2020

Chapter 2: 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 2. The Physical Data Model

A *physical data model* is a representation of a database design based on particular user requirements which will include the database artifacts required to achieve the goal of implementing a well-performing, reliable, and easy to use and manage database. Most business rules are implemented in the relational data model through the structure of relations and integrity constraints. The physical data model specifies how the storage of data of this particular logical structure is organized and managed. This model largely defines the database performance, as well as the ease of use and maintenance.

The design of the physical model involves a particular DBMS and requires knowledge and understanding of how data are stored and managed by this DBMS. Furthermore, when performing physical modeling, it is necessary to know the specific business rules that define how the database will be used, such as: the expected size of the database, the types of requests to the database and the required performance of these requests, the expected number of users (including the number of users who may work concurrently), and other physical considerations. This chapter describes the basic principles of data storage organization and the benefits of different storage solutions, and demonstrates some approaches to physical design for various database usage scenarios.

Goals of Physical Modeling

The most general goals of physical modeling are to define the physical features of the data, the most important of which are: where data are stored, how storage is organized, what are the data types of the tables' columns, so that users are able to perform all the necessary data operations as quickly as needed. It is also important that the database is easy to use and maintain. For this, knowledge of how the DBMS manipulates data is crucial.

> **The physical model of data is tailored to user requirements with the help of features of the chosen DBMS.**

Often, some user requirements considered during physical design conflict with each other in terms of applying the features of the DBMS for implementing them. For example, data storage organization for the best performance of read queries may compromise the performance of modifying queries. That is why every physical design decision has to be analyzed for not only expected benefits, but also possible disadvantages or complications. In many cases, the final physical data model is the result of compromises – if all user requirements cannot be satisfied in the best possible way, then we choose a solution that is reasonably good for most of them.

Performance and Data Availability

The physical data model has significant impact on database performance. Query processing, performance, and database tuning are discussed in detail in Chapter 5. If described on a very general level, query processing involves locating requested data on disk, fetching the data from disk into memory, processing the data, and then writing the data, possibly modified, back to disk. Locating and fetching data are the most expensive steps in terms of consumption of computer resources and time. The physical model has to support such data storage so that locating and fetching data can be executed in the most efficient way.

Other Goals

It is important that a well performing database is easy to use and maintain, and appropriate physical design can significantly reduce the support and maintenance efforts and costs.

The most important integrity rules are implemented in the relational model (primary key, foreign key). The physical design can enhance the consistency and integrity of the data through specifying the data types of columns, providing additional constraints, and using special features of the DBMS. The more completely the required consistency and integrity are implemented, the easier it is to support and maintain the database.

Tables

The data in relational databases are stored in tables. Usually, tables are built for relations of the relational model. The definition of the table includes the data types of columns, the location of data, how the storage of data is organized, and how the DBMS should maintain the storage and access to data.

Discussions are provided on the Manufacturing Company case (see Appendix 1 with case studies) with the following relational data model (primary keys of relations are underlined, and foreign keys are in italics):

```
Title (titleCode, titleDescription, salary)
Department (deptCode, deptName, location, deptType)
Employee (ID, emplName, emplType, deptCode, titleCode)
```

Data Types of Columns

The column data type defines the values that can be inserted into the column and the kind of operations that can be performed on these values. The SQL standard defines numeric, character, date, and timestamp[1] data types. In addition to these data types, different DBMSs support other data types, in particular, BLOB – Binary Large Object – for representing large amounts of binary data such as images, video, or other multimedia data, or CLOB – Character Large Object – for storing large amounts of character data.

The character and numeric Oracle data types are used to create the following table for the relation Title of the Manufacturing Company case:

```
CREATE TABLE Title (
   titleCode CHAR(2) PRIMARY KEY,
   titleDescription VARCHAR2(15),
   salary NUMBER (7));
```

The column data type is chosen so that it can *represent all possible column values* and allow for *performing necessary operations* on the column. In the above example, the character data type is used for the attribute titleCode because title codes may include letters. Though numeric data from the column salary can be presented by either the character or numeric data types, the latter was used to allow different numeric operations on the column, e.g. calculating the average salary. Although the column type `NUMBER(5)` can accommodate all possible salaries (it is mentioned in the case description that salaries of employees are less than $100,000), it may be not enough to present the results of calculations on the column, e.g. `SUM(salary)`, hence, the column type is `NUMBER(7)`. Often, numeric columns have to be longer than is required by their possible values in order to enable various calculations on them.

Data types can enforce *data integrity*. For the table Title, title codes cannot be longer than two symbols, and the chosen data type guarantees it. However, if title codes were composed of digits only, then we have two choices: 1) use the numeric data type because the character data type would not prevent inserting values with symbols

[1] Added in the new SQL 2003 standard.

other than digits, and 2) use the character data type and apply a specific `CHECK` constraint to ensure that the symbols of the value are digits.

One more consideration for choosing the data type is *economical space usage*. For example, for columns with variable length, like titleDescription, it is better to use the character data type with variable length – `VARCHAR(15)` – in which each value of the column in the database contains as many symbols as it actually has, e.g. the title description 'DBA' is stored as three symbols. In the case of the fixed length character data type – `CHAR(15)` – all values of the column, regardless of their actual length, will be padded with spaces and stored in the database as strings fifteen symbols long.

It is important to understand the details of data types supported in a particular DBMS. For example, Oracle supports two character data types with variable length: `VARCHAR` and `VARCHAR2`. Where appropriate, it is recommended to use `VARCHAR2` as: 1) `VARCHAR` can store up to 2000 bytes and `VARCHAR2` can store up to 4000 bytes; 2) `NULL` values of the column declared as `VARCHAR` will occupy space while `NULL` values of columns declared as `VARCHAR2` will not.

Some DBMSs offer additional possibilities for column management. For instance, a very convenient feature is automatically generating and assigning of a value to a column when inserting data. Imagine a situation when a company wants each employee to have a unique ID, like in the table Employee of the Manufacturing Company case. In a company with thousands of employees, support of IDs can be rather complicated. Using special data types or special database objects that enforce automatic assignment of a new column value can be extremely beneficial. Examples of such features are the `AUTONUMBER` data type in MS Access and the `IDENTITY` column in MS SQL Server. If, for example, the column ID in the table Employee is declared as `AUTONUMBER`, then for every inserted new employee the system generates the next integer number and assigns it to the ID column: '1' for the first inserted employee, '2' for the second, and so on. In this case, the data type guarantees support (inserting values) and the uniqueness of values. In releases before Oracle 12 there was no feature equivalent to `IDENTITY` or `AUTONUMBER`; instead developers had to use the sequence object (is illustrated in the section with illustration of the physical model at the end of the chapter). In Oracle 12 there is the `IDENTITY` feature on the numeric column. For example, we can use one of the options (`GENERATED ALWAYS`) provided by this feature in the definition of the ID column of the table Employee:

```
CREATE TABLE Employee (
  ID NUMBER GENERATED ALWAYS AS IDENTITY,
  emplName VARCHAR2(30),
  …
);
```

Then, we will be simply inserting data about employees while the system will provide the values of ID for these inserts (note that an attempt to insert the ID directly will cause an error):

```
INSERT INTO Employee (emplName, …) VALUES ('John', …);
```

Another interesting feature of Oracle is *virtual columns*[2] – columns that are derived from other columns of the table but are not stored on disc. If for example employees of our case have a bonus that is calculated as a particular percent of the salary, then we can have the following table definition:

[2] Compare this feature with creating a view on the table. Views are introduced in the section 0.

```
CREATE TABLE Title (
   titleCode CHAR(2) PRIMARY KEY,
   titleDescription VARCHAR2(15),
   salary NUMBER (7),
   bonus NUMBER GENERATED ALWAYS AS (ROUND(salary*0.3,2)) VIRTUAL);
```

Note that data is not inserted in the virtual column; we can select from the virtual column as from a regular column:

```
INSERT INTO Title (titleCode, titleDescription, salary)
VALUES ('T1', 'DBA', 60000);

SELECT * FROM Title WHERE titleCode = 'T1';

  TITLECODE   TITLEDESCRIPTION   SALARY     BONUS
 ----------   ----------------   ------     -----
  T1          DBA                60000      18000
```

When choosing the data type, consider whether it allows for the support of all possible column values and operations, enforces the column's integrity, is economical, and makes support of the column easier.

Constraints

DBMSs offer additional table and column constraints that are not included in the relational data model and that can improve data consistency and integrity. The most common constraints are:

- `NOT NULL`: Requires that the column on which the constraint is defined has a value for every row.
- `CHECK`: Defines a condition that is evaluated every time a row is inserted or the columns involved in the constraint are updated: each row has to satisfy the predicate.
- `UNIQUE`: Enforces uniqueness of a column or combination of columns in a table.

`CHECK` and `UNIQUE` constraints, when based on one column, can be declared in line with this column as shown in the example below. When a constraint includes multiple columns, it has to be defined on the table level as for example complex primary keys.

Consider, for example, business rules that require that each title has to have a description and salaries are in a specific range. While the relational model does not reflect these requirements, they can be enforced in the definition of the table Title:

```
CREATE TABLE Title (
   titleCode CHAR(2) PRIMARY KEY,
   titleDescription VARCHAR2(15) NOT NULL,
   salary NUMBER CHECK (Salary BETWEEN 30000 AND 90000));
```

More complicated integrity and consistency rules that cannot be declared through table constraints (each DBMS has specific limitations on what can be included in constraints) have to be implemented with the help of triggers – special database procedures (discussed in the section with illustration of the physical model at the end of the chapter).

