Chapter2

Databases - basic concepts

Essential reading

"*An Introduction to Database Systems*", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter1 and Chapter2* (pp. 2-74).

Alternatively

"*Database Systems: A Practical Approach to Design, Implementation and Management*", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 1 and Chapter 2* (pp. 4 – 70).

Further reading

"Database Systems Concepts", second edition, by H. F. Korth and A. Silbershatz, published by McGraw-Hill, 1991, [ISBN 0-07-100804-7], *Chapter 1* (pp.1-20) — for a more concise and basic reading

Introduction

Informal introduction

We shall start the study of this subject by asking ourselves the (obvious) question "What is a database?". If you think for a few seconds, you will definitely remember some real life situations in which a *database* was used. For instance, you are in a shop and the thing you are trying to buy is not available on the shelves. A shop assistant kindly approaches you and asks you "How may I help you, madam?"[1]. "Well, I'd like to buy a skirt like this ... and like this. A friend of mine bought one a few days ago", you say, "but now it is not available on the shelves. Could you please check whether you still have it in stock or not?". "I won't be a second, madam", the assistant answers, walking towards a computer terminal and after a minute or two, after he *checked the database*, he returns: "The item you are asking for is not available at the moment in our shop, but you can find it at our branch on Oxford Street". Alternatively, I could offer you a very similar item and at a lower price too (he found it by searching their products database according to your description).

Let us consider a different situation. Suppose you are trying to find out whether a certain book exists in Goldsmiths College's library. You connect yourself to the library's *database* (catalogue)[2] and *search* it providing the title of the book. The (database) system returns you the information that there are two volumes, out of a total of four, available for a one month loan and also tells you where they are stored. You can also reserve a volume if you want. The list of examples can continue ... when using your credit card, a database that stores details about your account has to be accessed; a factory maintains a database containing information related to their suppliers, the products that need to be supplied, the products they manufacture, the employees, the departments, etc.

---

[1] I hope that the male readers will still carry on reading!

[2] You can even do this remotely by "telnet"-ing the library's server from your own computer.

Activity: Before you read further, try to identify the features of a database system based on some applications you can recall.

A *database system* is a system that *stores* data and offers at least the following features (this is a working definition; it is going to be refined in a subsequent section):

- *retrieve* data
- *insert* new data
- *delete* unnecessary data
- *update* old data

Consider, for example, the information that a CD store maintains about the products it sells. The corresponding data could be organised in a form of a table, as shown in Figure 1, and can constitute a part of the shop's database.

Title	Band / Artist	Genre	Location	Price	InStock
Closing Time	Tom Waits	pop	ShaCow29	14.99	22
Golden Saxophone	collection	atmosphere	CaCk00	2.99	3
The best of John Williams	John Williams	classic (guitar)	MA10	10.49	15
Unplugged	Eric Clapton	blues	VP111	14.99	7
Brahms - Le Sinfonie	Celibidache - Milano	classic	ME01	3.99	24
...	...	...	...	...	...

Figure 1: A part of a CD shop's database

"Golden Saxophone" is old and nobody seems to buy it anymore so the three CDs are going to be thrown away. Therefore, the corresponding *record* - i.e. row in the table - has to be *deleted*. There is no big demand for John Williams's CD, so the price is going to be brought down to 8.99 - therefore, the corresponding *field* - i.e. column in the table - of the record has to be *updated*. A set of 10 new CDs, "Men's Power, Posh Girls", pop music, costing £14.95, has been made available in the shop - therefore, the corresponding information (record) must be *inserted* into the table. Eric Clapton "Unplugged" is quite popular, so 10 new CDs are purchased from the supplier, and the price is to be raised by £2. The content of the database after these changes is depicted in Figure 2.

Title	Band / Artist	Genre	Location	Price	InStock
Closing Time	Tom Waits	pop	SCow29	14.99	22
The best of John Williams	John Williams	classic (guitar)	MAv10	8.49	15
Unplugged	Eric Clapton	blues	VP111	16.99	17
Brahms - Le Sinfonie	Celibidache - Milano	classic	MExx01	3.99	24
Men's power	Posh Girls	pop	Swaa89	14.99	10
...	...	...	...	...	...

Figure 2: The part of the CD shop's database (Figure 1) after some update operations

From this example we can infer that a database has a *structure* and a *content*. The structure is represented by the heading of the table. The content is represented by the body of the table. The content changes in time, it has a quite dynamic nature. The structure can change too, but it far less dynamic than the content. For instance, a new column – the year in which the product was released – can be added to the above table. The structure is called the *intension* and the content is called to the *extension* of the database (this issue is tackled in more detail later in this chapter).

Another aspect, not that apparent from this example, is the fact that a database stores a *large amount* of data.

So far, a database system is, for us, nothing more than a system that *manages data*. But is any system that manages data a database system? Is there anything that all database systems have in common, that distinguishes them from other software systems? The

answer is obviously yes. In order to understand the "*database approach*", we shall first have a brief look at *file based systems*. In appearance (behaviour) they are similar to database systems, but they are conceptually (qualitatively) different. We shall identify the drawbacks of the file-based approach to data management and then introduce the database approach as a solution to most of these drawbacks.

File based systems

We shall start with the definition of file based systems.

Definition: A file based system is a collection of application programs, each managing its own data.

In a file based system, permanent data is stored in various files of ad-hoc structures. Each application program defines and handles *its own* data files independently on the others. This approach is called the *de-centralised* approach. Moreover, each application program works with its data at the physical level[1], manipulating records as they are organised on the external support (in the persistent memory).

Take, for example, an estate agent's office, for which we shall consider the Sales and the Contracts department. Each department maintains its own data, as depicted in Figure 3.

