications' execution.

This chapter discusses how to use these requirements for designing appropriate distributions of data.

When performing the design and implementation of the distributed database, it is important to understand that the physical data model has to correspond to the logical model and preserve all semantics of the latter.

Review of Basic Features of the Distributed Database

The basic features of distributed databases which were discussed in Chapter 2 on the physical model of data are as follows:

- The distributed database is a collection of logically interrelated shared data stored across several physically independent databases.
- Participating databases are linked by a network.
- Each participating database is controlled by a DBMS autonomously.

- Relations of the logical data model can be split into fragments.
- Fragments and unfragmented relations are implemented as tables in participating databases.
- Fragments and relations can be replicated in several participating databases.
- Each participating database is involved in at least one global application, i.e. application processing data from several participating databases.
- The distributed database has to appear like a centralized database to the users.

The distribution and replication of data have to provide better database performance, improve reliability and availability of data, and make the database more scalable. These are achieved by localizing the data through fragmentation and replication, and providing users with easy access to needed data.

General Approaches to Distribution

Types of Fragmentation

Implementing entire relations in different databases of a distributed database is not a typical distribution scenario. It is important to remember that the main goals of distributed databases are to improve performance, availability, and reliability through the localization of data. If, for example, the table Department is stored in the Boston database and the table Employee is stored in the New York database, then this distribution does not localize data for the applications of the Manufacturing Company case. First, each of the local applications needs to access remote databases to get necessary data about local departments and employees, and second, to get this local data, the local application has to process data about departments and employees from other locations as well. This design neither improves reliability nor availability, as if one of the databases goes down, most of the users will have to wait to do their processing.

Usually, distributed design involves fragmenting relations of the logical model and allocating the fragments in the database at the locations where users most often need data from the fragments.

Relations can be fragmented *horizontally*, where *rows* of a relation are stored in different databases, and *vertically*, where *columns* of a relation are stored in different databases. Horizontal and vertical fragmentations of the relation Department are shown schematically in Figure 3 and Figure 4, respectively.

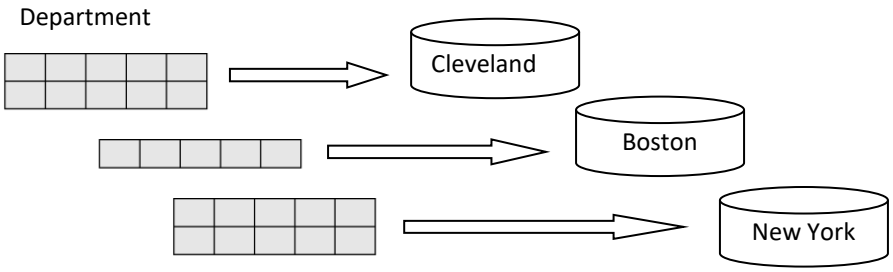

Figure 3. Horizontal fragmentation.

Horizontal and vertical fragmentations can also be combined in a hybrid fragmentation.

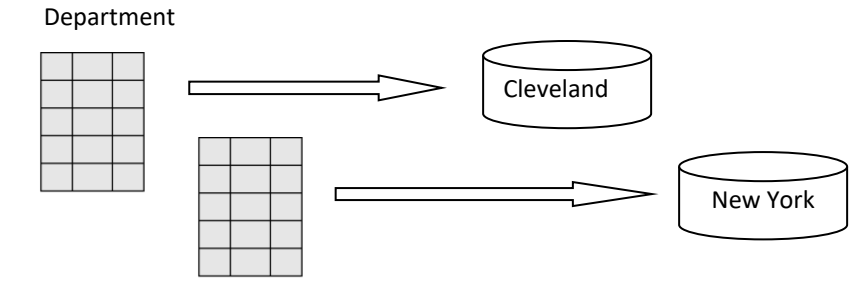

Figure 4. Vertical fragmentation.

The results of data processing should not depend on whether the database is implemented as centralized or distributed. Fragmentation must abide by the following simple rules:

1. *Completeness*. Data should not be lost in the process of fragmentation. Each data item must be included in at least one fragment.
2. *Disjointness*. Data should not be duplicated in the process of fragmentation to avoid inconsistency and incorrect results of queries. Each data item must be included in not more than one fragment.

These two rules of correctness of fragmentation mean that:

Each data item must be included in one and only one fragment.

In addition to these formal rules we want fragmentation to result in an even distribution of data across fragments, e.g., we do not want one of the fragments to contain 90% of records of the table and another one to contain 10% as such fragmentation will undermine the goals of distribution.

If necessary, all data from a correctly fragmented relation have to be made available, and to accomplish this there is an operation for each type of fragmentation that allows restoring all the data from the fragments. The discussion of fragmentation and restoring data from fragments is provided with the help of the relational operations that are described in Appendix 2.

Horizontal fragmentation

Information Requirements

Case 1.

In the Manufacturing Company case, users of each city work with data about local departments (the departments located in the city).

This situation makes us consider horizontal fragmentation of the relation Department in three fragments: the first will contain all the rows of the relation with location 'Boston', the second – the rows with location 'New York', and the third – the rows with location 'Cleveland'. Data is localized because users in each city have the data they need in their local databases (i.e., the databases they directly work with), and each local application deals with needed data only and is not burdened with support of other data.

Case 2.

> *In addition to the requirements for Case 1, there is another application in Boston that processes data about all IT departments, and there is a similar application in Cleveland which works with data about the Business and Marketing departments.*

The previous distributed solution does not provide data localization for these applications – to get the needed data about the departments of a particular type, users have to access remote databases (i.e., the databases to which the users are not connected directly). Most probably, the performance of the applications of this case, as well as the availability of data, will be worse than in the centralized database. The fragmentation suggested for the Case 1 does not offer data localization for the new applications. However, if we decide to reconsider the fragmentation and localize the data for the new applications from Boston and Cleveland by allocating all rows of the IT departments in the Boston database and the rows of all other departments in the Cleveland database, the distribution is not localized for the applications dealing with data about local departments.

Here we encounter one of the mentioned problems of distributed design (and physical design in general) – often we have to make our decisions for conflicting user requirements. To resolve this conflict, we have to find out which group of users' needs are more important. Usually the "importance" of an application is judged by how often it accesses data and how crucial its performance is. If, for example, the users of the applications mentioned in Case 1 access data about local departments several thousand times per day for each city, while the users of the additional applications mentioned in Case 2 access data for a particular department type several times a month, then performance of the first group of applications seems more important and we will try to localize the data for this group of applications.

There exists an empirical 80/20 rule: 80% percent of data processing in a company is performed by 20% of the applications. The needs of the applications that fall into these 20% should be considered first.

These two simple cases show that horizontal fragmentation is beneficial only if it localizes data for the most frequently used applications.

> **Horizontal fragmentation of a relation is performed based on information about what rows of the relation are accessed, how frequently and by what groups of users.**

Primary Horizontal Fragmentation

When fragmentation is performed by applying user requirements about what rows of the relation are accessed, from which sites and how frequently, it is called *primary* horizontal fragmentation.

A general rule for correctness of horizontal fragmentation is that each row of the relation must be included in one and only one fragment.

> **Each row of a relation must be included in one and only one horizontal fragment.**

After the most important applications are defined, the designer must concentrate on the relations involved in these applications and analyze the conditions by which rows of these relations are accessed from different sites. These conditions define the local needs in data. For requests in SQL, these are the WHERE clauses of queries executed from different applications.