Size and Location

When a table is created, the DBMS allocates disk space according to the size specifications. It is important to estimate accurately the expected size and growth rate of the table. The size of the table depends on the expected number of records, the length of one record (or an estimated average length if the record length varies), the amount of free space in data blocks, and some other parameters. Each DBMS offers a methodology of estimating the table size. The expected size and growth rate of the table are used to determine the storage space parameters.

Data Storage

Design of data storage requires an understanding of data storage organization in databases. See Appendix 3 for details of data storage in Oracle.

Storage Hierarchy

Physically, data are stored in data files. Each database has at least one data file. Data files are *logically* organized in storage spaces called *tablespaces*, and storage space for data objects, e.g. tables, is specified in tablespaces and not directly in data files. As a result, a database object can be stored in more than one data file, but always in one tablespace. For example, in Figure 1, the table Title is stored in the tablespace USER_DATA and has extents in two data files: df1.dbf and df2.dbf. Within a data file, each extent is a contiguous set of data blocks (or data pages). A data block is the smallest unit of operation and manipulation of the DBMS – if the system needs to access a particular row of a block, it reads the whole block. In some databases it is possible to have blocks of different sizes, e.g. in Oracle, different tablespaces can have different block sizes. In Oracle, the default size of the block is 8K.

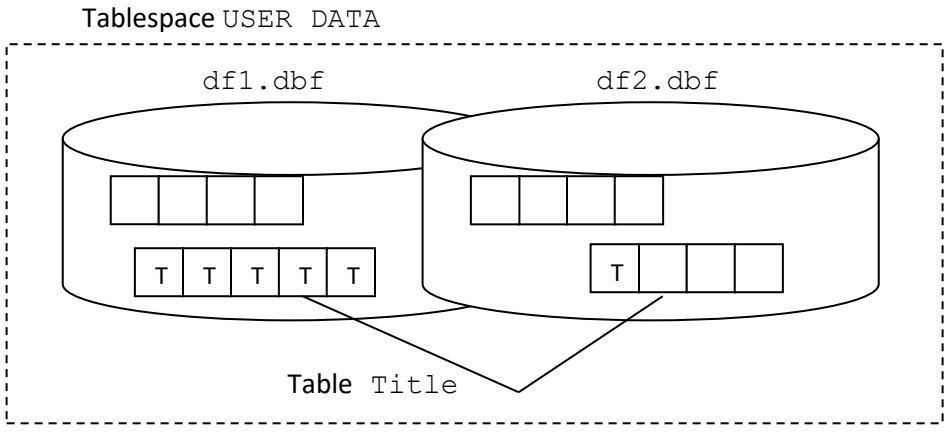

Figure 1. Storage hierarchy.

In summary, data storage consists of tablespaces. Each tablespace is a logical container for one or more data files. Data files contain extents of data blocks with data.

Here are some considerations of leveraging tablespaces to achieve the goals of the physical design:

- Improve performance

- Storing different tablespaces' datafiles on separate disk drives reduces the contention of read/write operations on big objects, for example, on a big table.
- Including several datafiles in a tablespace overcomes limitations of a single disk to accommodate a big object, e.g. a big table as it will be stored across several disks.

- Ease of management, availability
  - Storing data of different applications in different tablespaces prevents multiple applications from being affected if a tablespace must be taken offline.
  - Most DBMSs allow back up of individual tablespaces.

Data Blocks

At the finest level of granularity, a database stores data in data blocks (also called logical blocks or pages). One data block corresponds to a specific number of bytes of physical database space on disc. The rows of a table are physically stored in data blocks. When accessing the requested rows, the system reads the blocks in which these rows are stored. The fewer the number of blocks that are accessed during request processing, the better is the performance. This chapter discusses different approaches to storage organization that help in reducing the number of block accesses.

First, it is important to store rows of a table in the most effective way. Not all space in the data block is dedicated to data storage (Figure 2).

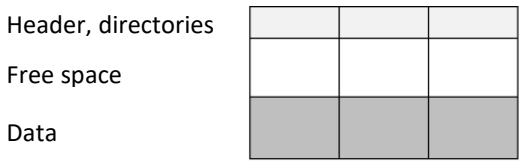

Figure 2. Data blocks.

Some space is occupied by the block header which contains information about the block that is used by the system, directories which contain information about the objects stored in the block and the addresses of the rows of the block. In addition, each block must contain free space to accommodate updates of existing rows that may cause the rows' expansion. Consider the situation in Figure 3 (block headers are not shown). The data block contains several rows and free space (Figure 3a). After the third row is updated, there is not enough free space in the block to fit the updated version of the row and the row has to be moved to another block. This moving of a row from one block to another is known as row migration as shown in Figure 3b. However, the system cannot change the initial address of the row when it was inserted (as it is already used in for example Indices). As a result, the information for the third row of our example is stored in two blocks – the row itself is now stored in the new block, and the address of this actual row (the 'forwarding address') is contained in the initial block used for the row (which will be accessed when the row is requested because the system remembers it as the storage block of the row). This situation is undesirable because in order to access the row, the system has to access two blocks instead of one. Numerous migrating rows may cause decreased performance.

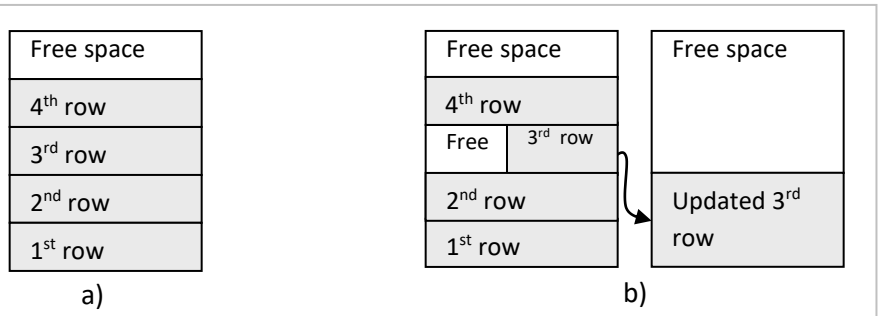

Figure 3. Migrating rows.

The DBMS allows specifying the percent of free space in the blocks of a table. Obviously, data blocks have to be as full as possible to store more rows, but, on the other hand, there must be enough space left for updates of the rows that are already in the block.

In addition to migrating rows we can have chained rows that are stored across several blocks. This situation can happen when for example the size of the row exceeds the size of the block. Chained rows are stored similarly to migrating rows: a block contains a part of the row with the forwarding address to the next block containing the continuation of the row.

Depending on the nature of the data and the database activities, database programmers have to find a compromise between partially full blocks and the risk of having migrating rows.

Appropriately chosen data types can improve data storage, e.g. choosing the `VARCHAR` data type for columns with variable length will result in more rows per block than when using the `CHAR` data type.

The size of the block also has an impact on performance and overall quality of data storage. Smaller block size is recommended for short rows but does not benefit long rows (as only few rows can be stored in a block with usually a high possibility of migrating rows). On the other hand, the overhead of the header is relatively high for small blocks. In Oracle, it is possible to have different block sizes for different tablespaces.

The table below summarizes some considerations when choosing the size of the block.

	Small blocks	**Large blocks**
Advantages	• Good for small rows with intensive random access as it reduces block contention (typical for Online Transaction Processing databases).	• Has lower overhead of the header; there is more room to store data. • Permits reading more rows into the buffer[3] with a single read/write (depending on row size and block size). • Good for very large rows.

[3] Buffer and its role in data processing are explained in the chapter on query processing.

Disadvantages	• Has relatively large space overhead due to the block header. • Not recommended for large rows. There might only be a few rows stored for each block; the change of migrating rows is significant.	• Wastes space in the buffer when accessing small rows. For example, with an 8 KB block size and 50 byte row size, 7,950 bytes loaded in the buffer are not used.

Smaller block sized (in Oracle, 2 KB or 4 KB) are usual for Online Transaction Processing (OLTP), while larger block sizes (in Oracle, 8 KB, 16 KB, or 32 KB) are usual for data warehouses.

Heap Storage

Depending on the type of storage, data blocks are filled by the records of a table differently.

Heap storage adds new rows to the available free space in the blocks that already contain the table's data. If there is not enough space in these blocks, then the system goes to the next unused block of the table's extent and places the row there[4]. As a result, rows are stored without any particular order. Figure 4 shows the process of inserting data into the table Employee of the Manufacturing Company case that uses the heap storage organization:

a) Rows for the first three employees are inserted into the first block. Then, none of the next four rows can fit in the rest of the available space in the first block, and these rows are placed in the second block (Figure 4a).
b) One row (row 6) is deleted from the second block, and hence a free space becomes available there (Figure 4b).
c) For the next two new rows, the first, for the employee with `ID` = 10, is short enough to fit in the first block, and the second, for the employee with `ID` = 11, is placed in the second block (Figure 4c).

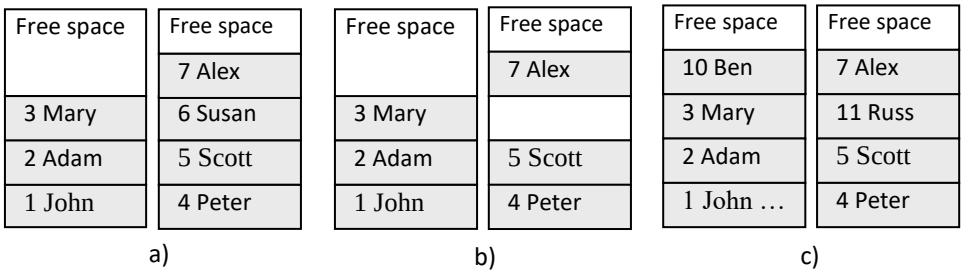

Figure 4. Heap storage of data.