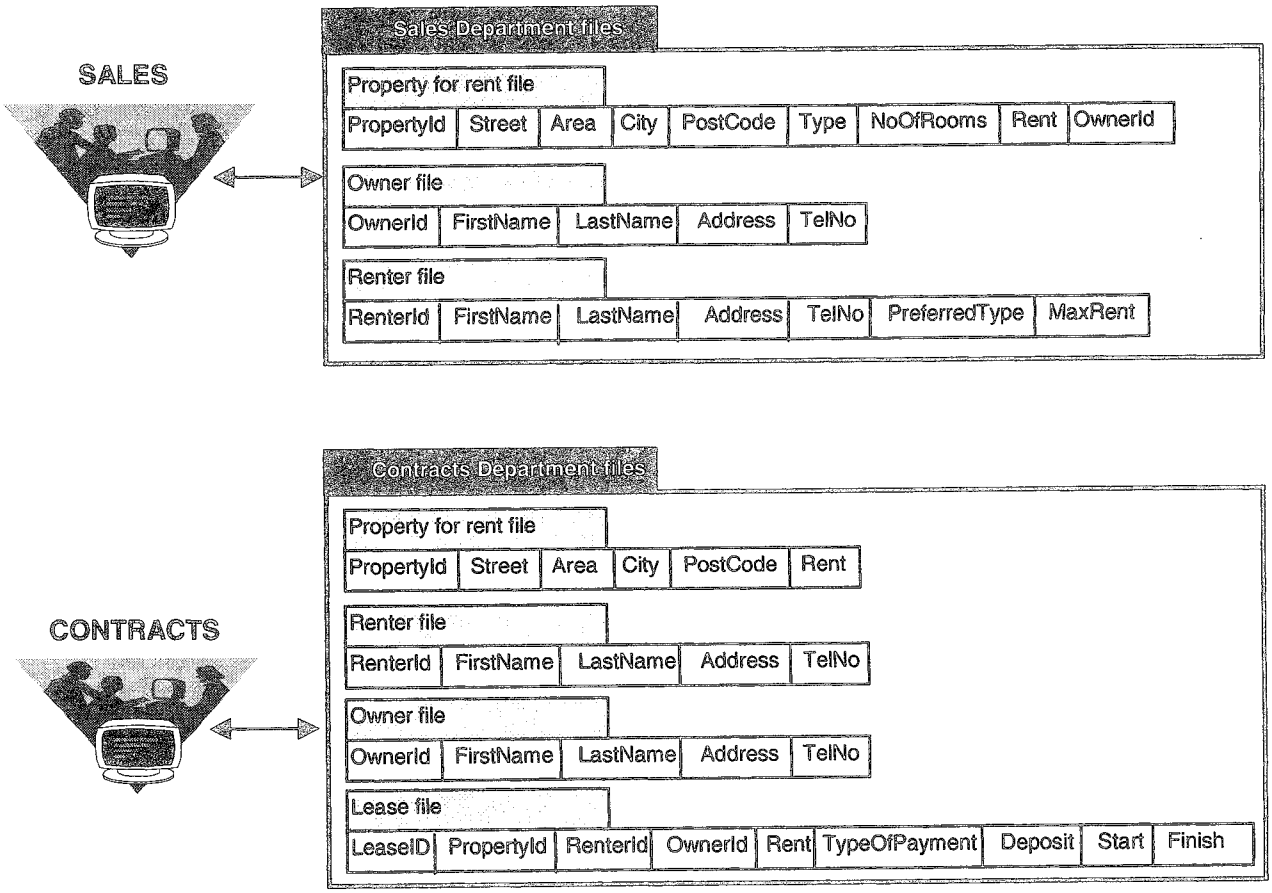

Figure 3: A file based system for an estate agent's company

[1] This structure, however, is not "purely" physical, as linked clusters would have been. It is physical in the sense that it is dependent on the platform, because the access to the files is made through the primitives of the operating system.

The Sales Department needs:

- sufficiently detailed information about the properties for rent, so that adequate advice can be given to customers (e.g., Type and NoOfRooms in the Property for rent file);
- sufficiently detailed information about customers, such that their needs can be appropriately matched to the offer (e.g., PreferredType and MaxRent in the Renter file);
- "identification" information (such as name, address, phone-no) about customers, the properties on offer and their owners.

The Contracts Department needs:

- detailed information about the renting contracts (in the Lease file);
- "identification" information (such as name, address, phone-no) about customers, the contracted properties and their owners.

Certain drawbacks of this solution are apparent. Yet, they are limitations of the file-based approach in general. The most important are enumerated below.

Duplication of data Different applications might have to make use of the same information, but, because each application has its own files, data is duplicated (e.g. the "identification" information in the example above). This aspect has at least two detrimental consequences. On the one hand, duplication is *wasteful*. On the other hand, data can become *inconsistent*, in that it can have different values in different files (belonging to different applications), even though it represents the same bit of information. For example, the address of an owner, Mr. J. Morris, might be updated in the Owner file belonging to the Sales Department, while the Contracts Department might still have Mr. Morris's old address.

Separation and isolation of data Data is *scattered* among different files, each file belonging to a certain department. A department has access to its own files, but no access to the files of the other departments. Therefore, files belonging to different departments *cannot be used together* in order to create more complex data. Moreover, because they are based on different infrastructures (platforms, development software, etc.) files belonging to different departments cannot be transferred (copied) across.

Program-data dependence Each file belongs to a certain application program. The (physical) structure of data is defined inside the application program; this could easily (and usually does) lead to *incompatible file formats* between applications, therefore to the impossibility of sharing data between them. Another aspect is that *data definition is embedded in the application program*. Thus, if the physical structure of data is to be changed (e.g. instead of representing a year with two digits, it is to be represented with four[1]) then the application program must be changed. Moreover, *the methods of access and manipulation of data are embedded in the application program* (e.g. fixed queries); to change them, the application program must be modified.

In the file-based approach, the accent is placed on functionality (provided by the application program); data modelling takes only a secondary priority. This approach leads to the above mentioned drawbacks. However, if the approach is "turned upside-down" and data is considered as the central point, then these drawbacks can be removed. Informally, this represents the database approach.

[1] This is a reference to the "millennium bug". Consider a banking system that used a two-digits representation of the year. Suppose you opened an account in 1998 (represented as"98"). What interest would you be paid in January 2000 (represented as "00")? Would it be the interest for –98 years? Is this possible? Would it be for 98 years? Is this correct?

Databases and database management systems

We shall start with the definition provided in (Connolly 1999).

***Definition:** A database is a (shared) collection of logically related persistent data (including its description) as part of the information system of an organisation.*

Let us now explain this definition. A database is a *large* repository of data, in which data is *defined once* and *stored once*. Data that, in the previous approach, was scattered in different files, of different formats, having different owners, is now *integrated* with minimum redundancy[1] (duplication), as a single resource. Different application programs will *share* this common resource, usually concurrently. The representations in Figure 4 and Figure 5 illustrate the differences between the database approach and the file-based approach.

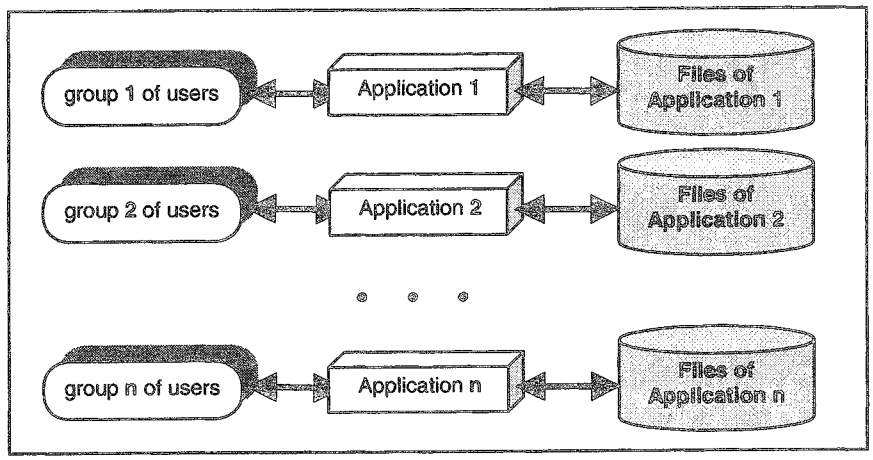

Figure 4: The file based approach

The raw data, integrated in a common (for all applications) database (see Figure 5), is managed by a *database management system* (DBMS), which provides a shared access to it, for all the applications in the system.

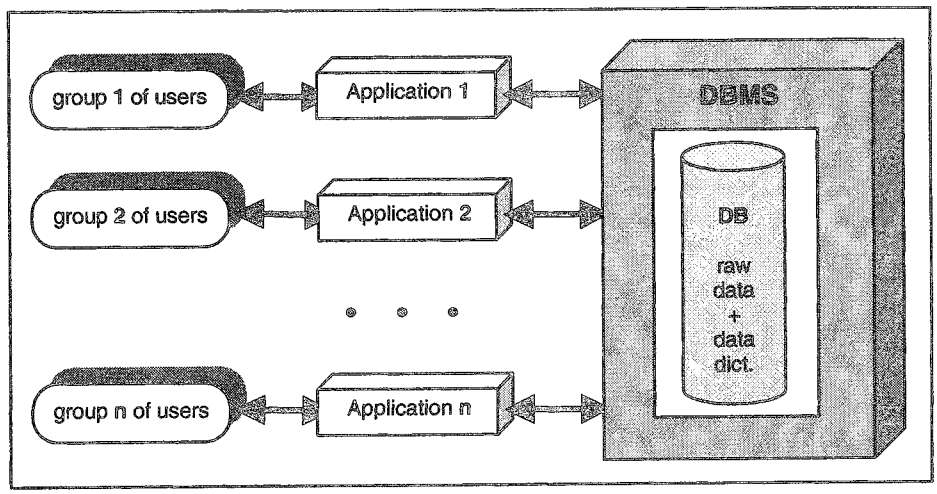

Figure 5: The database approach

[1] There are situations, as you will see in the next chapters, when redundant data is necessary. For example, to store intermediate results that are often used (by means of snapshots) or to implement the atomicity of a set of operations (transactions). However, in such situations redundant data is intentionally included, it does not arise as a drawback.

***Definition:** A database management system is a software system that provides a set of primitives for defining, accessing and maintaining a database.*

A database stores both the raw data and its description. We say that the information[1] stored in a database is self describing. The description of the raw data is known as the *system dictionary*, *data dictionary* or *meta-data.*

The consequence of this approach is *program-data independence* which means that the structure of data may change without affecting the application programs that use it. This basic definition is going to be refined and better explained later in this chapter.