Let us consider the fragmentation of the relation Department for Case 1. Applications access data about local

departments with the help of the following query, where '?' stands for one of the three locations of the company.

```
SELECT * FROM Department WHERE location = ?,
```

The conditions (also called predicates), which define rows used by each site, are:

P_1 : location = 'Boston'

P_2 : location = 'New York'

P_3 : location = 'Cleveland'

Horizontal fragmentation is defined with the help of the relational operation *Selection* (see Appendix 2 for relational operations). For the relation $R(A_1, A_2, \ldots, A_m)$ and predicates p_i, i from 1 to N, N fragments of R are defined with the help of the selection operation on R by conditions of p_i:

$$R_i = \sigma_{p_i}(R),\ 1 \le i \le N$$

For our case, the three local needs for data expressed by predicates on the attribute location of the relation Department produce three fragments of the table:

$$Department_1 = \sigma_{location = \text{'Boston'}}(Department)$$

$$Department_2 = \sigma_{location = \text{'New York'}}(Department)$$

$$Department_3 = \sigma_{location = \text{'Cleveland'}}(Department)$$

The fragmentation is correct because it is complete and disjoint.

Boston:

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
003	Marketing	Boston	Production

New York:

deptCode	deptName	location	deptType
002	Budget	New York	Business
005	Purchasing	New York	Business

Cleveland:

deptCode	deptName	location	deptType
004	Database Support	Cleveland	IT

The initial relation is restored from its horizontal fragments with the help of the relational operation *Union*:

$$Department = Department_1 \cup Department_2 \cup Department_3$$

The initial relation is restored correctly if the fragmentation is correct.

Case 3.

The users in Boston and New York work with data about local departments; users in Cleveland work with data about local and New York departments.

The predicates, which define rows of the relation Department used on each site, are:

P_1 : location = 'Boston'

P_2 : location = 'New York'

P_3 : location = 'Cleveland' OR location = 'New York'

The fragmentation of Department by these predicates is the following:

Boston:

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
003	Purchasing	Boston	Production

New York:

deptCode	deptName	location	deptType
002	Budget	New York	Business
005	Purchasing	New York	Business

Cleveland:

deptCode	deptName	location	deptType
002	Budget	New York	Business
004	Database Support	Cleveland	IT
005	Purchasing	New York	Business

This fragmentation is obviously incorrect – it is not disjoint, since the rows for the departments of New York are included in two fragments: in New York and Cleveland. If someone who was unaware of this special fragmentation decides to count departments of the company, the result will not correspond to the actual number of departments. The duplication of data about the New York departments in the Cleveland database makes data inconsistency possible (remember that the databases are supported independently).

Duplication caused by violation of the disjointness rule should not be confused with replication. Replicas are synchronized copies of objects of one database in another database and they are declared as such when created. For a distributed database, we can replicate whole fragments or tables. The support of replicas (keeping the copies of data in sync) is provided by the DBMS.

To produce the correct fragmentation, we need to analyze the frequencies of access of data about the New York departments from the New York and Cleveland sites. If the local New York application needs this data more often than the application in Cleveland, then we can use the fragmentation used for the Case 1. If, on the other hand, Cleveland applications are more important, then we will have the following fragmentation:

P_1 : location = 'Boston'

P_2 : location = 'Cleveland' OR location = 'New York'

Case 4.

> *The Boston users need data about the IT departments; Cleveland users need the data about all other departments. The New York users do not work with data about departments.*

The predicates for this case are:

P_1 : deptType = 'IT'

P_2 : deptType = 'Business' OR deptType = 'Marketing'

Fragmentation by these predicates gives two fragments:

$Department_1 = \sigma_{deptType = 'IT'}$ (Department)

$Department_2 = \sigma_{deptType = 'Business' \text{ OR } deptType = 'Marketing'}$ (Department)

If a new department of the type 'Advertising' is added to the company, the fragmentation becomes incorrect – it will be incomplete because the new department will be not placed in any fragment.

A different set of predicates gives a better fragmentation solution for this case:

P_1 : deptType = 'IT'

P_2 : deptType ≠ 'IT'

The discussed cases show that the correctness of fragmentation is defined by correctness of the set of predicates. When building the set of predicates we cannot simply map the conditions of data access from each site. For the fragmentation to be complete and disjoint, the set of predicates must satisfy the conditions of completeness and disjointness:

- *Disjointness.*: For each two predicates p_i and p_j, where $i \neq j$, for each row of the relation: p_i AND p_j = FALSE.
- *Completeness*. For all N predicates, for each row of the relation: p_1 OR p_2 OR … p_N = TRUE.

> **The correct primary horizontal fragmentation is defined by correct predicates. The set of predicates for fragmentation must be complete and disjoint.**

In all previous cases the predicates were simple and defined on one attribute. Consider a more complicated situation.

Case 5.

> *Each city of the company has two offices: the first office is working with data about local IT departments, and the second office is working with data about all other local departments.*

For this case, local access to data is defined by composite conditions on two attributes:

P_1 : location = 'Boston' AND deptType = 'IT'

P_2 : location = 'Boston' AND deptType ≠ 'IT'

P_3 : location = 'New York' AND deptType = 'IT'

P_4 : location = 'New York' AND deptType ≠ 'IT'

P_5 : location = 'Cleveland' AND deptType = 'IT'

P_6 : location = 'Cleveland' AND deptType ≠ 'IT'

It is easy to check that the fragmentation by these predicates is correct (note that for our example some of the fragments are empty). However, this fragmentation does not make sense because there are more fragments than databases, and we will not be able to allocate all fragments. It is important to remember that fragmentation has to be performed with understanding of the architecture of the company's network and availability of database servers.

For this case, we could use the fragmentation of the Case 1 based on the 'location' attribute or the fragmentation of the Case 4. based on the 'deptType' attribute; the latter fragmentation, however, results in a poorer localization of data. Distribution of data can be combined with local measures for performance improvement. For example, if data usage patterns call for it, for the fragmentation by the 'location' attribute we may consider the partitioning of each fragment by deptType.

The set of predicates must not only be correct, but also *relevant*. In this case it means that there must be at least one application that accesses the resulting fragments differently from the others. If, for example, for the Case 1 we choose the set of predicates:

P_1 : deptType = 'IT'

P_2 : deptType ≠ 'IT'

then applications on each site will always access both fragments. This fragmentation does not fulfill the goals of the distributed database – data localization – because it is irrelevant (though it is correct).

The distributed design of a database must result in a *minimal* fragmentation, which is *correct* and *relevant*.

Derived Horizontal Fragmentation

In the Manufacturing Company case, users access data about local departments and employees assigned to these departments. We performed fragmentation of the relation Department in the previous section. Let us consider localization of data about employees. If we start with locating the whole table Employee in the Boston database, then users in Boston will be able to access data about employees working in the Boston departments with the help of the query:

```
SELECT e.*
FROM Department1 d, Employee e
WHERE d.deptCode = e.deptCode;
```

As we can see, the above join query always accesses only those rows of the table Employee that are joined to rows of the fragment table $Department_1$. Therefore, it does not make sense to keep the whole table Employee in the Boston database – rows of employees that do not work in the Boston departments are never

accessed by Boston users. Similarly, the New York and Cleveland users are dealing only with rows of the Employee table that can be joined to the New York and Cleveland fragments, respectively, and they do not need the whole table Employee on their sites.

When two relations, one of which is a parent and another one is a child of the relationship, are used in the same application (this is a very common situation), and the parent relation is fragmented, then it is reasonable to consider fragmenting the child relation by joining it to the corresponding fragment of the parent relation (Figure 5).