For tables using the heap storage method, the search for rows by a specific condition results in accessing multiple blocks across which the requested rows are scattered. Let us consider the query:

[4] If there are no blocks in the extent, the system allocates a new extent in correspondence with the table growth specifications.

```
SELECT e.emplName, d.deptName
FROM Employee e INNER JOIN Department d
     ON e.deptCode = d.deptCode
WHERE deptCode = '002';
```

For the Employee table using heap data storage as shown in Figure 5a, the search for the employees of the second department (`deptCode = '002'`) requires the system to access all data blocks of the table Employee (two in this case) and one block of the table Department (the requested rows are shown in bold italics).

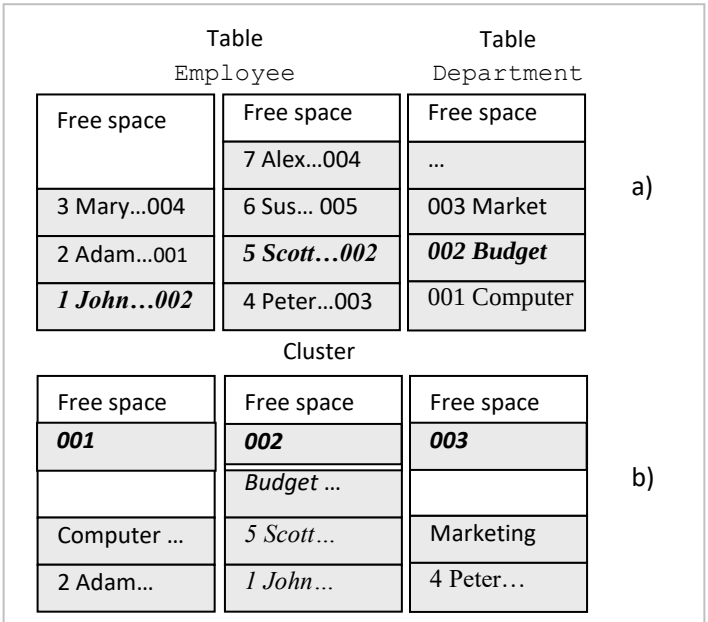

Figure 5. Heap and clustered organization of data.

Cluster Storage

If database applications frequently access data about employees of a particular department, then their performance can be improved by keeping the rows of employees of the same department together in the same block or blocks. In this case, as Figure 5 b shows, the system needs to access fewer blocks. For join queries, as the one we are discussing, performance can be further improved by storing the row of the respective department in the same group of blocks. Such an organization of storage in which rows that are similar by some criterion are stored in the same block is called *clustering*, and the group of blocks with similar rows is referred to as a cluster.

A column (or columns) that define how rows are clustered is called the *cluster key*. In our example, the cluster key is the column `deptCode` – rows of the tables Employee and Department that have the same `deptCode` value are stored together, and the value of the cluster key is stored only once.

Unlike heap storage, where the location of a row is random, the locations of rows in clustered storage are predefined. When a new row is inserted, it is not placed in the first block with available space, but in the block

designated for storing rows with the same cluster key value as the inserted row[5]. Note that a table when stored as clustered usually occupies more blocks than when it is stored as heap. In our example, assume that all records of employees of the five departments are stored in three blocks of the heap, however, the clustered by `deptCode` storage will occupy five blocks according to the number of departments.

Access to data includes locating the data and reading the data into memory. Clustering can provide a more efficient means to read the requested data. But before continuing this discussion we need to introduce a new database object – index.

Indices

As we have just seen, rows in tables with heap storage organization are stored without any particular order and access to a specific portion of data in the table in many cases requires access to most of the table's data (this is called a full table scan). For example, to find the employee with `ID` = 1, the system has to perform the reading of all the data in the table Employee. Indices are special database objects that make access to data more efficient.

Role of Indices

The most common are B-tree (balanced tree – a generalization of a binary search tree) indices. The description of their organization is beyond the goals of this book and can be found in [Connolly 2010], [Silberschatz]. The role of the index is similar to the role of the last and first names of the telephone directory in finding the address or telephone of a person. Telephone directory searches are simplified by ordered last and first names that allow one to apply some reasonable search strategies, e.g. if the last name starts with 'B', then we do not need to look in the end of the directory and can concentrate on its beginning. Indices are built for the columns of the table by which data are searched, like ID in the example above. Values of indexed columns are stored in an ordered way (similarly to the last and first names in the phone directory) in the index, and for each set of values the index contains pointers to the corresponding rows (blocks) of the table. For example, if users access data about employees by `ID`, it would make sense to create the index on this column:

```
CREATE INDEX i0 ON Employee(ID);
```

The index contains values of the column `ID` and pointers (addresses) to the corresponding rows of the table Employee. With the index, a request for data about a particular employee will result in:

1) Searching the index and finding the requested values of ID
2) Reading pointers to the corresponding rows of the table
3) Accessing the blocks containing these rows of the table Employee. Figure 6 schematically[6] shows such access.

[5] If the block is full, then the system adds a new block to the cluster and places the row in this block. The cluster becomes chained.

[6] B-tree indices are self-balanced trees.

The index on the column ID of the table Employee was created to demonstrate indexed access.

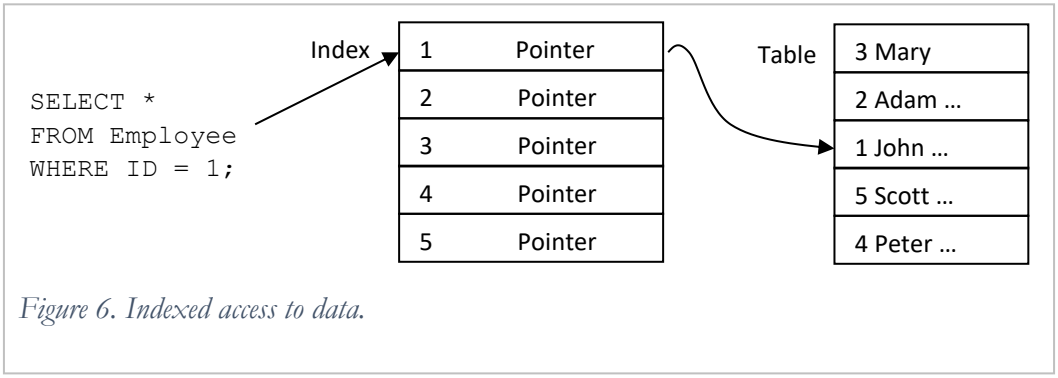

Figure 6. Indexed access to data.

However in reality the index on the column ID would have been created automatically by the DBMS because the DBMS creates an index for the primary key of every table. Such an index is called the primary index. Indices on the non-key columns of the table (or on some of the key attributes, or on key attributes in the order different from the order of these attributes in the primary key) are called secondary, and they are created to improve the performance of queries that access data based on conditions on these columns. For example, for queries that request data about employees given their name it is reasonable to consider the index on the column `emplName`. In many cases, indices can dramatically improve database performance.

The storage parameters for indices are defined similarly to the storage parameters of tables. It is recommended to store tables and their indices in different tablespaces with the files located on different disks. In this way, the performance is improved because the system can access a table and its index in parallel.

Some DBMSs support other types of indices, e.g. MS SQL Server offers hash indices and Oracle – bitmap and function-based indices. The purpose of different types of indices is to improve the performance of read, delete and update queries (whereas insert queries are actually slowed by them). The discussion of indices and the conditions when they are beneficial is continued in Chapter 5 on query processing and performance.

Cluster Storage and the Use of Indices

The clustered storage organization that we illustrated above can be supported in two ways. The first type of clustering which is called *index clustering* requires a cluster index. Queries for clustered rows are accessed through the cluster index: first, the system reads the address of the table block for the requested cluster key value in the cluster index, and then it goes to the block with the rows. Cluster indices are smaller than table indices. The cluster index contains as many entries as there are values of the cluster key, while the table index contains as many entries as there are rows with values of the index column. For example, the index for the cluster built on the column `deptCode` of our discussed example has 200 entries – as many as there are departments, while the index on the column `deptCode` of the table Employee has 4000 entries – as many as there are rows in the table.

Many DBMSs also support another type of cluster known as *hash cluster*. Rows in the hash cluster are stored based not on values of the cluster key itself, but based on the result of applying the hash function to the cluster key. The hash function applied to the cluster key defines the location of a new row when data is inserted, and

the location of an existing row, when data is retrieved. Appropriately defined hash clusters perform the access to a row in a single disk read.

The hash and index clustering on the same cluster key result in similar storage. However, access to hash clusters is different from access to index clusters (see Figure 7). The hash cluster does not need an index to support it. When a row with a particular value of the cluster key is requested, the system applies the hash function to *calculate* the address of the row. In the index cluster, the address of the table row is *found* in the cluster index search.