This approach, of separating the data definition from application programs, is similar to *data abstraction*, wherein the internal definition of an object is kept separate from its external definition. An outside system can only see the exterior of the object. As far as the external definition remains unchanged, any changes in the object's internal definition do not affect the outside system.

A database management system automatically performs a lot of the housekeeping tasks (which, otherwise, would have been the responsibility of the application programmer). As a result, the user – i.e. the person that defines and uses the database – is presented with a "clean" and "powerful" set of "tools" for database development and exploitation. A more detailed description of both a database systems and a database management systems is provided in the following sections. At this stage, it is important that you broadly understand why database systems are needed and what their main benefits are.

The three level ANSI / SPARC architecture of a database environment

We mentioned program-data independence as one of the most important advantages offered by the database approach. How should a DBMS be developed to achieve this? Is there a *general structure* for which DBMS compliance would guarantee program-data independence? The answer is yes. This independence can be achieved if data is considered at two (or more) levels of abstraction: one, low level, describing how data is organised on the physical support[2]; another one, high level, describing the (logical) structure of data, irrespective of its physical representation.

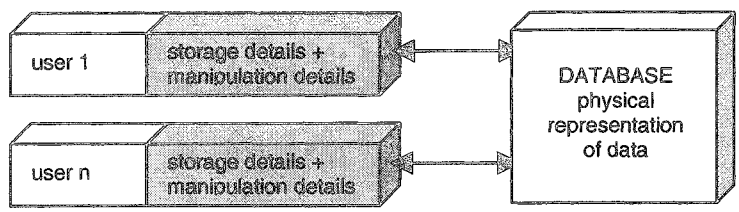

Figure 6: The users have direct assess to the physical representation of data

This idea can be further refined:

- Users and application programs should be "protected" from the aspects related to the physical representation of data, such as storage and accessing details. They should be freed from considering such aspects and should be able to take into consideration only the *logical* structure of data. Rather than having to deal with such aspects in each application program (see Figure 6), it would be much better if these problems were to become the responsibility of the system (DBMS) that manages data (see Figure 7).

[1] In this module "data" and "information" are used synonymously. Certain authors, in certain situations distinguish between the two; however, in this context such a distinction is not necessary.

[2] The physical support is customarily referred to as "persistent/permanent support/memory".

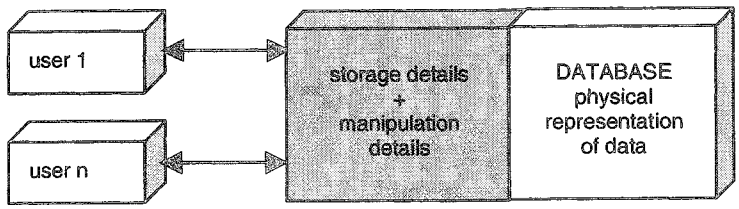

Figure 7: The users are "protected" from the physical data representation details

- Conversely, it should be possible to change the physical representation of data, providing its logical structure remains the same, without affecting the users.
- The database integrates the whole information required within a certain organisation. However, individual users only need access to certain parts of this "pool of information". Therefore, each user needs to have a customised view of the database and needs to be able to change it without affecting the other users.

These aims are achieved if the database system is developed according to the ANSI/SPARC[1] architecture (ANSI, 1975). Most of today's DBMSs comply with it.

The ANSI / SPARC architecture consist of three levels of abstraction (see Figure 8). The *external level* represents the way data is viewed by individual users. The *conceptual level* represents the way the organisational data (i.e. all data that is relevant for the organisation) is structured. The *internal level* – represents the way data is physically stored.

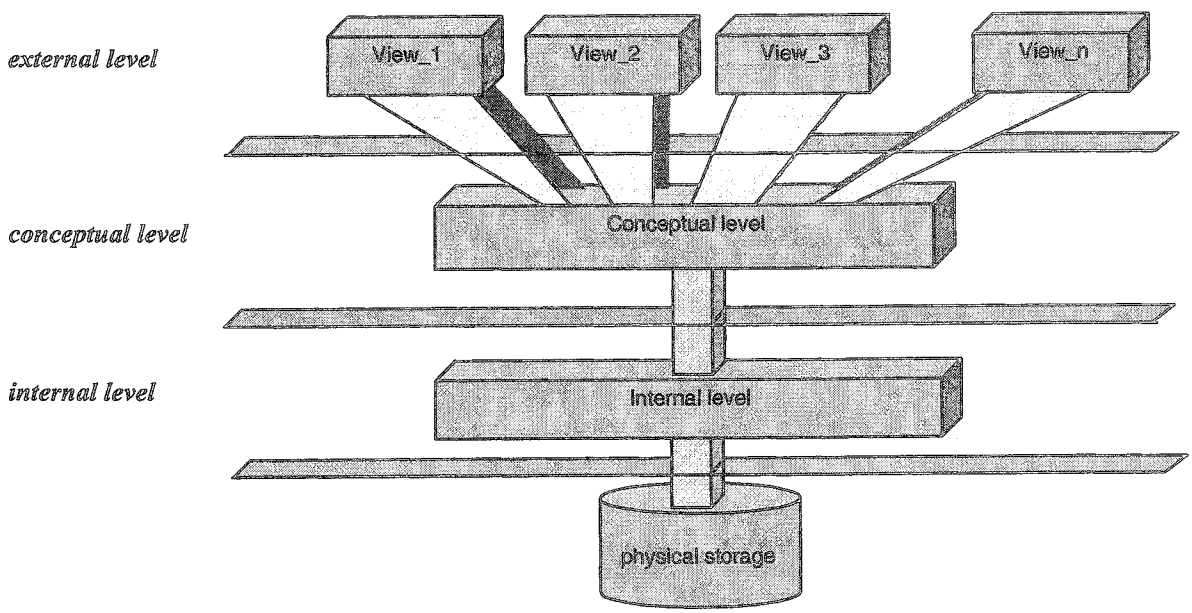

Figure 8: The three ANSI/SPARC levels of abstraction for data

The external level incorporates each user's view of the database. A user's external view of the database consists only of the data needed by that user. Other data might exist in the database (since data is centralised in a common resource), but the user does not have to be aware of it.

[1] The American National Standards Institute (ANSI) – Standards Planning and Requirements Committee (SPARC) adopted this architecture in 1975.

For instance, suppose that the database of a software company that includes information about its employees. On the one hand, the Personnel Department's view of the employees (i.e. the data that is relevant to them), on the one hand, might consist of: name, address, sex, date of birth, qualifications, department for which the employee works, NI number, current salary, job contract details (duration, date of employment, etc.), details about previous job (optional). The Personnel Department has to be able to access data about any employee of the company.

On the other hand, the Development Department's view of the employees might consist of: ID (identification within the department), name, telephone number, timetable, the projects in which the employee is involved including the position in each project, the objectives and their deadlines. Only the data about its employees is relevant for (i.e. needed by) the Development Department.

The whole information about the company's employees is stored in the database. However, the two departments need access to different projections of data. The data relevant to each department represents the department's external view of the database (Figure 9).