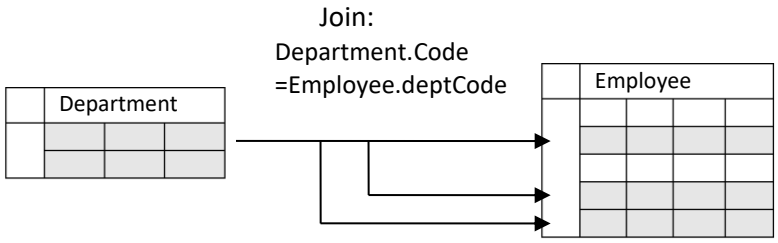

Figure 5. Derived fragmentation of a child relation.

This is called *derived* fragmentation, and each child fragment is defined by the *Semi-join* operation:

$$\text{Child Fragment}_i = \text{Child Relation} \blacktriangleright \text{Parent Fragment}_i$$

The number of derived fragments is equal to the number of primary fragments.

For our case, the derived fragmentation of the relation Employee is defined by the following formulas:

$$\text{Employee}_1 = \text{Employee} \blacktriangleright_{\text{deptCode}} \text{Department}_1$$

$$\text{Employee}_2 = \text{Employee} \blacktriangleright_{\text{deptCode}} \text{Department}_2$$

$$\text{Employee}_3 = \text{Employee} \blacktriangleright_{\text{deptCode}} \text{Department}_3$$

Employee_1 (Boston):

ID	empName	emplType	deptCode	titleCode
2	Adam	Consultant	001	T3
4	Peter	Full-time	003	T2

Employee_2 (New York):

ID	empName	emplType	deptCode	titleCode
1	John	Full-time	002	T1
5	Scott	Consultant	002	T1
6	Susan	Full-time	005	T5

Employee$_3$ (Cleveland):

ID	empName	emplType	deptCode	titleCode
3	Mary	Part-time	004	T4
7	Alex	Part-time	004	T2

The diagram of the derived fragmentation is shown in Figure 6. Each of the databases has a primary fragment of the relation Department and the corresponding derived fragment of the relation Employee. Each derived fragment is the child of the corresponding primary fragment.

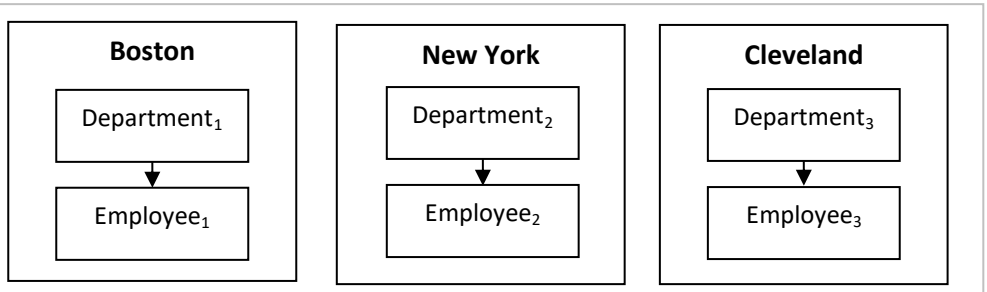

Figure 6. Diagram of derived fragmentation.

Sometimes a relation has more than one fragmented parent relation. Consider the situation, when in addition to fragmentation of the relation Department, the relation Title is fragmented by conditions on the attribute salary (e.g., the Boston site is interested in titles and employees with high salary, while the New York site is working with titles and employees with low salaries):

P_1 : salary > 30000

P_2 : salary ≤ 30000

The relation Employee is the child of two fragmented relations: Department and Title. From which primary fragmentation should we derive the fragmentation of Employee? Once again, we have to analyze user requirements. How will data from the table Employee be used more frequently – in a join with the table Title or in a join with the table Department? If the queries with the join of Employee and Department are more frequent, then we will decide to derive the fragmentation of the relation Employee from the fragmentation of Department. Figure 7 shows the diagram of this fragmentation. Access to the data about local departments and employees is localized: for example, the retrieval of data about the Boston departments and employees uses the Boston fragments Department$_1$ and Employee$_1$. However, access to the data about titles and employees is distributed, e.g. the query about employees with high salaries involves the fragment Title$_1$ and all three fragments of the relation Employee.

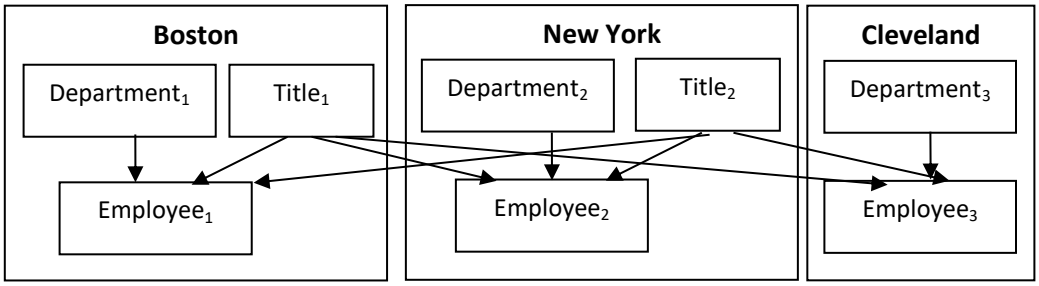

Figure 7. Diagram of complex derived fragmentation.

An additional consideration is the amount of data transferred between databases – fragmentation that requires minimal data transfers is preferable. If it is difficult to make a decision about a derived fragmentation based on access frequencies and amounts of transferred data, it is recommended to choose the fragmentation with a simpler join condition.

The correctness of the derived fragmentation follows from the correct primary fragmentation and referential integrity constraint.

Reconstruction of the derived fragmentation is performed by the *Union* operation:

$$\text{Employee} = \text{Employee}_1 \cup \text{Employee}_2 \cup \text{Employee}_3$$

Derived fragmentation is often beneficial because it localizes logically related data, which is used together. Derived fragmentation is illustrated by a simpler diagram of fragmentation, which in turn illustrate intra- and inter- query parallelism. For example, for the derived fragmentation in Figure 7, the global application that needs the data about all departments and all employees of the company can distribute a global query to the sites and execute parts of the query in parallel (the parts of the query do not depend on each other). For the same derived fragmentation, the local applications can be executed in parallel and independently because each of them uses data only from the local site.

Vertical Fragmentation

Information Requirements

The goal of vertical fragmentation is to keep close attributes – attributes that are accessed together in applications – in the same fragment. It is difficult to define the closeness of attributes formally; often neither designers nor users can specify it. The closeness of attributes for vertical fragmentation (called affinity) is calculated based on the following information requirements:

- Usage of attributes in applications.
- Frequency of execution of applications on different sites.

The Correctness of Vertical Fragmentation

Vertical fragmentation is defined by the relational operation *Projection*:

$R_i = \Pi_{Xi}(R)$, where X_i is a subset of attributes of R.

The rules of correctness for the vertical fragmentation are:

1. *Completeness*. Each attribute must be included in at least one fragment.
2. *Disjointness*. Each non-key attributes must be included in only one fragment; the primary key attributes are included in each vertical fragment.

The initial relation is restored from vertical fragments with the help of the relational operation *Join*.

Case 6.