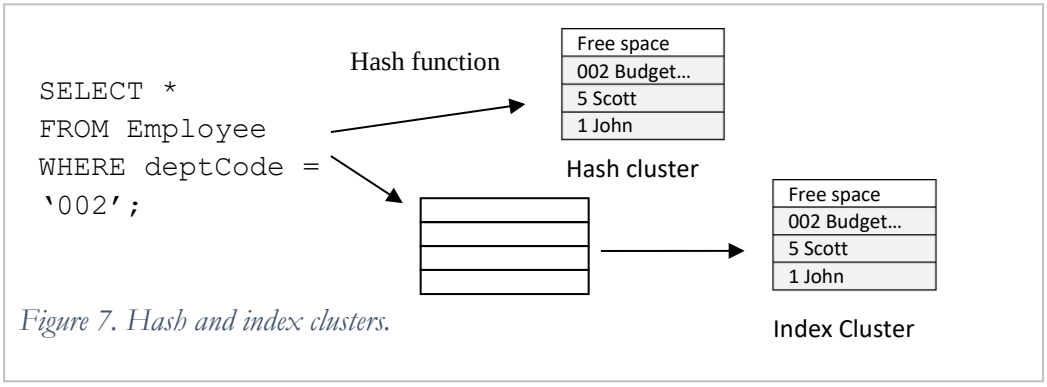

Figure 7. Hash and index clusters.

In addition to heap and cluster types of storage, most DBMSs offer other storage solutions, e.g. partitions and index-organized tables that are discussed later in this chapter. Each storage solution is aimed at reducing the number of disk accesses during query processing. Different solutions may be beneficial for different types of access to data. For example, the cluster solution works well for the select query discussed above, but it causes problems when the cluster key is frequently modified. Chapter 5 discusses database performance and gives recommendations about using different types of storage depending on types of data access.

DBMSs have other parameters that define how data are stored and processed, e.g. compression and encryption parameters. For each DBMS these parameters have to be considered for the physical data model because they have impact on database performance and ease of maintenance.

Transparency of the Physical Model

The physical data model is transparent to users – they do not know where the data are physically stored or how data storage is organized. Transparency is supported by the architecture of the DBMS.

3-level Database Architecture

Commercial DBMSs comply with the three-level architecture of the database suggested by the American National Standards Institute (ANSI) and Standards Planning and Requirements Committee (SPARC) in 1975. This architecture (Figure 8) defines three levels of abstraction of the data or three levels of data description in the database; it provides a separation of higher-level semantics of data from the physical implementation and makes the physical implementation transparent to users of the database.

The *external schema* defines how users see the data (users' views of data). For example, Scott and John are users of the discussed Manufacturing Company Case database: Scott works with ID, name, and salary of employees – this is his view of the database content; John works with employees' names, and codes and names of employees' departments – this is how he sees and understands data from the database. The external schema of

the database consists of a number of views for different users. DBMSs have features that allow for implementing the external level. One of the database objects used for this purpose is a *view*. A view is a saved query. The following view for Scott represents Scott's needs in data:

```
CREATE VIEW vw_Scott AS
SELECT ID, name, salary
FROM Employee e, Title t
WHERE e.titleCode = t.titleCode;
```

When working with the database, Scott will be using the view, e.g.:

```
SELECT * FROM vw_Scott;
```

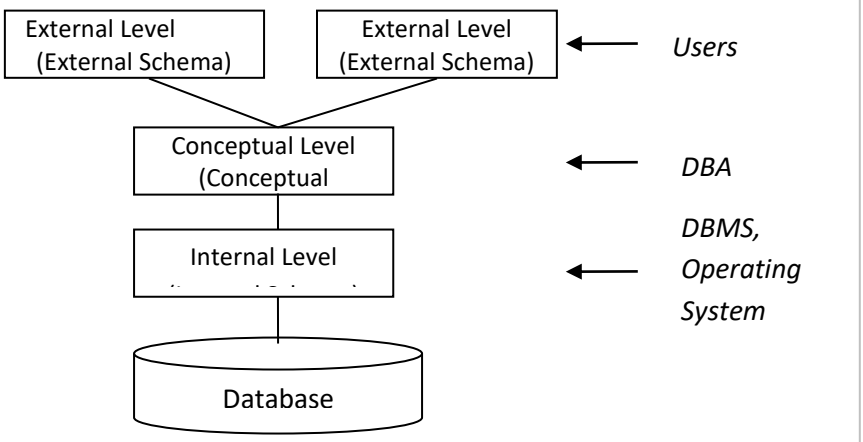

Figure 8. Three-level database architecture.

The view makes the database transparent to Scott – not only does Scott not know where the data are located and how data storage is organized, he does not even know what tables he is using.

The *internal schema* defines how the DBMS and operating system see the data of the database; it includes descriptions of data structures and file organizations, storage space allocations for objects of the database, rows placement, data compression and data encryption, and other physical parameters of data.

The *conceptual schema* serves as a link between the external and internal schemas; it provides a mapping from the internal schema to the external schema. This level supports the external level – each external view is based on definitions of objects in the conceptual level. The conceptual schema reflects the developer's vision of the database; it contains the definitions of tables, columns, and constraints.

The 3-tier architecture provides independence between the users' perception of the database and the physical implementation of the data. Such independence has the following benefits:

- Users see the data according to their need: If the users need changes, their view of data can be changed without rebuilding the database and without affecting the views of other users. For example, if Scott's needs changes and he wants to see ID, name, title, and salary, then anew view can be built without any changes to the database or to the existing view for John.

- Users see data the way they want to see it, completely unaware of the conceptual and physical aspects of the data: The user is unware of which tables the data comes from, where data are stored andin what data structures or formats, and what other structures were involved in processing the data (like indices or clusters). For example, Scott does not know that he is working with the tables Employee and Title and what structure the tables have, or that the DBMS uses the primary index on ID to retrieve data about a particular employee.
- Reorganizing data storage does not affect the conceptual model of the database and users' views: For example, the table Employee can be moved from one disk to another without the users knowing about the move, and data processing applications referencing this table will remain valid and unchanged.
- Redesigning the conceptual model does not necessarily affect the users' views: If the database programmer adds a column to the table Employee, none of the previous views based on this table need to be redefined or changed since they would not be using the new column. Some changes on the conceptual level may cause rebuilding user views, but the users still will be unaware of these changes. For example, if the table Title starts storing salaries in thousands of dollars (and not in dollars as before), then Scott's view will be changed:

```
CREATE VIEW vw_Scott AS
SELECT ID, name, salary * 1000 as salary
FROM Employee e, Title t
WHERE e.titleCode = t.titleCode;
```

  Scott, however, will not notice the change. Of course, more "radical" changes of conceptual schema may cause users to change their perception of the data, like, for example, deleting a column from a table.

The Physical Data Model in Oracle

Heap Storage Organization

Table storage parameters are defined in the `CREATE TABLE` statement:

```
CREATE TABLE Title (
   titleCode CHAR(2) PRIMARY KEY,
   titleDescription VARCHAR2(15),
   salary NUMBER CHECK (Salary BETWEEN 30000 AND 90000))
PCTFREE 10
PCTUSED 40
TABLESPACE users
STORAGE (  INITIAL 50K
           NEXT 50K
           MAXEXTENTS 10
           PCTINCREASE 25 );
```

The *TABLESPACE* clause defines the tablespace where the table will be located (see the explanation of tablespaces above in this chapter and in Appendix 3). The *STORAGE* parameters specify the initial size and expansion of the table. In the above example, the initial size of the table is set to 50K. If expansion of the table is needed, the system allocates not more than 10 additional extents – the second extent will be equal to 50K, and every next extent will have the size of the previous extent increased by 25 percent. Such an approach for defining and capturing storage for a table is flexible and dynamic – the storage space for the table is allocated when needed.

The parameters *PCTFREE* and *PCTUSED* define how allocated data blocks are filled with data[7]. PCTFREE sets the percent of space in the block that has to remain free to accommodate future updates of the rows of the block. Once the block no longer has its free space percentage greater than or equal to PCTFREE, it is removed from the list of available blocks the DBMS keeps. For example, if PCTFREE is set to 20, it would mean that the block allows for row inserts until 80% is occupied, leaving 20% free for updates to existing rows in the block.

PCTUSED defines what percent of space in the block has to become free again (the percentage of the block used must be less than PCTUSED) for the system to return the block to the list of available blocks that can add new rows. For example, if PCTUSED is set to 40, it would mean that no new rows can be inserted until the amount of used space falls below 40%.

The appropriate use of these parameters can increase the performance of writing and reading operations on tables or indices, decrease the amount of unused space in the data blocks, and decrease the amount of migrating rows between data blocks. Chained rows – rows stored across several blocks – can occur not only when the length of the row exceeds the size of the block, but for the tables with more than 255 columns.

When the system has to insert a new row in a table, it looks into the list of available (free) blocks for this table, i.e. the list of blocks that allow adding a row, or in other words, the blocks for which the free storage percentage is above PCTFREE. After the block becomes full, it is deleted from the list of free blocks and can be returned to the list only after the percent of its used space drops below PCTUSED. A lower PCTUSED increases the unused space in the database, but reduces the processing cost of UPDATE and DELETE because the block will be moved out of the list of the free blocks less often. A higher PCTUSED improves space efficiency, but increases the cost of INSERT and UPDATE, because the block is eliminated from the list of free blocks and is returned back into the list frequently.

A lower PCTFREE leads to less unused space in the table blocks (more records in the block) and fewer blocks to accommodate the table. However, when this parameter is small, the possibility of migrated or chained rows is increased. A high PCTFREE setting, on the other hand, may result in sparse table storage but a lower possibility of chained rows.