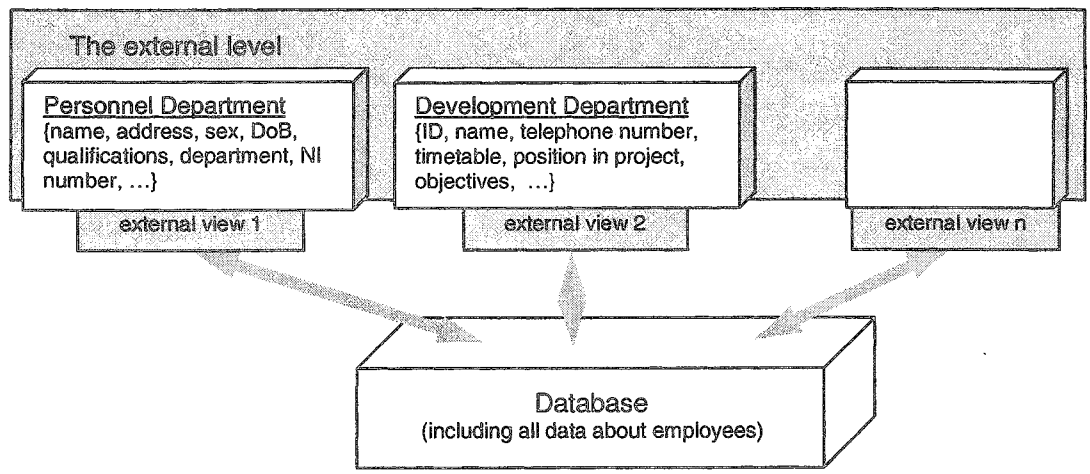

Figure 9: The external level of a database

Moreover, different users may want to see the same data, but in different formats (e.g. the name might consist of two fields, first name and last name, or just one field, comprising the whole name). The external level might also include derived data (i.e. calculations based on stored data; e.g. age, based on date of birth).

The conceptual level represents the logical structure of the database (of the overall data required by the organisation). It can be viewed as the or the union of the whole set of views at the external level. Conversely, any view should be derivable from the conceptual level. The conceptual level represents the information stored in the database about the real life system's entities (objects) and the relationships between them. The representation of data at this level is still independent from any physical considerations, i.e. it specifies "what" rather than "how" it is stored.

The internal level describes the physical representation of data. As opposed to the conceptual level, the internal level specifies *how* data is stored. It is at this level where the physical data structures and file organisations are defined. The internal level is situated at the interface between the DBMS and the Operating System (OS). It is quite common that the OS's file management primitives are used by the internal level of the DBMS. However, there is no clearly defined boundary between the OS and the internal level. Therefore, below the internal level there might exists another level, namely the *actual* physical support (cylinders, blocks, clusters, etc.) (see Figure 8).

Schemas and mappings

Before we are going to look into how the three abstraction levels are defined within a DBMS, the distinction between the description of the database and the content of the database must be made clear.

Definition: *The description of the database is called the* database schema *or the database* intension; *this is specified at the creation of the database and it is not expected to change very often.*

Definition: *The raw data that populates a database at a particular point in time is called a database* instance *or the* extension *of the database.*

Consider, for example, a (miniature) database that maintains information about the employees of a company. The required information for each employee is: name, date of birth, address, job, and pay scale. Suppose that the database is organised in the form of a table. Then, the specification of the table's heading can be considered as the database schema (or database intension), whereas the content of the table at a specific moment in time can be considered as the database instances (or database extension) (Figure 10).

Database schema (names of columns + the types associated with them)

Name	DOB	Address	Job	Scale
String	*Date*	*String*	*String*	*Int*

A database instance

Name	DOB	Address	Job	Scale
A. Johnson	2/04/1960	London	Programmer	12
B. Holiday	3/10/1947	Leeds	Analyst	14
C. Clark	12/08/1971	York	Programmer	10

Figure 10: Database schema and database instance

The overall database schema consists actually of three types of schemas, one for each level of abstraction:

- *external schemas* - describe the external level; one for each view;
- *a conceptual schema* (only one) - describes the conceptual level; all the definitions should only take into consideration the structure of data and should totally disregard the implementation aspects;
- *internal schema* - describes the internal level; it defines the physical records, the methods of representation, indexes, etc.

Accordingly, two mappings must also be defined between the three schemas, namely:

- *external / conceptual* - defining the correspondence between an external schema and the conceptual schema;
- *conceptual / internal* - defining the correspondence between the conceptual and the internal level.

Figure 11 illustrates these ideas on a (minuscule) database of a company, maintaining information about its employees. Two *external views* of the database are considered in this example, one as needed by the Finance Department and one as needed by the switchboard. The information needed by the Finance Department refers to the name age and salary of each employee and is defined by its corresponding external schema. Each employee must be uniquely identified. Since two employees might have the same name, a unique identifier, ID, is used.

The information needed by the switchboard, defined by the respective schema, only refers to the name, job and telephone number of each employee. Note that the switchboard needs separate access to the first and last name of each employee. For a unique identification of each employee, the switchboard uses the job title in conjunction

with the name (under the assumption that two employees having the same name will not have the same job title as well).

Other external views might as well exist, e.g. for the Personnel Department, but we shall not worry about them here.

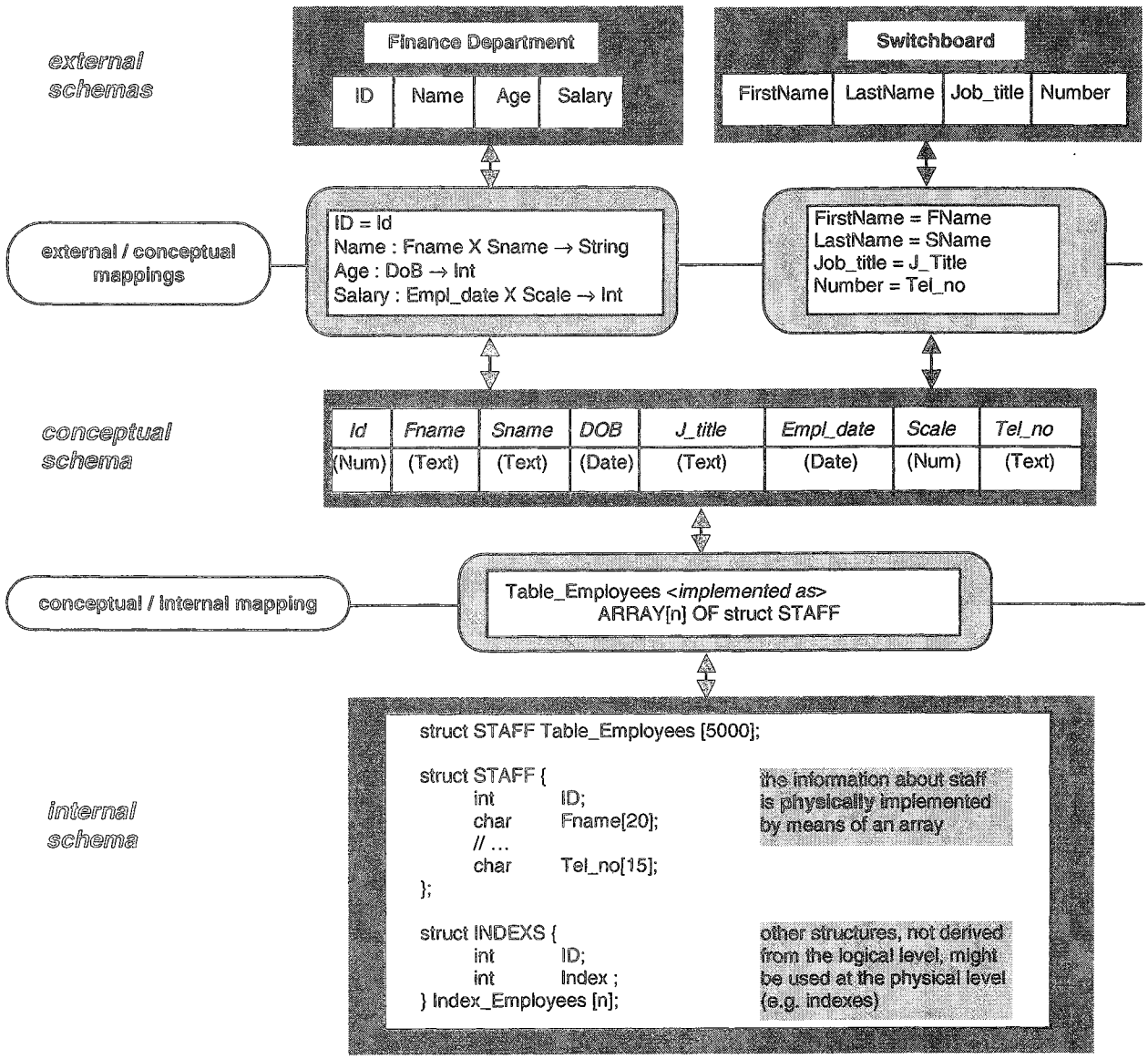

Figure 11: Schemas and mappings

The *conceptual schema*, that defines the conceptual level, results as the "common denominator" of the two views. It specifies the identifier, first name and last name, date of birth, job title, employment date, salary scale and the telephone number of each employee.