> *The New York users need data about the names of all employees and codes of the departments to which employees are assigned. The Cleveland users are working with data about the types and title codes of all employees.*

The following vertical fragmentation will localize data for this case:

$$Employee_1 = \Pi_{ID,\ empName,\ deptCode}(Employee)$$

$$Employee_2 = \Pi_{ID,\ empType,\ titleCode}(Employee)$$

$Employee_1$ (New York):

ID	empName	deptCode
1	John	001
2	Adam	002
3	Mary	003
4	Peter	001
5	Scott	002
6	Susan	005
7	Alex	004

$Employee_2$ (Cleveland):

ID	emplType	titleCode
1	Full-time	T1
2	Consultant	T3
3	Part-time	T4
4	Full-time	T2
5	Consultant	T1
6	Full-time	T5
7	Part-time	T2

The initial relation is restored by joining the fragments:

$$Employee = Employee_1 \blacktriangleright\blacktriangleleft_{ID} Employee_2$$

Hybrid Fragmentation

Applying fragmentations of different types to the same relation is called hybrid (also mixed or nested) fragmentation.

Case 7.

> *Users in each city process data about the local departments and the names of local employees. In Cleveland, they have an additional office that works with data about the types and title codes of all employees, and this office has its own database server.*

Local users of each city do not need all the data about local employees, only the ID and name. They also need the department code to associate an employee with the local department to which the employee is assigned.

We can start with the vertical fragmentation of the relation Employee to separate the data about employees that is needed by all local users from data needed by the additional application in Cleveland, and then perform horizontal fragmentation to localize data about departments and employees.

Step1: vertical fragmentation of Employee is similar to the fragmentation of the Case 6.

$$Employee_1 = \Pi_{ID,\ empName,\ deptCode}\ (Employee)$$

$$Employee_2 = \Pi_{ID,\ empType,\ titleCode}\ (Employee)$$

Step 2: primary horizontal fragmentation of Department.

$$Department_1 = \sigma_{location\ =\ 'Boston'}\ (Department)$$

$$Department_2 = \sigma_{location\ =\ 'New\ York'}\ (Department)$$

$$Department_3 = \sigma_{location\ =\ 'Cleveland'}\ (Department)$$

Step 3: derived horizontal fragmentation of the vertical fragment $Employee_1$.

$$Employee_{11} = Employee_1 \blacktriangleright_{deptCode} Department_1$$

$$Employee_{12} = Employee_1 \blacktriangleright_{deptCode} Department_2$$

$$Employee_{13} = Employee_1 \blacktriangleright_{deptCode} Department_3$$

The relation Employee can be restored from the fragments with the help of the union and join operations:

$$Employee = (Employee_{11} \cup Employee_{12} \cup Employee_{13}) \blacktriangleright\blacktriangleleft_{ID} Employee_2.$$

Data is distributed in the following way.

Boston database:

$Department_1$

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
003	Marketing	Boston	Production

$Employee_{11}$

ID	empName	deptCode
2	Adam	001
4	Peter	003

New York database:

$Department_2$

deptCode	deptName	location	deptType
002	Budget	New York	Business
005	Purchasing	New York	Business

$Employee_{12}$

ID	empName	deptCode
1	John	002
5	Scott	002
6	Susan	005

Cleveland database in the second office:

Department$_3$

deptCode	deptName	location	deptType
004	Database Support	Cleveland	IT

Employee$_{13}$

ID	empName	deptCode
3	Mary	004
7	Alex	004

Cleveland database in the first office:

Employee$_2$

ID	emplType	titleCode
1	Full-time	T1
2	Consultant	T3
3	Part-time	T4
4	Full-time	T2
5	Consultant	T1
6	Full-time	T5
7	Part-time	T2

The relation Title was not fragmented because there were no user requirements that would make the distribution of this relation relevant. Let us discuss the allocation of the table Title. The relation Title is a parent to the relation Employee; for this fragmentation it is a parent to the fragment Employee$_2$ because the attribute titleCode, which is the foreign key to Title, is included in the fragment Employee$_2$. Because the table Title is used together with the table Employee$_2$, it has to be located in the same database – the first office in Cleveland.

Diagram for this hybrid fragmentation is shown in Figure 8.

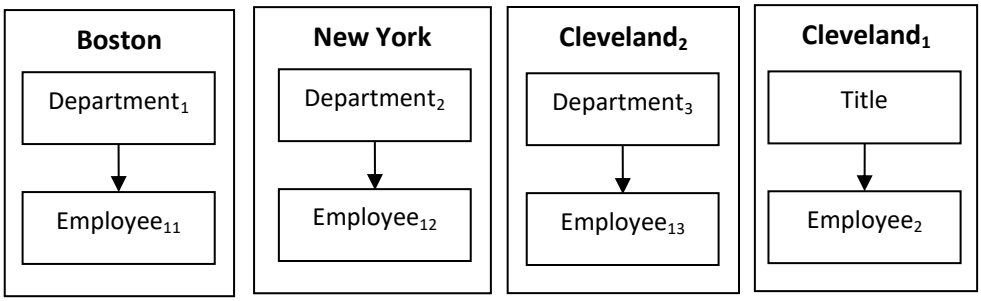

Figure 8. Diagram of the hybrid fragmentation.

Allocation and Replication

Allocation

Distributed design includes the allocation of fragments and relations in individual databases comprising the distributed database. Some fragments and relations can be replicated in several databases. Decisions about the allocation and replication of data are determined by the primary goals of distributed databases: improving performance and increasing data availability through data localization.

The allocation of *fragments* is defined by the fragmentation itself – we fragment in order to localize data. Therefore, each fragment goes to the database where the conditions of data define access for users directly

working with it. For example, because users of Boston request data about local departments with the help of queries constrained by … WHERE location = 'Boston', we defined the predicate p_1: location = 'Boston', and therefore, the fragment $Department_1$ obtained by this predicate will be allocated in the Boston database. Derived fragments are allocated in the same database as their corresponding primary fragments.

The goal of allocation of *unfragmented relations* is the same as the general goal of distributed design – to localize data and minimize data transfer costs. For example, in the case of the hybrid fragmentation, the unfragmented relation Title is allocated in the database of the first Cleveland office because it is used together with $Employee_2$, which is located there.

For the case of derived fragmentation shown in Figure 6 we did not discuss the allocation of the unfragmented relation Title. This relation can be used locally together with each fragment of the relation Employee. Because we do not expect the table Title to be large and updated often, it will not be expensive to replicate it on each site, giving local access to it for each application. In this case, replication enforces the localization of data – see Figure 9 with replicas of Title at New York and Cleveland databases.

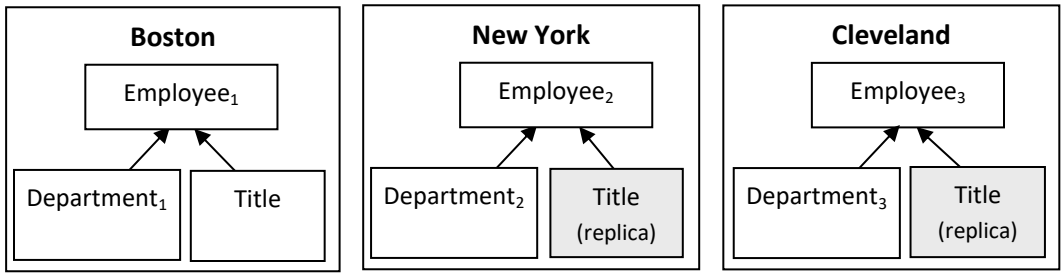

Figure 9. Replication of unfragmented relations.

When making decisions about allocation and replication, the following factors are taken into consideration:

- *Sites where relations and fragments are used.* A fragment is allocated on the site, where data from it is used, and unfragmented relation is allocated on the site where it is most frequently used.
- *Size of relations and fragments and amounts of requested data*. In order to localize data it is often beneficial to replicate a relation of a modest size. A fragment or a relation is allocated to minimize the amount of data transferred between databases.
- *Expected frequency of data modifications*. It is not recommended to replicate a relation that is often modified – modifications of several copies of data located in different databases may worsen the performance of modifying queries.