The settings of PCTUSED and PCTFREE are dependent on the type of the activities expected on the table. For example, for a table that experiences frequent updates which may increase the size of the rows PCTFREE = 20 and PCTUSED = 40 is appropriate. On the other hand, for a very large table, for which storage efficiency is important and which experiences little insert, delete, and update activities, an appropriate setting is PCTFREE = 5 and PCTUSED = 90.

Clustered Storage Organization

The table Title in the previous section was stored as a heap. For some applications, it is beneficial to store a table in a cluster. Oracle supports two types of clusters: index and hash.

---

[7] The new feature – Automatic Segment Space Management (ASS Management) – automates management of some aspects of the physical model and does not allow to specify PCTFREE, PCTUSED, and some other parameters that are used for manual storage management.

It is important to remember that clusters improve performance of some read queries, but intensive updating activity may result in the necessity to reorganize clusters and therefore can worsen performance.

To improve the performance of queries that are based on the join of the tables Employee and Department, we can consider storing these tables in an index cluster with the deptCode column as the cluster key. The cluster is created with the help of the following command:

```
CREATE CLUSTER Emp_dept (deptCode CHAR(3))
TABLESPACE USER_DATA
PCTUSED 80
PCTFREE 5
SIZE 40
STORAGE (…);
```

The command specifies the type of the cluster key, the tablespace in which the cluster is located, data packing parameters of the cluster blocks and the storage settings. The parameter SIZE defines the expected length (in bytes) of one row in the cluster and, therefore, how many rows can be placed in one cluster block. If the parameter is too high, then fewer records are placed in one block and the cluster occupies more space than it actually needs. On the other hand, if the size is set too low, chaining of data may occur.

The cluster can be used for allocating the tables Employee and Department. Note that the cluster storage is already organized and you do not have to specify the tablespace and packing parameters for the tables – data are stored according to the corresponding parameters of the cluster:

```
CREATE TABLE Department (
      deptCode CHAR(3) PRIMARY KEY,
      deptName VARCHAR2(15),
      location NUMBER)
CLUSTER Emp_dept (deptCode);

CREATE TABLE Employee (
      ID NUMBER PRIMARY KEY,
      emplName VARCHAR2(20) NOT NULL,
      emplType VARCHAR2(10),
      deptCode CHAR(3) REFERENCES Department,
      titleCode CHAR(2) REFERENCES Title)
 CLUSTER Emp_dept (deptCode);
```

The index cluster needs a cluster index. The index has to be created before inserting the first record into any of the two cluster tables:

```
CREATE INDEX Emp_dept_index
    ON CLUSTER Emp_dept;
```

Hash clusters are created similarly to index clusters; however, they do not need the cluster index. The hash function, applied to the cluster key, defines the address of a row. This is really a "direct" and very efficient way to access data. DBMS documentation contains recommendations on using hash functions.

The following hash cluster with the hash function on the column deptCode is created for the table Employee:

```
CREATE CLUSTER Employee_cluster (
```

```
deptCode CHAR(3))
PCTUSED      80
PCTFREE      5
SIZE         40
HASHKEYS     10
STORAGE (…);
```

The important parameters of the hash cluster are the expected number of values of the cluster key column (HASHKEYS) and the average row size (SIZE). The above values for these parameters are based on the company having 10 departments and an average row size of 40 bytes. These parameters are used by the DBMS to specify and limit the number of unique hash values that can be generated by the hash function used by the cluster and the number of rows per cluster block. The prepared cluster is used to store the table Employee:

```
CREATE TABLE Employee (
      ID NUMBER (5,0) PRIMARY KEY,
      emplName VARCHAR2(20) NOT NULL,
      emplType VARCHAR2(10),
      deptCode CHAR(3) REFERENCES Department,
      titleCode CHAR(2) REFERENCES Title)
)
CLUSTER Employee_cluster (deptCode);
```

In this example, the system hash function is used. It is possible to use user-defined hash functions as well. For every new row of the table Employee, the system calculates the hash value based on the deptCode (the value defines the address of the block for the row) and as a result, the rows with the same value of deptCode are stored together. For the retrieval of data about employees of a particular department, the same hash function is applied to the value of deptCode in queries with the condition:

```
…WHERE deptCode = x
```

The value of the hash function defines the block address of the rows of the query and enables the system to go directly to the blocks with requested data.

Index-Organized Tables

Oracle supports another organization of data storage – index-organized tables. In such tables, all table data, both the primary key and all non-key columns are stored in the index in key-sequenced order. In a heap organized table, the primary key index stores the primary key with the address of the corresponding row (in Oracle the address of the row is called the ROWID) in the table being indexed. For a key-based query, two accesses are required. First, the index must be accessed to find the address of the row (ROWID), and then a second access is made on the table. For index-organized tables only one access is needed, because the index contains the entire data in each row: primary key and non-key data.

The following statement creates the index-organized table Employee:

```
CREATE TABLE Employee (
      ID NUMBER (5,0) PRIMARY KEY,
      emplName VARCHAR2(20) NOT NULL,
      emplType VARCHAR2(10),
      deptCode CHAR(3) REFERENCES Department,
      titleCode CHAR(2) REFERENCES Title)
      ORGANIZATION INDEX;
```

Index-organized tables have several advantages:

- Fast access to data by queries with equality or range conditions on the primary key of the table (or any left-most part of the primary key): In the case of indexed access to a regular table, there are at least two block accesses: one to the index and another to the table, while for the index-organized table there is only one access to the table itself.
- Efficient storage: Because the indexed columns are not duplicated as in the "table plus index" case, and ROWID (address of the row) is not stored, the index-organized table occupies less space than the heap table plus its primary index.

Index-organized tables are ideal for those applications, which require fast primary key access.

If the index-organized table is accessed by queries with conditions on the non-key columns, it loses its performance advantage. Secondary indices on such storage organization are less efficient than they are on the heap table.

Partitions

Oracle 8 introduced a new storage feature – partitioning. Partitioning addresses key issues in supporting very large tables and indexes by letting you decompose them into smaller and more manageable pieces called *partitions***.** Imagine that the database for the Manufacturing Company case is implemented as a centralized database. Local users of Boston have to perform access to the tables where the rows of Boston departments and employees are scattered across multiple blocks that contain data about the departments and employees of other locations as well (unless data is clustered by location). Accessing Boston data requires reading multiple blocks and the efficiency of such access is low.

Partitioning combines the benefits of the logical integrity of the centralized database and the physical independence of data in the distributed database (see discussion of distributed databases below in this chapter and in Chapter 3). Partitioning unambiguously assigns each row to assigned particular partition based on the partition key. The partition key is a set of one or more columns that determines the partition for each row. The following statement creates the table Department partitioned by location, which is the partition key:

```
CREATE TABLE Department (
…
)
PARTITION BY HASH(location)
PARTITIONS 3;
```

This is an example of hash partitioning: for each of the three locations, the system hash function returns a particular value, and the rows with the same hash value are stored in the same physical partition. Figure 9 shows the difference in access to data for partitioned and non-partitioned tables.

Oracle also supports range and hybrid (combination of hash and range) partitioning. Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition as shown in the following example, which creates a table Title with two range partitions: one for the titles with low salary (less than 50000), and another for the titles with high salary (between 50000 and 100000). Note that partitions in this case have different physical characteristics (to accommodate differences in managing titles with low and high salaries) :

```
CREATE TABLE Title (
```

```
…
)
PARTITION BY RANGE(salary) (
PARTITION low_Salary VALUES LESS THAN (50000)
 TABLESPACE TS1
 STORAGE (INITIAL 5M, NEXT 1M, PCTUSED 75, PCTFREE 15)
PARTITION high_Salary VALUES LESS THAN (100000)
 TABLESPACE TS2
 STORAGE (INITIAL 2M, NEXT 20K, PCTUSED 80, PCTFREE 10));
```

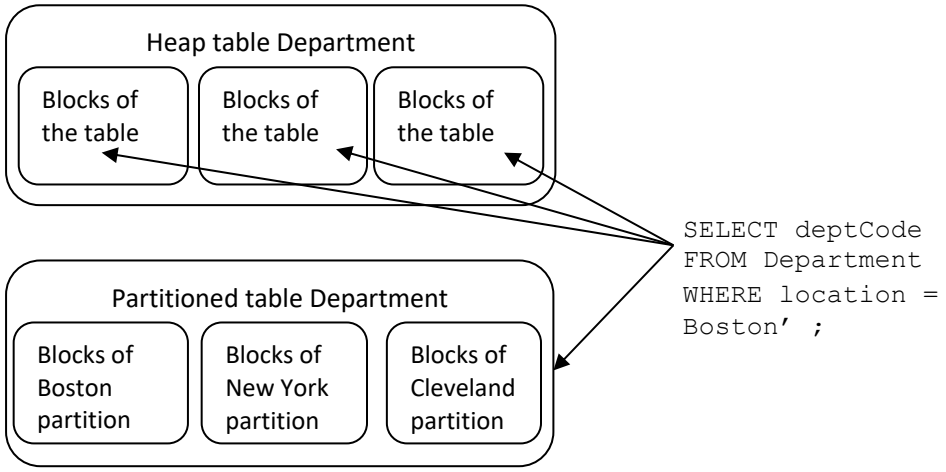

Figure 9. Not partitioned and partitioned table.

Storage parameters of each partition – location, size, and block packing – are specified similarly to the storage parameters of a separate table. Access to data on the partitioned table can be supported by global table indices. Additionally, a partition can have separate partition indices. For example, the partitioned by location table Department can have the global index on the deptType, and the Boston partition can have the index on the deptName.