The link between the external schemas and the conceptual schema is made by the *external / conceptual mappings*. The mapping corresponding to the Finance Department's view is defined as:

Finance Department schema	conceptual schema
ID = Id	Id
Name = concatenate(Fname, Sname)	Fname, Sname
Age = current_year() - year_of(DOB)	DOB
Salary = formula(Empl_date, Scale)	Empl_date, Scale

The other mapping, between the Switchboard's view and the conceptual schema is self explanatory.

The *internal schema* consists of the data structures that are used to represent (implement) the conceptual schema. For the above example, this is struct STAFF, in a C-like hypothetical language. It also can include other structures (i.e. not derived from the logical level), used for pragmatic reasons (e.g. efficiency). In the above example, an index was defined, INDEXS, in order to make the retrieve operations (from Table_Employees) faster.

The *conceptual / internal mapping* links the definition of data at the conceptual level (the *what*) with the way it is actually represented (the *how*); the table Employees is represented (implemented) as an array of records of type "struct STAFF". Note that the index, "Index_Employees", is used purely at the internal level (i.e. it is not mapped at the conceptual level). The reason stems from the fact that the index does not describe the data as such, but it is rather a means for facilitating the access to data.

As a final point, the issue of data independence can now be reconsidered. One of the main advantages provided by the three level architecture of a database system is the provision of data independence. There are two types of data independence:

Definition: Physical data independence is the immunity of application programs to changes at the internal / physical level.

Definition: Logical data independence is the immunity of application programs to changes at the conceptual level.

For instance, in the example above, the Employees table can be represented (implemented) by means of a linked list. If the conceptual / internal mapping is accordingly modified, provided that the conceptual schema stays the same, the application programs (situated above it) remain unaffected.

Logical data independence is more difficult to achieve since application programs typically rely heavily on the logical structure of data. However, suppose another view is needed, for the Personnel department, which requires information about the address and the family (marital status, dependants) status of each employee. The conceptual schema can be extended accordingly, without affecting the other two views.

The components of a database environment

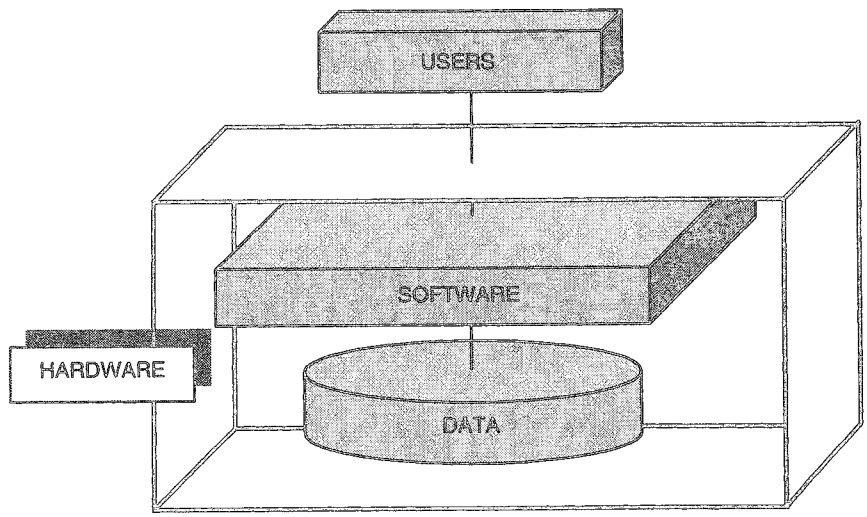

Figure 12: The components of a database system

In the main, a database environment consists of (see Figure 12):

- *data* - representing the information needed for a certain organisation;
- *hardware* - as the support for both the stored data and the software component;
- *software* - serving two purposes: (1) the management of the stored data and (2) a further processing of data according to the user's needs;
- *people* - broadly divided into two categories: (1) the developers of the database system and (2) the users of the database system.

Data

Data can be classified into two categories, namely:

- primary data – data that is stored on permanent support (such as disks or tapes).
- derived data – information that can be inferred from primary data.

On the one hand, the derived data can be viewed simply as the output of the application programs (the results of processing of the primary data), in a form suitable for the user's needs. On the other hand, it can be viewed as the input provided by users, which is then processed by the application programs, in order to be stored on the permanent support.

The focus of a database system is on primary (stored) data. This has to be appropriately identified, appropriately described (modelled) and then accordingly implemented. The primary data has two important characteristics; it is:

- *integrated* – rather than existing in separate systems (e.g. belonging to different departments of an organisation), it has been gathered together into a single system;
- *shared* – all the applications belonging to the corresponding information system have common access to (parts of) it
- *extensive* – database systems are developed for data intensive applications; it is cheaper to use other approaches (not the database approach) for applications that do not require the manipulation (storage and retrieval) of a substantial amount of data.

Finally, stored data, as previously mentioned, does not include only the raw organisational (i.e. relevant to the organisation) data, but also its description (meta-data or system dictionary or catalogue).

Hardware

In terms of hardware, the requirements for a database system can vary substantially, from a single PC to a network of computers. The hardware specifications are dictated by the requirements of the organisation. Financial constraints imposed by the policy of the organisation play also an important role in specifying the hardware resources. Two major aspects must be considered.

- there should be sufficient memory space
  - *secondary (permanent) storage* space (for example, disk space), so that all the required data can be stored;
  - *primary* (*temporary*) *storage* space (internal memory), required for intermediate results and computations, so that efficiency of data manipulation can be achieved;
- there must be sufficient *computational power* (fast processor(s)), to provide for the efficiency of data manipulation.

Usually DBMS vendors provide recommendations for a minimal configuration required for a certain size of an application. However, minimal configurations often do not provide acceptable performances.

Software

The software component can be seen as being stratified, consisting of the following three layers (Figure 13):

- the *operating system* (OS); positioned at the base, provides the necessary routines for accessing the hardware resources (such as file handling or memory management routines);
- the *database management system* (DBMS); placed on top of the OS (meaning that it uses the OS's routines), provides all the necessary primitives for data management (such as languages for schema definition, for data manipulation, etc.);
- *application programs*; placed on top of the DBMS (meaning that they use the DBMS's routines), provide data formats and computations beyond the capabilities of the DBMS.

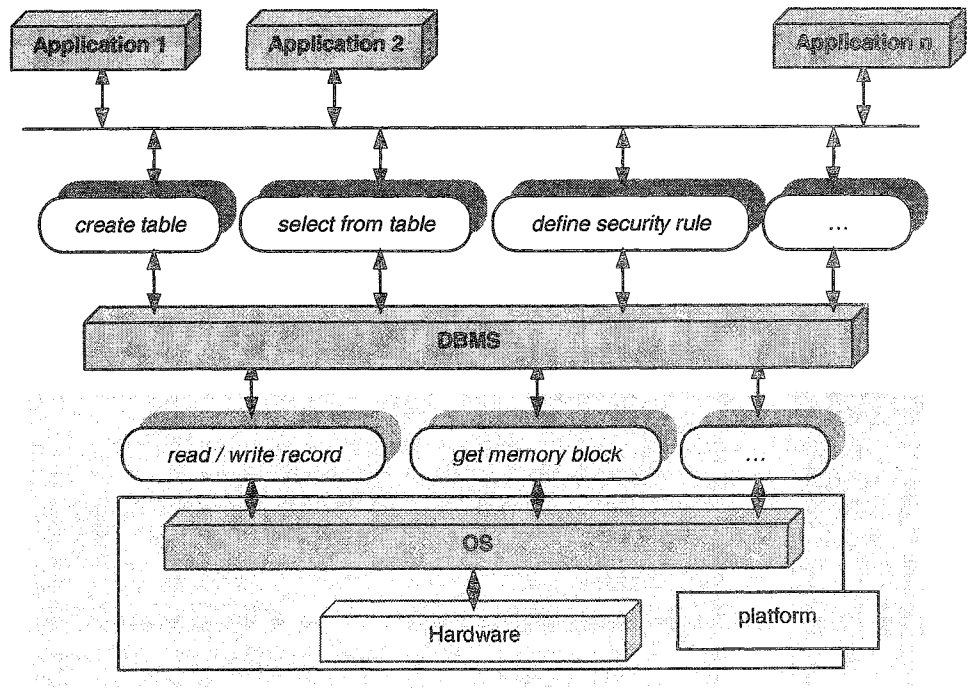

Figure 13: The stratified structure of the software component

The operating system together with the hardware - herein encompassed under the concept of "platform" - are not relevant for the topic at hand, so they are going to be left aside. Moreover, even the details of the interface between the DBMS and the OS are left out since they are platform dependent. We are mainly going to concentrate on the (generic) features provided for the user-end (applications) by DBMSs.