Replication

The main purposes of replication are improving data availability and reliability, and localizing data. If we want data to be available as the business requires it (often it is 24/7), we will want to have data in a separate database and be synchronized with the data in the store we are working on. Further in this chapter we will show that in some cases of distribution it is beneficial to replicate some data to fully localize data access.

The most important support problem of replicated data is the synchronization of replicas. In some cases, replicated data has to be updated immediately after the modification of the source data; such replication is

called *synchronous*[1]. An alternative to synchronous replication is *asynchronous* replication, where the update of replicated data is delayed (the delay may be from seconds to hours or days). Asynchronous replication is used when the immediate modification of replicas is not beneficial to the performance of the distributed database or when immediate modification is not possible (e.g., the remote site is not available).

In the replicated database it is important to define the sites from which replicated data can be modified; such sites are called *master sites*. Sites on which data is available only for reading are called *slave* or *snapshot*. Figure 10 shows the scheme of access to data in the replicated database with two master and one slave sites. When data is modified on one of the master sites, modifications are replicated on the two other sites of the database: the master and the slave. The data can be read from all three sites.

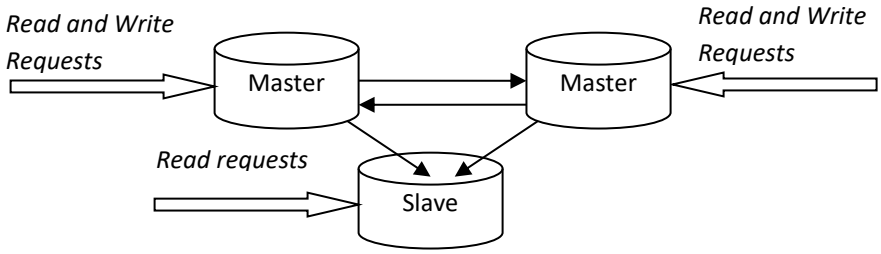

Figure 10. Synchronization of replicas on master and slave sites.

Replication in Oracle

Oracle supports multi-master and snapshot replication. Replicas are defined and supported by special database tools.

Additionally, snapshot replication of separate tables can be implemented with the help of the Oracle *materialized view* object (the materialized view is the advanced snapshot, which can be refreshed when the source table is modified).

An interesting feature of Oracle replication is *procedural replication*. When transactions change large amounts of replicated data, the network is overloaded by transfers of changed rows to synchronize replicas. In order to avoid this, data modifications are implemented in stored procedures. Then, such stored procedures are replicated on different sites. When the modification procedure is started on one site, the replicated copies of this procedure are started on other sites and modify the data in the same way as the data are modified on the initiation site. Therefore, only calls to procedures are passed across the network

Preserving Semantics of the Relational Model

Integrity Control

The semantics of the database should not be affected by the way data are stored. One of the mentioned problems of the distributed database is the necessity of additional support of relational semantics. The

[1] Note that synchronous replication can be supported with the help of triggers where for each operation in one database a similar operation is performed in other databases.

semantics of the relational model are expressed by the structure of the relations and two integrity constraints: the primary and foreign keys. Correct fragmentation preserves the structure of the relations. However, integrity constraints, as we will show, are not preserved in the fragmented database and require special attention.

The Primary Key

In Case 1, the relation Department was fragmented into three fragments by conditions on the column location. For each fragment of Department, the attribute deptCode is defined as the primary key to support the constraint locally:

$Department_1$ (deptCode, deptName, location, deptType)

$Department_2$ (deptCode, deptName, location, deptType)

$Department_3$ (deptCode, deptName, location, deptType)

However, local constraints do not prevent duplicating values of deptCode in the distributed database. For example, though we have the department with deptCode '001' in the Boston database, it is possible to add another department with deptCode '001' to the New York fragment:

Boston:

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
003	Marketing	Boston	Production

New York:

deptCode	deptName	location	deptType
002	Budget	New York	Business
005	Purchasing	New York	Business
001	…	…	…

Most distributed DBMSs do not support global integrity constraints, and to preserve the semantics of the relational model we need to implement its support. Usually, in addition to integrity constraints, data integrity is supported by special procedural database objects called *Triggers*. Triggers are special procedures that are defined on particular events of a table and they start every time these events happen. A global primary key of the relation Department can be implemented with the help of a trigger for the insert event on each fragment of the relation. For every inserted row, the trigger will check the other fragments for duplication of the deptCode of the newly inserted row and if it finds a duplicate, then the insertion will be rejected (see the section of this chapter with examples for illustration of triggers for integrity support).

Let us discuss a different situation with fragmentation of the relation Department: users in Boston work with data about departments with codes below '003', users in New York work with data of all other departments, and users in Cleveland do not work with departments' data. This leads us to the following fragmentation of Department:

$$Department_1 = \sigma_{deptCode < '003'} (Department)$$
$$Department_2 = \sigma_{deptCode >= '003'} (Department)$$

Each fragment inherits the primary key of the initial relation to support uniqueness locally. Do we need to support a global primary key in this case? Obviously not – any record added to the second fragment cannot have a duplicate in the first fragment, and any record added to the first fragment cannot have a duplicate in the second fragment. Disjointness of predicates, and in this case predicates that are defined on the attribute of the primary key, guarantee this.

> **If predicates, by which a relation is fragmented, include an attribute of the primary key, then global integrity is enforced by the disjointeness of the predicates. When fragmentation is performed by conditions based on the non-key attributes, then additional measures are required to support global integrity.**

If the primary key of a derived fragment includes the foreign key by which the derived fragmentation is performed, then the primary key of the derived fragment is enforced globally by the fragmentation. Otherwise, like in the case of the derived fragmentation of the relation Employee of the Manufacturing Company case, additional global support of the primary key of the derived fragment is required.

Support of global integrity with the help of triggers compromises the localization of data. When performing local data processing, an application has to access remote databases, and, therefore, becomes dependent on their availability. The local performance of modifying operations becomes worse. To avoid these complications, it is recommended to apply other measures. For example, for fragmentation of the relation Department based on the location attribute, we can agree that each database uses its own pool of values for the attribute deptCode. This agreement will be enforced by the CHECK constraint on each fragment of the relation Department, e.g. if in the New York office codes of departments cannot exceed '333', then the New York fragment will be created as follows:

```
CREATE TABLE Department (
    deptCode CHAR(3) PRIMARY KEY CHECK (deptCode <='333'),
    . . . );
```

In some cases, the uniqueness of a primary key is supported outside the particular business. For example, if instead of IDs of employees, the Manufacturing Company used their social security numbers, the uniqueness of which is supported globally across the country, there will be no need to support the uniqueness of IDs across the company. However, in the case of the distributed solution, the uniqueness of social security numbers does not guarantee that the same employee is not assigned to several departments in different cities, and we will have to use triggers on the independent databases comprising our distributed database.

The Foreign Key

The relation Employee has two foreign keys: deptCode and titleCode. In the case of the derived fragmentation of the table Employee discussed earlier, the fragmentation of the relation Employee was derived from the fragmentation of the relation Department and each fragment of Employee contains employees assigned to the departments of the primary fragment of Department located on the same site. Each fragment of Employee has the local fragment of Department as a parent. This is clearly seen in the diagram of the fragmentation in Figure 7.

The foreign key constraint on a field that participates in derived fragmentation has to be defined as a foreign key on each derived fragment to the corresponding primary fragment.