Note the differences between clustering and partitioning:

- All blocks of a clustered table are organized in the same way according to the storage settings of the cluster. Blocks of different partitions can be organized differently because storage of each partition is defined separately.
- A table is clustered if for each value of the cluster key we expect not too many records that will be stored in one or several blocks. Partitions are beneficial for storing large numbers of records. The examples above were used for demonstration purpose only – obviously, partitioning the tables Department and Title does not seem appropriate. A better example could be partitioning the table Employee by emplType if the table contains hundreds of thousands of records.

Changing the Physical Model

New user requirements and or changes in data usage can require changes of the physical model. Often the physical model needs to be changed when we discover that our assessment of some physical parameters was not correct, e.g. we have a table with many migrating rows, or our partitions cannot accommodate new data. Some changes require rebuilding the table, some other changes can be handled by the table alteration.

For example, we discovered that partitioning our table Department can be beneficial for our queries and hence we create a new table called Department_new with the new partitioning requirements:

```
CREATE TABLE Department_new (
      …
      )
      PARTITION BY HASH(location)
      PARTITIONS 3
      AS SELECT * FROM Department;


INSERT INTO Department_new
(SELECT * FROM Department);
```

After the new table is created, we can drop the original table and rename this new table to the original table name. Note that constraints may require additional attention.

The following example illustrates alteration of the table when we need to create a new partition either on the high end of the partitioning range

```
ALTER TABLE Title
ADD PARTITION very_high_Salary VALUES LESS THAN 150000;
```

or in the middle of it

```
ALTER TABLE Title
SPLIT PARTITION high_Salary AT (75000)
INTO (PARTITION modest_Salary,
       PARTITION high_Salary);
```

Transparency of the Database

Various physical aspects of the data in Oracle databases, such as the actual organization of the data (heap, cluster, index, index-organized or partitioned storage of data), the percent of free space in data blocks, the existence of indices are transparent to database users and in many cases to database programmers.

Database programmers work with the conceptual schema of a table, which is available from the data dictionary. For example, the following command shows the structure of the table Employee:

```
SQL> DESCRIBE Employee;

 Name             Null?                  Type
--------           ----------------           ----------------------
ID                      NOT NULL                    NUMBER
EMPLNAME                NOT NULL                    VARCHAR2(30)
EMPLTYPE                NOT NULL                    VARCHAR2(10)
. . .
```

When inserting a new row into the table Employee, the programmer uses this information without knowing the details of how the row is physically written into the database:

```
INSERT INTO Employee(ID, emplName, emplType, …)
```

```
VALUES (1234, 'John', 'Full-time', …);
```

If physical parameters of the table are changed or the table is moved to another tablespace, this query will remain unchanged because it is independent of the physical schema of the table Employee.

The Oracle DBMS, on the other hand, uses the internal schema to process data from the Employee table. When inserting a row, the DBMS must find the block for the row placement (according to parameters PCTFREE and PCTUSED, and the type of storage), check whether the tablespace has to be extended if all allocated data blocks are full, represent the data changes in all indices on the table, and perform some other actions. All this processing is hidden from the users and programmers.

Distributed Data Storage

A *distributed database system* allows applications to access data from local and remote databases. The distributed implementation of a database can significantly improve its *performance*, *reliability* and *availability*. In a distributed database, data are physically stored in several databases. This allows for better performance of some database applications and makes data accessed easier by local sites. Replicating data on different sites of the distributed database improves the reliability of the whole database and improves accessibility of data by local users.

The discussion of distributed databases is limited to the solution supported by most commercial DBMSs. A collection of multiple *logically interrelated* and *physically independent* databases is considered a distributed database if there exists at least one application that uses data from these different databases; such an application is called *global*.

According to this definition, the distributed database is composed of separate autonomous databases that are supported independently. In case all databases in the distributed database are implemented in the same DBMS, such a database is called homogenous. Discussion of other distributed solutions with databases physically dependent on each other (tightly integrated or semi-autonomous) or implemented in different DBMSs (heterogeneous) is beyond the scope of this book and can be found in [Őzsu].

There are several instances where distributed databases are beneficial:

- Localizing access to specific portions of data
- Improving reliability and availability of data
- Improving performance: data are localized for the greatest demand, and access to multiple database systems is parallelized.

Distributed solutions are justified for large databases because of their complexity and cost.

A distributed DBMS is a software system that enables management of a distributed database.

Promises of Distributed Databases

The possible advantages of distributed databases are discussed using the example of the Manufacturing Company case. Because the offices in the three different cities deal with local data – data about local departments and employees working in these departments – it may be reasonable to distribute the data so that a database used by a particular office stored only the local data. Such a solution can have several benefits.

Improved Performance

The mentioned distribution of data means that each database contains parts of the tables Department and Employee with data about local departments and employees working in these departments. Parts of tables residing in different databases are called fragments, and the process of splitting a table is called fragmentation.

Such data localization can improve the performance of local applications in the cities (Figure 10) because:

- Each local application handles smaller amounts of data
- If data are brought closer to local users, access to the data is faster
- The management of a smaller database is easier.

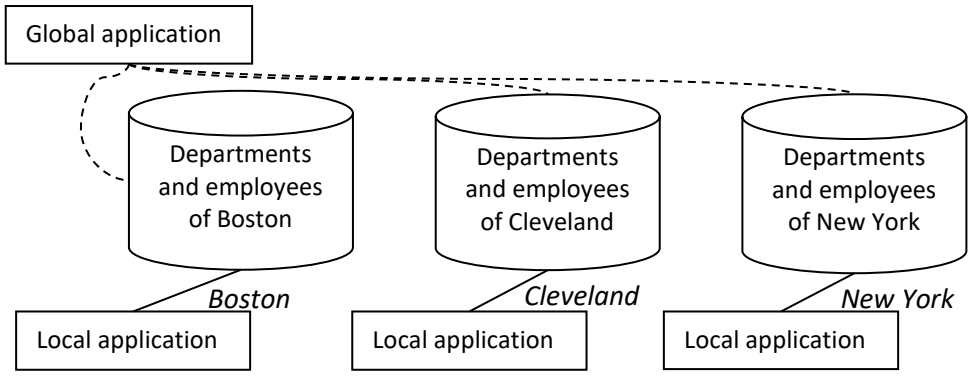

Figure 10. Localization of data.

The performance of some global applications may be improved as well. For example, if a global application requests data about all the employees of the company with a particular title code, then the global request is distributed to three databases, and data processing is performed in parallel (such execution is called intra-query parallelism). The parallel processing of smaller amounts of data is faster than the processing of the whole amount of data. Suppose there are 3000 employees evenly distributed across the databases of the three cities (1000 employee rows in each local database). Then the retrieval of data about employees of a particular title is performed as a search on 1000 rows in each of the three databases in parallel, and data could be retrieved faster than using a search on 3000 rows in the centralized database. However, the performance of distributed queries also depends on data transfer between databases; the cost of data transfer may be significant and can compromise the performance of some distributed queries.

In addition to intra-query parallelism, the distributed database can provide greater inter-query parallelism, when more queries can be executed in parallel increasing the throughput of the database – three databases in our case can process more requests simultaneously than the centralized database.

The distribution and localization of data may also improve the performance of other applications in the company because the load on the company's network will be reduced.

Scalability

In a distributed solution, an expansion of a database or an increase in the number of database users is easier to handle than in a centralized one – new servers can be added to the distributed database without affecting the

performance of existing databases. In the centralized solution, on the other hand, to accommodate growth of the database or an increase in database load, it is necessary to upgrade hardware and software, which may make the database temporarily unavailable and require modification of the database or database applications[8].

Increased Reliability and Availability

Using several databases allows for replicating data and increasing data reliability. Figure 11 shows that if the Boston database goes down or even is destroyed, the application can switch to the Cleveland database that keeps a synchronized copy of the Boston data (replica).

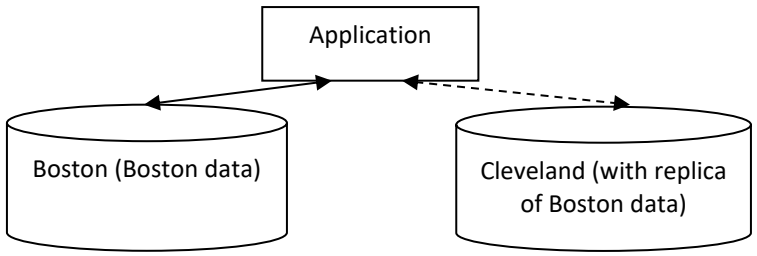

Figure 11. Replication of data.

Distribution also improves data availability. Imagine that the database administrator has to perform some reorganization of the Cleveland data. In a centralized database, this activity may result in the temporary unavailability of the tables Department and Employee. If data are localized (data for Boston's departments and employees are kept in the Boston database), then the availability of this data to local applications is not dependent on the condition and availability of data and database servers in other locations.

Complicating Factors

A distributed solution, though it might bring significant improvements in the performance and reliability of the database, is more complicated and costly. The following are the complexities of the distributed approach:

- Lack of standards and methodology:
  - ✓ As with physical design in general, there is no straightforward methodology of data distribution design, often it is a 'trial and error' approach.
  - ✓ There are no standards or methodology on converting a centralized database into a distributed one.
- Complex design of data distribution. Decisions on splitting tables into fragments, allocating and, possibly, replicating fragments and unfragmented tables are often difficult to make.
- Complex implementation, including implementation of integrity constraints and transparency of data distribution.