The DBMS is the layer that protects users and application programs from dealing with physical details of data implementation. In the main, the DBMS provides a set data management primitives of a higher level of abstraction than the OS, shielding thus the application programs from unnecessary physical details; it provides support for[1] schema definition, data manipulation, data security and data integrity.

The application programs can be of two kinds:

- user developed;
- provided together with the DBMS by its developer.

[1] The features of the DBMS are discussed in more detail in the following section.

The former class of applications are written in a third generation language, such as "C", "Java", "Pascal", "Ada", etc. Support for database access is provided by means of an incorporated or *embedded data sub-language*. The data sub-language provides a set of powerful features for data storage and retrieval, whereas the host language provides a powerful set of features for computations and data formatting. The statements written in the embedded sub-language are translated (by a pre-compiler) into calls to the routines of the DBMS.

The latter class of applications are generically known as *fourth generation tools*. Their main purpose is to allow a rapid development of user applications, without requiring the user to write any conventional code.

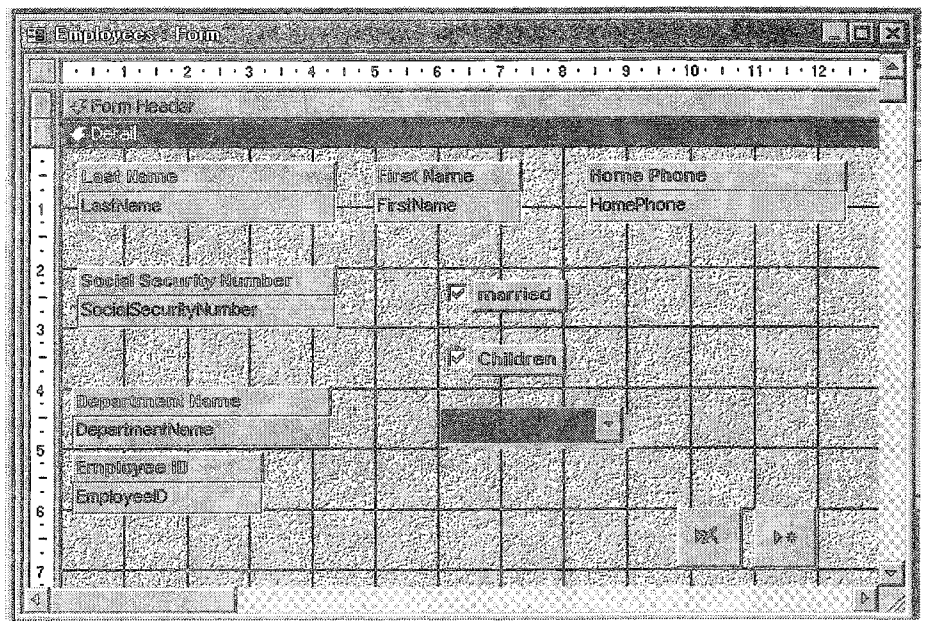

Figure 14: A form generator in Microsoft Access

Activity: Find details about commercial third generation languages embedded with a data sub-language or about commercial DBMSs accompanied by fourth generation tools. Discuss their data processing capabilities.

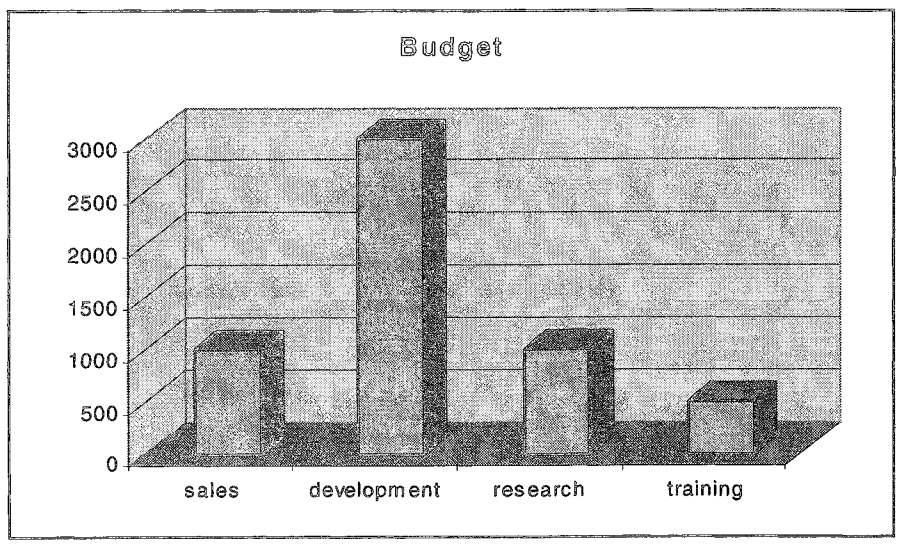

Figure 15: The output of a graphical systems

Examples of fourth generation tools include:

- query language processors;
- spreadsheets;
- reports / forms generators (Figure 14);
- graphical systems (Figure 15);
- statistical packages;
- application generators.

The DBMS can also be referred to as *server* or *backend (server)*, whereas the application programs are referred to as *clients*, or *front-ends*. The rationale is simple: clients use the services provided by a server with respect to data management. Since this division can be made, it is possible to think that the backend runs on one machine whereas the clients run on different machines, giving rise to the idea of *distributing processing*; this issue is discussed in the "Database architectures" section.

DBMSs and database languages

The database management system is the software through which all the access to the database is made. This is a concise but quite restricting definition. In reality the DBMS is responsible for much more. Some of its important features are presented below.

Data definition

The DBMS must provide support for schema definition, which includes, the specification of data types, structures, constraints and security restrictions, or for the modification of an existing schema. This is achieved by means of a *data definition language* (DDL). The statements made in DDL for a specific database system represent the system's catalogue (or data dictionary[1]). In theory, there should be a DDL at each level of abstraction (i.e. external, conceptual and internal). In practice, though, there usually exists a single DDL that allows definitions of data at each level.

Data manipulation

The DBMS must provide support for data manipulation. In particular is has to support:

- the retrieval of existing data, contained in the database;
- the deletion of old data from the database;
- the insertion of new data;
- the modification of out of date data.

This is achieved by means of a *data manipulation language* (DML). There can be a DML at each level of abstraction. At the external and conceptual level the DML is concise, comprehensive and easy to use, in other words the accent is placed on its expressive power (leaving efficiency as a second goal). At the internal level, conversely, the emphasis is placed on the DML's efficiency. This means that its statements are complex (and, probably, not that straightforwardly expressible), but quite efficient.

These languages (DDLs and DMLs) are called data *sub-languages* because they do not include constructs for the control of flow (i.e. they are computationally incomplete). Users can use them directly in order to define and access the database. However, for applications that require more complex data processing (e.g. formatting) they are embedded into a third generation language.

Some authors prefer to further divide DMLs into two categories, namely *procedural* and *non-procedural* (declarative). Within a procedural language one must specify how the result is going to be obtained / computed, whereas within a declarative language one has to specify only what result must be obtained (what the result looks like), the system being "responsible" for its computation. Since there are no pure declarative or pure procedural languages - they (DMLs) range between the two - any classifications of this

[1] Some authors consider the term data dictionary to be more general than system's catalogue (Connolly 1999).

kind are rather ad-hoc in nature. For example in certain situations SQL can be considered procedural while in others it can be considered procedural.