The situation with the second foreign key – titleCode – is different and depends on several design decisions. First, we need to know where the parent relation Title is located.

If we decide to replicate the unfragmented relation Title on each site, then this parent relation will be fully available to each child fragment of the relation Employee. Such a distributed approach will enable us to declare the attribute titleCode of each fragment as the foreign key to the corresponding (located on the same site) replica of the relation Title (see Figure 9).

Figure 11 shows the diagram of fragmentation for a different distributed solution, when the relation Title is located on the Cleveland site only and the New York and Boston fragments of Employee have no local parent table for the foreign key titleCode. The foreign key cannot be declared to a remote parent table and, therefore, in the New York and Boston databases this constraint has to be implemented with the help of a trigger on each fragment of the relation Employee (see the section of this chapter with examples). For every insert of an employee, the trigger will check the Cleveland table Title for the existence of the title code of the inserted row; if the title code is not found, then the insertion will be rejected. In the Cleveland database, the fragment of Employee can reference the local relation Title.

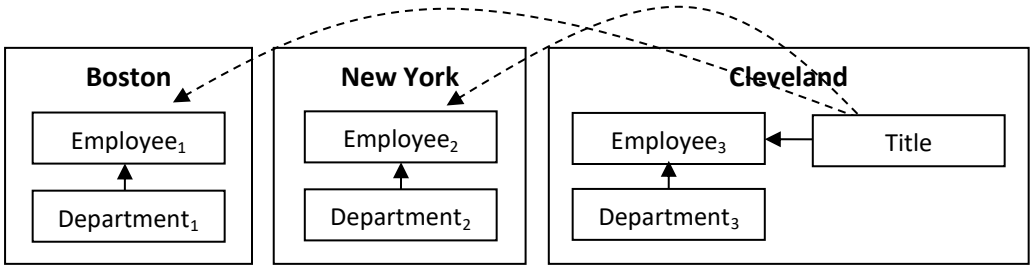

Figure 11. Referential integrity in a distributed database.

The support of a foreign key that is not involved in a derived fragmentation depends not only on the location of the parent relation, but also on whether it is fragmented or not. Consider a situation when the relation Title is fragmented by conditions on the salary attribute:

$$Title_1 = \sigma_{Salary < 20000} (Title)$$

$$Title_2 = \sigma_{Salary >= 20000} (Title)$$

The fragments are located in the New York and Cleveland databases, and the diagram of fragmentation is shown in Figure 12.

The diagram of fragmentation in this case is even more complicated than in the previous example: each fragment of Employee is related to two fragments of Title. Referential integrity for titleCode has to be supported with the help of triggers. In the Boston database, the trigger will have to check two remote

databases – the New York and Cleveland – for existence of the title code of a newly inserted row. Triggers in New York and Cleveland will have to check the local and remote fragments of Title. If the value of the title code is not found, the insertion of the new row is rejected. Note that for this situation, for fragments of Employee in the New York and Cleveland databases, the foreign keys to the local fragments of the relation Title cannot be defined because they would limit the title codes of employees to the codes of their local fragment.

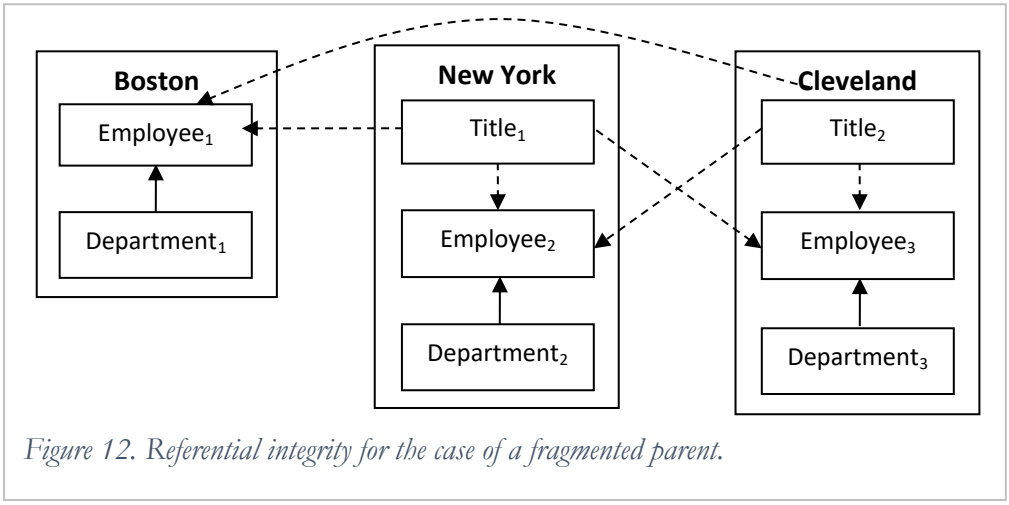

Figure 12. Referential integrity for the case of a fragmented parent.

The implementation of the foreign key constraint on a column that does not participate in derived fragmentation depends on where the parent relation is located and whether it is fragmented or not. The foreign key constraint cannot be established to the parent relation on another site or to a fragment of the parent relation.

The Integrity of Vertical Fragmentation

The main problem in a vertically fragmented relation is the synchronization of insert and delete operations on the fragments. For example, for the Case 6 we have to ensure that if a row is inserted in one fragment of the relation Employee, then the row for the same employee is added to the second fragment. Similarly, if data about a particular employee are being deleted from the database, the data must be deleted from both vertical fragments of the relation Employee. We can implement this with the help of special procedures for inserting and deleting data[2]; each of the procedures will perform a corresponding action on the fragments in both databases. Deletes can also be implemented via triggers.

Similar measures can be discussed for updates of the key attributes that are included in all vertical fragments. However we must remember that it is recommended to choose the key attributes of a relation so that they do not change their value during the lifetime of a row.

Other Constraints

Most DBMSs support the NOT NULL and CHECK constraints. These are single-row constraints – they are

[2] Direct `INSERT` and `DELETE` operations on the tables have to be forbidden, and users only will have permission to execute the procedures. See Chapter 4 on security that explains how to accomplish this.

separately evaluated on each row of a relation (as opposed to the primary and foreign key constraints, the evaluation of which involves other rows of the relation itself or the parent relation).

The NOT NULL constraint does not require any special support; it is applied on each fragment of a relation.

The CHECK constraint may change if a relation is fragmented by conditions on the column with the constraint. Assume that there is the CHECK constraint on the column salary of the relation Title:

```
…CHECK Salary BETWEEN 10000 AND 50000.
```

For fragments of Title:

$$Title_1 = \sigma_{Salary < 20000} (Title)$$

$$Title_2 = \sigma_{Salary >= 20000} (Title)$$

the CHECK constraint will be simplified:

$Title_1$: CHECK Salary >= 10000

$Title_2$: CHECK Salary <= 50000.

Transparency of Distribution

It is important to make the distributed database appear to users as a centralized database, or, in other words, to make the distribution transparent to users. In Chapter 2 we discussed the ways of achieving transparency of physical and logical details of the database with the help of synonyms, views, and procedures.

Transparency of distribution is needed for users who work with data from different databases; we call such users global. For these users, we need to reconstruct data from fragments and hide the allocation of fragments and relations. The simplest way to achieve this transparency is to use views with the reconstruction of all data: the union of fragments for horizontal fragmentation and the join of fragments for vertical fragmentation. Such views have to be created in the database where the users perform global access to the data. For the Manufacturing Company case, transparency of distribution has to be implemented in the New York database, where the users access data about all departments and employees of the company.