Complexity of Design

Decisions on the distribution of data often are difficult to make because of conflicting needs in data. For instance, in our case, in addition to the mentioned requirements about offices in three cities that use data about local departments and employees, we learned about users in Boston who process data about full-time

[8] In some cases, a cluster of database servers and a DBMS supporting the cluster architecture can be another approach.

employees, users in New York who perform similar processing of data about part-time employees, and users in Cleveland who work with data about consultants. In each city, the application, which supports data about the employees of a particular type, will have to access three different databases on the fragmentation shown in Figure 10. The availability of data for this application will be compromised because it will depend on the availability of three databases. The performance of this application may become worse because it will have to access two remote sites to perform its tasks (Figure 12).

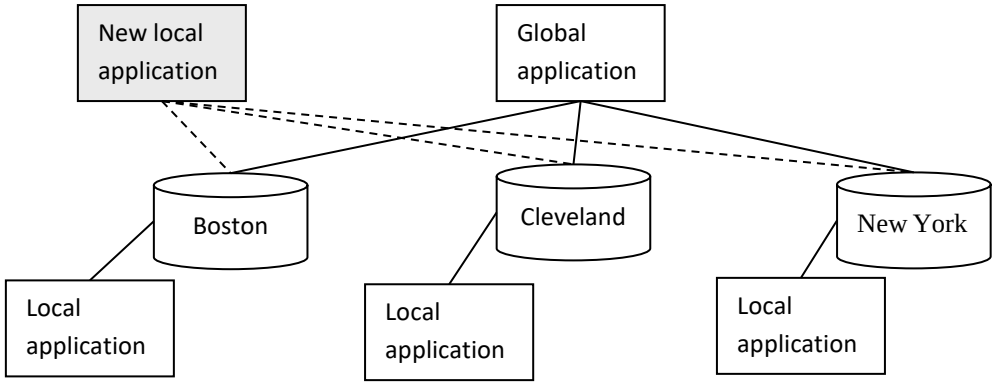

Figure 12. Local applications with conflicting requirements.

Localizing data by employee type for the new application, on the other hand, will make the existing local applications more complicated and, most probably, slower. The decision about the distribution of data about departments and employees depends on the importance of the local needs for data. Usually, a decision on the importance of an application is based on how frequently it accesses the data. If, for example, the application that supports the data about local departments and employees is executed more frequently than the new application, then the initial distributed solution will improve the performance of the more important application and be beneficial, even though it may make the performance of the other application worse.

The decision on the allocation and replication of fragments and tables often involves balancing between intentions to increase reliability and availability of data on the one hand, and the necessity to improve performance and lower the cost of data support on the other.

Complexity of Implementation

For distributed solutions, the implementation of some aspects of relational databases requires additional efforts:

- Most DBMSs do not provide transparency of distribution and replication of data. These transparencies, therefore, must be implemented by the database programmer (see the section on transparency of distribution below).
- Most distributed DBMSs do not support distributed integrity constraints. Usually, these constraints have to be implemented through database triggers or in applications.
- Resolving performance problems and tuning the distributed database are more complicated than working with a centralized database; they involve considering such additional issues as the distribution of data, communication costs, and obtaining sufficient locally available performance information.
- Additional performance problems may be caused by data replication.

- There are certain issues in implementing security in distributed and replicated databases; these issues are discussed in Chapter 4.

Some other problems, like distributed concurrency control and consistency of replicated data, recovery of replicas, switching from failed databases to functioning ones, distributed deadlock management are usually resolved by the DBMS.

The Distributed Database in Oracle

Oracle provides support of distributed databases. Consider a distributed solution for the Manufacturing Company case. Figure 13 shows part of this distributed database – the New York and Boston databases with localized data about the company's departments (the table Department in New York contains the rows of New York departments and the table Department in Boston – the rows of Boston departments). The case description mentions that the application that is executed from the New York office requires data about all departments. The application should be able to access the Boston's portion of the data about the departments. Obviously, the databases have to be connected physically via the network. Oracle makes a database "visible" to another database through a database link. A database link defines a one-way communication path from one Oracle database to another; it is a logical connection between the databases (made possible by the physical network connection between them) that specifies the name and location of the remote database.

In this case, the New York database has to "see" the Boston database and, therefore, has to contain the following database link (as in the Figure 13):

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

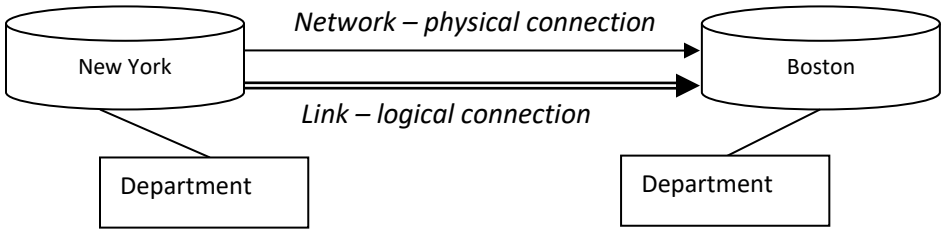

Figure 13. Distributed database in Oracle.

Users of the New York database can access the Boston table Department by explicitly specifying the database where the table is located:

```
SELECT * FROM Department@boston;
```

Note that the Boston database does not "see" the New York database through this link.

Transparency of Distribution

The physical distributed design of the database is not transparent to the users as the users must not only know that the database is distributed but also know the locations of the tables needed. If the table Department is relocated from Boston to another database (e.g. Cleveland), the way users access data will be affected and some applications will have to be rewritten, e.g. the last query of the previous section.

The goal of transparency of distribution is to make the distributed database appear as though it is a single Oracle database. The Oracle distributed database has features that hide (or allow hiding) the physical location of database objects from applications and users. Location transparency allows users to address objects such as tables in a unified way regardless of the database where the objects are located.

Location transparency has several benefits, including:

- Access to remote data is simple because database users do not need to know the physical location of the remote database object.
- Administrators can move database objects with no impact on users' requests or existing database applications.

An object's location can be made transparent with the help of *synonyms*. Synonyms are additional names for a database object. For example, we will create the synonym for the remote table Department of the Boston database in the New York database:

```
CREATE PUBLIC SYNONYM Department_Boston FOR Department@boston;
```

Now users of the New York database can access the remote table Department with the help of a simpler query that does not depend on the location of this table:

```
SELECT * FROM Department_Boston;
```

The users' view of the distributed database is simpler now (as shown in Figure 14) and they see it as the centralized database.

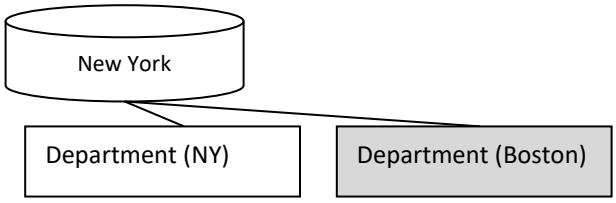

Figure 14. Transparency of distribution.

However, the database is still not transparent to users (or applications) that need to access data about all departments because they need to know that data about departments are fragmented and correspondingly build their requests:

```
SELECT * FROM Department
UNION
SELECT * FROM Department_Boston;
```

The fragmentation of data can be made transparent with the help of *views*. For example, for the global New York application we will build the following view:

```
CREATE VIEW Department_all AS
SELECT * FROM Department
UNION
```

```
SELECT * FROM Department_Boston;
```

Now data about all departments can be accessed through the view:

```
SELECT * FROM Department_all;
```

In many database applications, different transparencies are implemented with the help of *stored procedures*. The stored procedure is a database object implemented in the procedural extension of SQL (PL/SQL in Oracle). Users execute the procedure without knowing the details of how data are processed by the commands within the procedure. For example, the following procedure created in the New York database performs the inserting of a new row in the table Department of the Boston database:

```
CREATE PROCEDURE insert_Department
            (par_Code CHAR, par_Name VARCHAR2, par_Location VARCHAR2,
            par_Type VARCHAR2) AS
BEGIN
      INSERT INTO Department@boston
      VALUES (par_Code, par_Name, par_Location, par_Type);
END;
```

Adding a new Boston department can be performed by executing the procedure with the appropriate parameters' values:

```
EXEC insert_Department ('999', 'Accounts Payable', 'Cleveland',
'Business');
```

Stored procedures implement the transparency of data and data processing. Consider the case where we need to create a new department that requires some checking operations, and adding a row into the auditing table that keeps track of who created the new department and when they did it. In such a case, if these actions are implemented within a procedure, users are unawre of them and are relieved from executing these actions themselves for every insert operation.

Such objects as synonyms, views and procedures can be used for implementing the transparency of distributed databases. Note that every object is created by a particular database user and belongs to this user's schema. Other users need special permissions to be able to use these objects. Schemas and security issues are discussed in detail in Chapter 4.

Example: Building the Physical Model of Data

Let us discuss the physical design of the table Employee – a part of the physical data model for the Manufacturing Company case. This discussion is provided for the centralized database solution; the distributed design for this case is shown in Chapter 3.

Building the physical model of data includes making decisions about data types for the columns, additional integrity constraints, the initial size and growth of the table, the type of storage, and storage parameters.