An important requirement for DMLs is to allow *unplanned* or ad-hoc queries, i.e. requests that were not foreseen at the time of design. A main subsequent problem is the achievement of a reasonable efficiency.

Other features that the DBMS must provide include:

- support for *data integrity* - the system must ensure that there are no "contradictions" between the data values in the database; this is achieved based on a set of integrity constraints (part of the data dictionary);
- support for *security control* - the system must ensure that data is not accessed by unauthorised users / applications; this is achieved based on a set of security rules (part of the data dictionary);
- *recovery services* - the possibility to restore the database to a previously correct state in case of a crash;
- *concurrency facilities* - the possibility of accessing the database by more than one user simultaneously;
- support for *data communication*;
- user accessible *data dictionary.*

People

People, as a component of a database environment, can be classified in four categories, according to the role they play.

Data administrator — The data administrator (DA) is a person[1] who properly understands the data requirements of the organisation and is in charge of administering the organisation's data. This is the person who decides which data is relevant and which is not, who is in charge of applying the organisation's policy, standards, who decides upon the security policy, etc. The DA does not have to be a technical person, nor a manager. Rather, the DA is somewhere in between, liasing with the management on one hand, and with the technical team, on the other.

Database administrators — The database administrator (DBA) is the technical person who is in charge of the database system. More specifically the DBA is responsible for the DB's design, implementation and maintenance and deals with both the correctness of the implementation and the efficiency of the database system. The DBA, accordingly, must have good technical knowledge and is in charge of the definition of the DB schemas, integrity and security rules, access procedures, backup and recovery procedures, performance of the system, etc.

Application programmers — The application programmers are those who write programs that perform more complex processing of data (either computations or formatting). For this, they use either a third generation language, embedded with a database language, or a fourth generation tool. Their programs are used by end-users.

End users — The end users are the "beneficiaries" of the database system. They vary from naïve users, who interact with the system via application programs developed for specific tasks (e.g. a bank clerk) to more sophisticated users. A naïve user does not have to be aware of the functionality of the DBMS. All they need is reliable and easy to use programs, so that they can use them with minimum fuss. A sophisticated user, on the other hand, knows how to access the database directly, through the database language (DDL + DML) supported by the DBMS. Sometimes a sophisticated user might even develop applications, thereby becoming an application programmer.

---

[1] In this section, "person" should be understood as either *one person* or *a group of people.*

Advantages and disadvantages of database systems

The main advantages and disadvantages associated with the database approach are not described in detail here. You are advised to keep this issue "alive" in your mind throughout the whole module, and to continuously try to refine the answers. We shall start, firstly, with advantages.

Advantages

Redundancy can be reduced

In a file based system each application has its own private files. This often leads to data being duplicated in different files, wasting storage space. In a database approach, all data is *integrated*, therefore redundancy can be eliminated. Note that not all redundancy can or has to be eliminated. Consider, for instance, the relational model[1]. The implementation of certain links between data (more precisely, certain relations between entities) can only be done if some data is duplicated. Therefore, (some) redundancy is a requirement of a correct relational model[2]. On the other hand there are situations when, theoretically, redundancy can be eliminated, but it is not, for pragmatic reasons; e.g.: to improve performance and to enhance reliability (data recovery). However, rather than using it in an ad-hoc fashion (as in the file based systems case) the DBMS provides (should provide) mechanisms for specifying such redundant data and for controlling it (i.e. maintaining the consistency of the database; see below).

Inconsistencies can be avoided

This is, partly, a consequence of the previous point. A database is in an inconsistent state if the same item of information is stored in at least two places in the database, but differently. On the one hand, the database approach reduces redundancy, therefore the possibility of inconsistent data is smaller. On the other hand, certain redundant data can be made *known* to the DBMS, so that the system automatically enforces consistency; i.e. whenever some changes are made to some data in a certain place, the same changes are *propagated* to the same data that is stored (duplicated) in other places. Extant commercial DBMSs provide only limited support for preventing inconsistencies (i.e. for only a few special cases of redundancy).

Data can be shared

Since all data is centralised, every application could see it all (if allowed by the system's security constraints). Moreover, since all data is integrated, more information can be derived from the same amount of data. Both aspects considerably improve the accessibility of data.

Data independence

The immunity of applications to changes in the physical structure of data (representation and accessing methods) - physical data independence - and, possibly in the logical structure as well - logical data independence. This issue was discussed in a previous section, so it is not described here again.

Some other benefits of the database approach are:

- the *maintenance* of the overall information system can be improved due to data independence;
- *integrity* can be maintained - any DBMS should allow the specification of *integrity constraints* on data;
- *security* restrictions can be applied - any DBMS should allow the specification of *security rules* on data and users;

---

[1] This model is going to be introduced in a subsequent section, so do not worry if for the time being you do not know what it means. It has been named only for the sake of completeness.

[2] However, taking a different point of view, this aspect might not be considered redundancy, for this "duplicated" data is not redundant, but serves a well defined purpose, namely representing links.

- *standards* can be enforced;
- *concurrent access* can be better achieved;
- better *recovery* mechanisms can be devised;
- conflicting requirements (for the information system an organisation) can be balanced (i.e. "best for the organisation").

Disadvantages

Complexity

In the database approach, the information needed by an organisation is modelled and implemented as a whole. Typically, the volume of needed information is very big, therefore the process of developing and maintaining a database system is very complex. Consequently, this process, including

- data acquisition
- data modelling and design
- database implementation
- database maintenance

is more prone to malfunctions (errors) than in the file based approach.

Along the same lines, the DBMS itself is a complex piece of software, therefore expensive (as compared to a compiler, for instance). Moreover, in order to achieve a satisfactory level of performance, expensive hardware might be required. The cost associated with database systems could become a strong financial constraint upon an organisation.

Higher impact of failure

The database system is at the core of the information system of an organisation. All data is stored centrally, in the database. Consequently, most of the applications rely on this data. If the DBMS fails, the whole organisation is paralysed.

Performance

The DBMS is a *generic* piece of software, therefore a database application might be slower, from the point of view of the individual user, than a corresponding file-based application.

Architectures of database systems

It was mentioned in a previous section that the data of an organisation, integrated in a single database, is shared between many applications. Accordingly, a "natural" organisation of a database system is the *client-server architecture* (Figure 16). The DBMS is the *server* and the application programs are the *clients*. A server can also be referred to as *back-end* and a client as *front-end*.

In the client-server architecture, the DBMS (including the database) runs on a dedicated machine – the server machine. The server machine is tailored specifically to support the DBMS, both in terms of storage space and computational power. It has to provide:

- extensive and fast external (persistent) memory;
- powerful processing capabilities (fast processors) combined with sufficient internal memory.

The main requirement for the server machine is to provide sufficient resources, so that the DBMS can respond efficiently to the requests received from the clients (i.e. to provide *what* they need and to do this *in time*). The server machine is usually expensive, therefore it is an advantage to have this resource shared by the whole organisation.

The applications are run on different client machines[1]. Each client machine is usually tailored specifically to meet the needs of a certain application. For instance, if an

[1] It is possible, though, that some applications are run on the server machine.

application program performs complex graphical processing, then a powerful graphical workstation is required, whereas if an application only performs simple data entry, then a terminal might be sufficient. If these needs change, it is only the client machine that has to be "modified", making thus the client server architecture quite flexible.