Examples: Implementing a Distributed Database in Oracle

Let us follow the steps of distributed design for the Manufacturing Company case. Analysis of company's applications and local needs in data led us to the following distributed design.

Primary fragmentation of the relation Department:

$$Department_1 = \sigma_{location = \text{'Boston'}} (Department)$$

$$Department_2 = \sigma_{location = \text{'New York'}} (Department)$$

$$Department_3 = \sigma_{location = \text{'Cleveland'}} (Department)$$

Derived fragmentation of the relation Employee:

$Employee_1$ = Employee ▶ $Department_1$

$Employee_2$ = Employee ▶ $Department_2$

$Employee_3$ = Employee ▶ $Department_3$

The relation Title is left unfragmented.

Considering the needs in data, we decide on the allocation of fragments and relations that is shown in Figure 13:

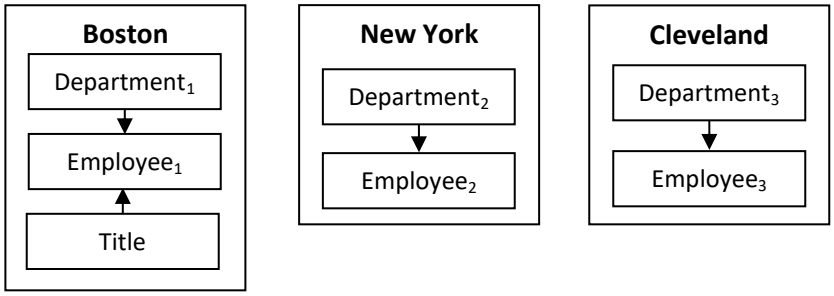

Figure 13. Allocation of fragments and unfragmented relations of the case.

Because the data from the table Title may be used by every site and the table is not large and it does not change often, it makes sense to replicate it on each site. However, for the demonstration of support of global referential integrity with the help of triggers we will allocate Title in the Boston database.

Because the New York and Cleveland databases are similar, we will continue our discussion for the Boston and New York sites only. Assume that in the New York database there is a database link to the Boston database. This link is needed for global users and implementation of the global integrity:

```
CREATE PUBLIC DATABASE LINK boston USING boston.ourcompany.us.com;
```

To implement global integrity in the Boston database, we will need the database link to the New York database:

```
CREATE PUBLIC DATABASE LINK ny USING ny.ourcompany.us.com;
```

Implementation of the distributed database will include the following steps:

1. Create tables for fragments and unfragmented relations in each database according to the design. The fragmentation of Department is enforced by the CHECK constraint on the attribute location. The correctness of the fragments of Employee is supported by referential integrity on deptCode of each fragment.
2. Create a trigger[3] on each fragment of Department to support the global integrity of deptCode (because primary fragmentation of Department is based on the non-key attribute location).

[3] We apply this type of global integrity support to demonstrate triggers. As mentioned before, it is better to constraint values of `deptCode` in each database with the help of the `CHECK` constraint.

3. Create a trigger on each fragment of Employee to support the global integrity of ID (because Employee is fragmented by the non-key attribute deptCode).
4. Implement the referential integrity of deptCode for Employee (fragments of Employee are derived from fragments of Department, and we can define deptCode of each Employee fragment as the foreign key to the corresponding fragment of Department).
5. Implement the referential integrity of titleCode. It is supported differently on two sites: the fragment of Employee in the Boston database can reference Title because it is located on the same site, while for the fragment of Employee in the New York database we will simulate referential integrity by the use of a trigger.
6. Implement local transparency.
7. Implement transparency of distribution in the New York database.

	Boston	**New York**

1.	`CREATE TABLE Department (`
	`deptCode CHAR(3) PRIMARY KEY,`
	`deptName VARCHAR2(20) NOT NULL,`
	`location VARCHAR2(25) CHECK`
	`(location = 'Boston'),`
	`deptType VARCHAR2(15));`
	
	`CREATE TABLE Employee (`
	`ID NUMBER PRIMARY KEY,`
	`empName VARCHAR2(30) NOT NULL,`
	`empType VARCHAR2(10),`
	`deptCode CHAR(3),`
	`titleCode CHAR(2));`
	
	`CREATE TABLE Title (`
	`titleCode CHAR(2) PRIMARY KEY,`
	`titleDescription VARCHAR2(25),`
	`salary NUMBER);`
		`CREATE TABLE Department (`
		`deptCode CHAR(3) PRIMARY KEY,`
		`deptName VARCHAR2(20) NOT NULL,`
		`location VARCHAR2(25) CHECK (location`
		`= 'New York'),`
		`deptType VARCHAR2(15));`
		
		`CREATE TABLE Employee (`
		`ID NUMBER PRIMARY KEY,`
		`empName VARCHAR2(30) NOT NULL,`
		`empType VARCHAR2(10),`
		`deptCode CHAR(3),`
		`titleCode CHAR(2));`

2.	*Create a trigger for support of the global primary key of Department.*	*Create a trigger for support of the global primary key of Department.*
3.	*Create a trigger for support of the global primary key of Employee.*	*Create a trigger for support of the global primary key of Employee.*

4.	`ALTER TABLE Employee`
	`ADD CONSTRAINT fk_employee_department`
	`FOREIGN KEY (deptCode) REFERENCES`
	`Department;`
		`ALTER TABLE Employee`
		`ADD CONSTRAINT fk_employee_department FOREIGN`
		`KEY (deptCode) REFERENCES Department;`

5.	`ALTER TABLE Employee`
	`ADD CONSTRAINT fk_employee_title FOREIGN`
	`KEY (titleCode) REFERENCES Title;`
		Create a trigger for support of the foreign key to Title in the Boston database.

6.	`CREATE VIEW vw_Employee AS` `SELECT * FROM Employee;`	`CREATE VIEW vw_Employee AS` `SELECT * FROM Employee;`

7.		`CREATE VIEW vw_allEmployees AS`
		`SELECT * FROM Employee`
		`UNION`
		`SELECT * FROM Employee@Boston;`

The trigger in the Boston database for support of the global primary key of Department can be implemented in the following way:

```
CREATE OR REPLACE TRIGGER tr_department_code
BEFORE INSERT OR UPDATE OF deptCode ON Department
FOR EACH ROW
DECLARE
   sDeptCode VARCHAR2(3);
BEGIN
   SELECT deptCode INTO sDeptCode
   FROM Department@ny WHERE deptCode = :NEW.deptCode;
   -- looking for deptCode of the inserted record in the table
Department
   -- on another site. If we are here, then the select was successful.
   -- If we have a duplicate, the insert must be rejected.

      RAISE_APPLICATION_ERROR (-20999, ' Duplication of deptCode');
EXCEPTION
   WHEN NO_DATA_FOUND THEN

   -- If we are here, then select did not return any data.
   -- No action is needed.
      NULL;
END;
```

The trigger in the New York database for support of the global foreign key titleCode of Employee could be implemented in the following way:

```
CREATE OR REPLACE TRIGGER tr_employee_title
BEFORE INSERT OR UPDATE OF titleCode ON Employee
FOR EACH ROW
DECLARE
   sTitleCode VARCHAR2(2);
BEGIN
   SELECT titleCOde INTO sTitleCode
   FROM Title@Boston WHERE titleCode = :NEW.titleCode;
   -- looking for titleCode of the inserted record in the table Title.
   -- If we are here, then the select was successful.
   -- The record can be inserted.
EXCEPTION
   WHEN NO_DATA_FOUND THEN
   -- If we are here, select did not return any data.
   -- Integrity is violated. Insert must be rejected.
 RAISE_APPLICATION_ERROR (-20999, ' Non-existing titleCode');
END;
```

If we chose another approach and decided to replicate the table Title on each site, then we could implement replication with the help of materialized views. Assume that the Boston site is in charge of supporting data about titles and it is the master site for this data. As in the previous design, the table Title will be created in the Boston database. The New York and Cleveland databases will have replicas of this table implemented as

materialized views:

```
CREATE MATERIALIZED VIEW Title
  PARALLEL
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT
  AS SELECT * FROM Title@Boston;
```

Additionally, we will need to create a view log on the master tables – the table Title in the Boston database as shown below:

```
CREATE MATERIALIZED VIEW LOG ON Title;
```

The option REFRESH ON COMMIT causes the view to be refreshed every time the table Title in the Boston database is modified. Applications on the New York and Cleveland sites will be able to use data from Title without remote accesses to the Boston database.