Decisions about a column data type are based on user requirements for a column's possible values and the expected data manipulations on the column. The following data types are chosen for the columns of the table Employee:

- *ID*. This column is the primary key of the table and its values are assigned within the company. Integer numbers are a good choice for this column for a couple of reasons. First, this makes the column short and simple – necessary properties for the primary key column. In addition, it allows for using the IDENTITY feature mentioned above or a special Oracle object – sequence – to automatically generate new unique values of this column for the new rows.
- *emplName*. VARCHAR2(30)The names of persons include letters and are of variable length. Columns of such type usually have the VARCHAR2 data type. The length of the column is limited to 30 symbols.
- *emplType*. VARCHAR2(10)allows storing the three possible values('Full-time', 'Part-time', 'Consultant')of the column. However, a better solution might be to store shorter values, e.g., the first letter, which will not only reduce resources usage but will also reduce the possibility of error when entering the value – we suggest CHAR (length 1 is the default).
- *deptCode and titleCode*. These columns are the foreign keys to other tables of the database and we will use the same data types as these columns have in their parent tables Department and Title, respectively. Assume that both of these columns have the CHAR data type as both may contain digits and letters. Both columns are fixed length: three symbols for *deptCode* and two symbols for *titleCode*.

The NOT NULL constraint on the column emplName enforces the requirement that each employee in the database has a name. The CHECK constraint on the column emplType ensures that values of the column are limited to 'F, 'P, and 'C'.

The table is stored as a heap. The table blocks do not need much free space because columns ID, deptCode, and emplType are fixed-length, and columns emplName and emplType are unlikely to change often. Therefore, PCTFREE can be low. Users do not expect many staff changes, which means there will not be too many deleting and inserting operations, and, therefore, PCTUSED can be high. Given these requirements, PCTFREE can be set to 5, and PCTUSED to 90.

Users expect to support about 4000 employees, and we can roughly estimate the initial size of the table using the average row length and the percent of free space. The average row length depends on the average length of the emplName column, which we estimate as 20. Then the average row length is 30 bytes: 4 bytes for ID, 20 bytes for emplName, 1 byte for emplType, and 5 bytes for deptCode and titleCode.

The initial size of the table is estimated as 126000 bytes: 30 bytes multiplied by 4000 rows and increased by 5% of free space. The estimation is rough and does not consider some other storage factors, e.g. space in the data blocks occupied by block headers and directories; the initial size is rounded to 130K.

The table is not expected to grow fast; and we will specify five possible extensions of 50K each.

Here is the resulting physical model of the table:

```
CREATE TABLE Employee (
    ID INTEGER PRIMARY KEY,
    emplName VARCHAR2(30) NOT NULL,
    emplType CHAR
          CHECK (emplType IN ('F', 'P',C')),
    deptCode CHAR(3) REFERENCES Department,
    titleCode CHAR(2) REFERENCES Title)
     PCTFREE 5
     PCTUSED 90
     TABLESPACE users
     STORAGE (  INITIAL 130K
```

```
NEXT 50K
MAXEXTENTS 5
PCTINCREASE 0 );
```

It is important to utilize specific features of the DBMS to make the support of data easier. In this example, we want to show how to use a special feature of Oracle – sequence – for generating unique values. We will create a simple sequence that starts with the value 1 and generates the next value with an increment of 1:

```
CREATE SEQUENCE seq_Employee;
```

To use the sequence object for generating unique ID numbers for the table Employee, we need to create a special type of procedure – a trigger on the table. The trigger that we need here will be automatically invoked for every insert statement on the table, generate the next value of the sequence, and assign this value to the ID column of the new row.

```
CREATE TRIGGER tr_ins_Employee      -- creating the trigger
BEFORE INSERT ON Employee           --the  trigger is invoked
                                    before every insert
FOR EACH ROW                        --the trigger action takes
                                    place for every inserted row
DECLARE
      vNext NUMBER;
BEGIN
      vNext:=                       -- request the sequence to
      seq_Employee.NEXTVALUE;       generate the next value
      :NEW.ID := vNext;             -- assign the new sequence
                                    value to ID of the new row
END;
```

Every insert statement

```
INSERT INTO Employee (emplName, emplType, deptCode, titleCode)
VALUES (…);
```

invokes the trigger and generates a new value for the ID column. Generating the new value and assigning it to ID is transparent to users.

Summary

The physical data model specifies where the data are stored and how data storage is organized. Building the physical model requires knowledge of the conditions under which the database will be used and user requirements for the expected sizes of tables, performance of various requests, availability of data, number of users concurrently working with data, and other considerations. In addition, developers have to understand the features of the DBMS and utilize these features properly to build the physical model.

The main goals of a good physical data model are providing the required performance and availability of data, economical space usage, and easy database maintenance. This chapter discusses the basic features of DBMSs for achieving these goals. Each DBMS has additional specific tools and features, and developers need to understand them and use them to full advantage.

The important steps in building the physical model are:

- *Choosing the data types for the columns of tables*. A data type has to: 1) represent all possible values of a column and 2) allow for all needed operations on the column's values. A data type also 3) imposes constraints on the column's values and it can be used to enhance the column's integrity. Additional considerations for choosing a data type include 4) performance (e.g. `VARCHAR2` vs. `CHAR`) and 5) the economical use of storage space. Some DBMSs offer specific data types that can substantially reduce the cost of data maintenance (e.g. `IDENTITY`).
- *Applying additional integrity constraints*. DBMSs support additional integrity constraints not included in the relational model, such as `NOT NULL`, `CHECK`, and `UNIQUE`. These constraints implement user requirements that are not presented in the relational data model and improve data integrity. More complicated user requirements are implemented with the help of special database procedures – triggers.While some requirements can be supported by the database application, we strongly advocate making the database responsible for data quality – such quality support measures cannot be bypassed and will be implemented only once to be leveraged by any application.
- *Defining storage type*. The type of storage organization is dependent upon how the data will be used. Developers have to choose between heap (or random) and organized storage. The most common type of storage organization is clustering, when rows that have the same value of a column (index cluster) or the same value of a special function applied to a column (hash cluster) are stored in the same blocks. When a request for data is constrained by conditions on the column used for clustering, because the requested rows are stored together, the system can access all needed data in a few block reads. DBMSs support other types of organized data storage, e.g. index-organized tables and partitions. In some cases, achieving the necessary performance requires the distribution and localization of data. A special case of data distribution – a replicated database – is the ultimate resolution of the problem of data reliability and availability.
- *Specifying storage parameters*. Storage parameters of a table define where (in which tablespace) the table is stored, its initial size and growth, the packing of data in data blocks, and other physical properties.

Review Questions

1. What are the main goals of physical data design?
2. What DBMS features are used for performing physical design?
3. How does storage organization affect performance on the database?
4. What types of organized storage do you know?
5. When is it recommended to use clusters?
6. In what situations is it beneficial to distribute data?
7. What are the benefits of the 3-tier architecture of a database?
8. How does the DBMS read data from a table and process the data?
9. What table storage parameters do you know? What is the role of each of them?
10. What types of organized data storage in Oracle do you know?
11. How would you implement transparency for a distributed database?

Practical Assignments

1. Describe the database storage hierarchy. Explain how a table's data is stored in a database.

2. Explain how parameters PCTFREE and PCTUSED control the packing of data in data blocks.

3. Choose the data types for the table `Student (ID, name, dateOfBirth)` and explain your decisions.

4. Choose data types for the columns of the table `T (A, B, C, D)` and specify column constraints for the following conditions:
    - Values of the column B are alphanumeric strings five symbols long, and for each row of the table, the column B must always have a value and this value has to be unique.

- The column C is designed to store numeric values not greater than 50000; users often request totals of the column C.
- The column D stores dates of the year 2019.

5. Estimate PCTFREE and PCTUSED for the following tables and situations:
   a. `CREATE TABLE T1 (field1 NUMBER, field2 CHAR(3), field3 DATE)`. Intensive updating, inserting and deleting activities are expected.
   b. `CREATE TABLE T2 (field1 NUMBER, field2 VARCHAR(250))`. Intensive inserting and deleting activities are expected.
   c. `CREATE TABLE T3 (field1 VARCHAR2(50), VARCHAR2(300))`. Intensive updating activities are expected.
   d. `CREATE TABLE T1 (field1 NUMBER, field2 CHAR(3), field3 DATE)`. Intensive retrieving activities are expected.
   e. `CREATE TABLE T3 (field1 VARCHAR2(50), field2 VARCHAR2(300))`. Intensive retrieving activities are expected.

6. A database contains two tables `T1(A, B, C)` and `T2 (C, D, E)`. The column C of T1 is the foreign key to T2. Explain in which of the following situations it is reasonable to use clusters, and create clusters and clustered tables.
   a. The data in the tables are often modified.
   b. The performance of select queries with conditions … `WHERE C = x` on the table T1 is important.
   c. The performance of queries with conditions … `WHERE C <> x` on the table T1 is important.
   d. The performance of queries with conditions … `WHERE C > x` on the table T1 is important.
   e. The performance of the queries on the join of the tables is important.

7. Explain when it is beneficial to store the table `T (A, B, C)` as index-organized.

8. Define the difference in conditions for your decisions to store the table `Employee` as clustered or partitioned by the attribute `emplType`.

9. Define the difference in conditions for your decisions to store the table `Employee` as distributed or partitioned by the attribute `emplType`.

10. What would be your choice of the storage organization of the table Employee if the following querying patterns were the most important ones:
    a. … `WHERE ID = x;`
    b. … `WHERE emplName = x;`
    c. … `WHERE deptCode <> x;`
    d. … `WHERE titleCode BETWEEN x and y;`
    e. … `WHERE titleCode =x;`

11. Describe distributed data design for one of the case studies from Appendix 1.

12. Build the physical data model for one of the case studies from Appendix 1.