Starting with Oracle 9i, the materialized view has a primary key and technically can be referenced by a table. Therefore, the titleCode attribute of fragments of Employee in the New York and Cleveland databases can be implemented as the foreign key to the corresponding materialized view. Each site will be fully localized and independent from others.

Summary

Distributing data across several database servers is one of the options used in performing the physical design of a database. The distributed database can be beneficial for a company that runs applications that need only specific portions of the data. By localizing the data and bringing the required portions of data directly to the users of the applications, the distributed database can significantly improve the performance of such applications. The distributed design can improve data reliability and availability by replicating data in several of the databases that are part of the distributed database; it can provide a better scalability of the database and database applications.

The decision on the distribution of data requires a thorough analysis of the company's structure, the different needs of users in data, and the organization of the company's network.

The design of a distributed database includes fragmentation of relations of the logical model, allocation of fragments and unfragmented relations, and replication. Fragmentation, allocation and replication are performed to achieve the main goals of the distributed database – localization of data and improvement of performance of the most important applications, and increased reliability and availability of data.

Relations can be fragmented horizontally and vertically. Each fragmentation must be complete and disjoint. If a parent relation is fragmented horizontally, it is often reasonable to consider derived horizontal fragmentation of a child relation. A global relation is restored by the union of horizontal fragments and by the join of vertical fragments.

The distributed database must be transparent to users – they should see it as a centralized database. The transparency of distribution must be implemented by database programmers.

Distributed DBMSs do not support global relational constraints. Preserving the semantics of the relational

model is one of the problems of implementation of the distributed database. Relational semantics is implemented with the help of special procedures – triggers, or by applying organizational measures and limiting possible values of the key attributes in different databases.

Though it was mentioned that one of the problems of distributed databases is the absence of a straightforward methodology of design, the following sequence of steps can serve as the design guidance:

- *Analyzing the structure of a company*. Research the topology of a company and its network. Define the locations of the databases.
- *Defining the needs in data*. Examine applications of the company, define the most important and frequently used ones. Understand the applications' needs in data.
- *Designing fragmentation*. Consider horizontal and/or vertical fragmentation of relations that could improve the performance of the company's important applications.
- *Performing derived fragmentation*. When performing horizontal fragmentation of several relations with relationships between them, first consider primary fragmentation of the owner of a relationship, and then proceed with derived fragmentation of the members of the relationship. Remember that derived fragmentation has several benefits: 1) it gives a simple diagram of fragmentation with no or few relationships between fragments and relations located in different databases; 2) it enhances the parallel execution of global queries; 3) it does not require special support of referential integrity. In summary, derived fragmentation results in a good and "clean" localization of data.
- *Allocating and replicating fragments and relations*. Allocate fragments according to distributed design and allocate unfragmented relations to enhance data localization. Consider replicating unfragmented relations that are used on different sites, are not too large, and are not modified too often. Replicate important data that are crucial for the functioning of the system.
- *Supporting relational semantics*. Implement global support of integrity constraints of the database.
- *Making distribution transparent*. Implement transparency of distribution, make the distributed database appear like a centralized database to the users.

Review Questions

- What are the promises and problems of distributed databases?
- How can distribution increase the reliability of the database?
- How can distribution improve the performance on the database?
- What is localization of data? How can localization be achieved?
- What are the information requirements for horizontal and vertical fragmentations?
- How do you define correctness for horizontal and vertical fragmentation?
- How many vertical fragments can you have for the relation R(A, B, C)?
- When is derived fragmentation possible? What are the benefits of derived fragmentation?
- How you restore a relation from its fragments?
- What are the benefits and disadvantages of replication?
- What transparencies must be implemented in the distributed database?
- How do you implement transparency of allocation and fragmentation in the distributed database?
- Does distribution change the semantics of the database?
- Which distributed situations require special support of global relational semantics?

Practical Assignments

1. It is know that the Manufacturing Company discussed in this chapter is planning to expand and will be opening several new offices in different cities. These offices will not have database servers and data for

them will be processed on the New York server. For this situation, redefine the predicates of horizontal fragmentation of the relation Department from the case 1.

2. For the Car Rental Company case explain why the following fragmentations are either incorrect or irrelevant:

   a) p_1 : city = 'X' AND type = 'truck'
      p_2 : city = 'X' AND type <> 'truck'
      p_3 : city = 'Y' AND type = 'truck'
      p_4 : city = 'Y' AND type <> 'truck'
      p_5 : city = 'Z' AND type = 'truck'
      p_6 : city = 'Z' AND type <> 'truck'

   b) p_1 : city = 'X' AND type = 'truck'
      p_2 : city = 'Y' AND type <> 'truck'
      p_3 : city = 'Z' AND type = 'truck'

   c) p_1 : year = '2018'
      p_2 : year = '2019'
      p_3 : year <= '2020'

3. The relation Title was fragmented by predicates:

   P_1 : Salary > 30000
   P_2 : Salary < 30000

   What rules of fragmentation are violated? How you will define the correct fragmentation for this case? Explain your answer.

4. What rules of fragmentation are violated in the following vertical fragmentations of the relation R (A, B, C, D):

   a) $R_1 = \Pi_{A, B, C}(R)$
      $R_2 = \Pi_{A, B, D}(R)$

   b) $R_1 = \Pi_{A, B, C}(R)$
      $R_2 = \Pi_{A, D}(R)$

5. What rules of fragmentation are violated in the following horizontal fragmentations of the relation R (A, B, C) where the attribute C can be any integer number:

   a) p_1 : 10 <= C AND C < 100
      p_2 : C >= 100

   b) p_1 : 10 <= C AND C < 100
      p_2 : C >= 100
      p_3 : C < 10

   c) p_1 : 10 <= C AND C <= 100
      p_2 : C >= 100

6. For cases from the Appendix 1.
    a) Design the distribution of data.
    b) Suggest an appropriate allocation and replication of data approach.
    c) Build the physical model of the distributed database in Oracle.
    d) Implement transparency of the distributed database.
    e) Implement integrity constraints for your distributed solutions.

7. In Chapter 2 we discussed that organizing storage, e.g. by clustering, in some cases makes data management more complex or expensive. If for example the table `Employee` is clustered by `deptCode`, and an employee is transferred to another department, then the system needs to do more work than in the case of the heap storage of the table. In case of a distributed database the burden of data management in such situations will be on the developer. For a case from the assignment 6 think how you will manage situations when due to changes of the value of the attribute by which you fragmented a table a record has to be moved from one fragment to another. Issues to consider: 1) your global support of the primary key of the table; 2) the foreign key of the child table(s); 3) the need to maintain the history of the business (we usually cannot simply delete records).