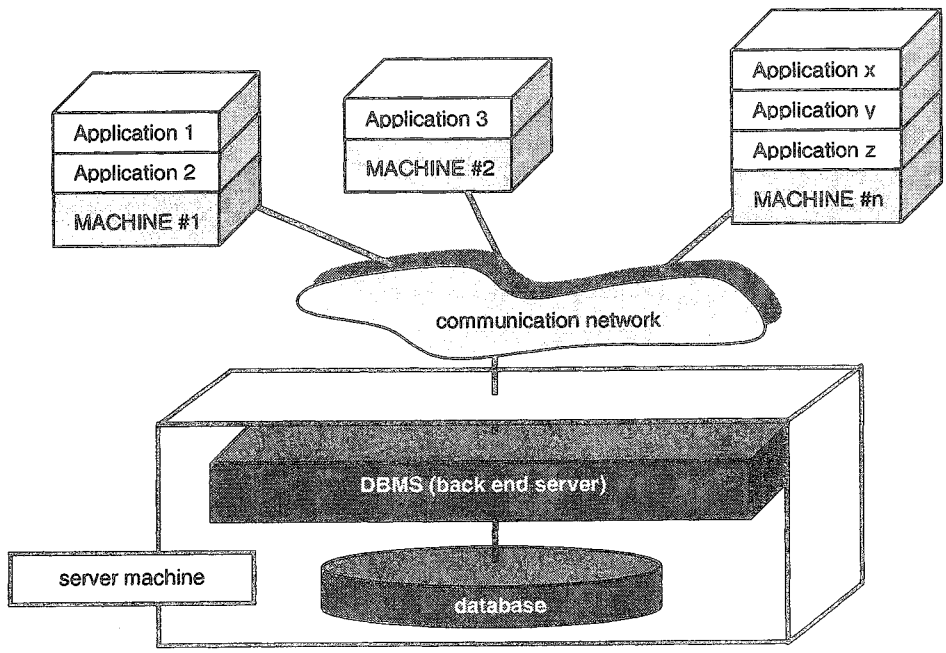

Figure 16: The client server architecture

The communication between an application and the DBMS is accomplished through the link between the client machine and the server machine, via a communication network.

Distributed database systems can be developed according to the client server architecture. Another possible organisation is to have the database itself distributed on several machines (Figure 17).

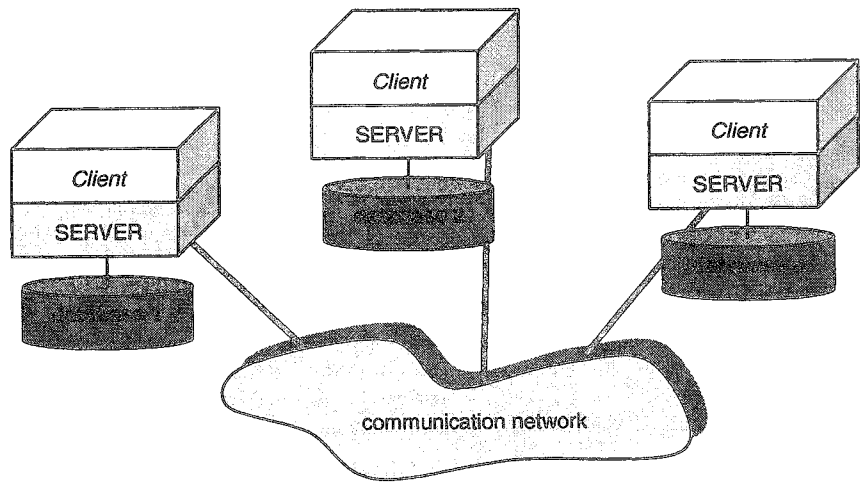

Figure 17: Distributed database system

In this architecture, each machine supports a part of the organisation's database and can become a client of the other servers (to which is linked). It is the *union* of the individual databases (i.e. the databases on each server) that constitute the database of the organisation. However, the individual databases should be possible to be exploited independently, as if they were not linked to each other. A whole chapter is dedicated to this issue in the volume 2 of this study guide

Data models

Physical data independence is one of the main advantages of database systems. This is achieved based on the conceptual level. Users work with data (both in terms of definition and manipulation) at the conceptual level, whilst the DBMS "takes care" of the physical details. We have also mentioned that at the conceptual level data is described purely in terms of its intrinsic characteristics – i.e. its logical structure. We have not answered the question "How is the logical structure of data described?" And, further, "How is data manipulated at the logical level?"

The answer is: "Data is modelled at the logical / abstract level by means (or within) a modelling theory". We are going to look at the relational theory (also called the *relational model*[1]) as the most used data modelling theory for commercial database systems. The relational theory consists of:

- a set of *concepts* (i.e. relational data objects) by means of which data (corresponding to a real life system) is modelled;
- a set of *operators* by means of which the objects of the model are manipulated;
- a set of *rules* that specify how the concepts and operators are allowed to be put together.

Figure 18 illustrates a parallel between data modelling within the relational theory and "modelling" a car by means of a set of Lego pieces.

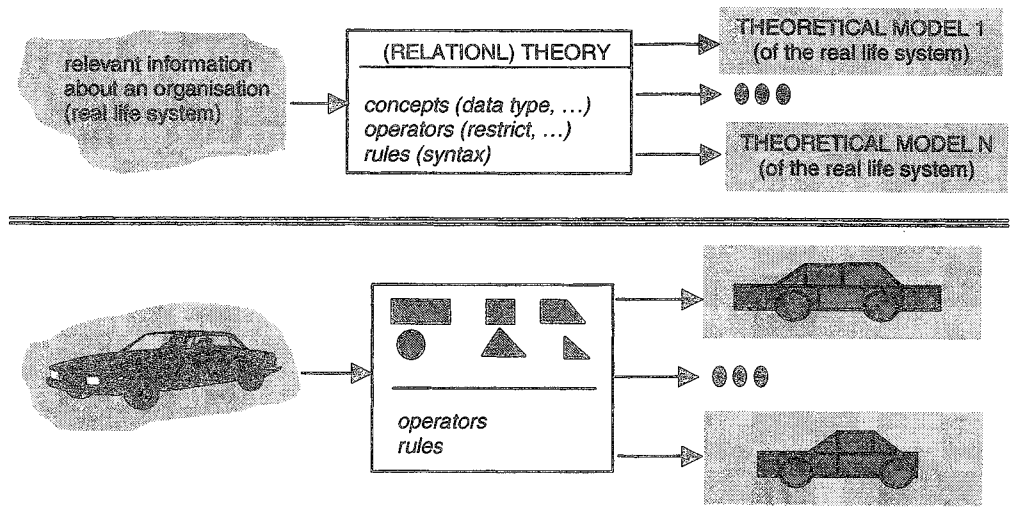

Figure 18: Modelling within a theory

The concepts of the relational theory correspond to Lego pieces. Examples of Lego pieces: rectangle, square, trapezoid, disc, triangle. Examples of relational concepts[2]: data type and table. Lego pieces can be used to model real life objects. Data types and tables can be used to model information about real life systems (e.g. organisations). A car was modelled by means of two rectangles, two squares, two discs, one trapezoid and one triangle. The information about a person can be modelled, within one table with three columns: one entitled "Name", of type "string of characters", one entitled "Age" of type "number" and one entitled "Address" of type "string of characters". By choosing another set of Lego pieces, the same car can be modelled differently. A different

[1] In order to make clear the distinction between a *model* of a real world system and the *theory* based on which the model was built, the term "relational theory" is used in this section to denote the theory. However, in the following chapters, we are going to use the term "relational model" to denote both the theory and a model; the specific meaning (theory or model of a system) will result from the context.

[2] These concepts are formally introduced in the next chapter.

relational model can be devised for "person" too; for instance, a table with only two columns, "Name" and "Date of birth".

The models (represented by one or more tables) can be manipulated by means of relational operators. For instance, there exists a relational operator, "restrict", that can be applied to the above table in order to restrict it to contain only people younger than 35.

Activity: Identify some "operators" corresponding to the Lego example.

Finally, there are the rules. The relational model can, for instance, have a rule that says "each column in a table must have an associated data type". A rule in the Lego example could be "every disc must be attached to a rectangle".

It is not sufficient to only devise a good theoretical model. The data model, eventually, has to be implemented. There are situations when a good theoretical model is all that is needed. However, the general aim in information systems is to automate or provide effective tools for information processing. Therefore, the requirement of a data model to be implementable comes as a consequence.

A software system that supports the relational theory is a relational DBMS. The implementation of an abstract data model in a DBMS is a database system (see Figure 19). As a matter of fact, none of the extant commercial DBMSs fully implement a theory. They usually impose certain restrictions and, accordingly, implement only a part of the theory. Therefore, certain abstract data models would have to be "adjusted" before they can be implemented.

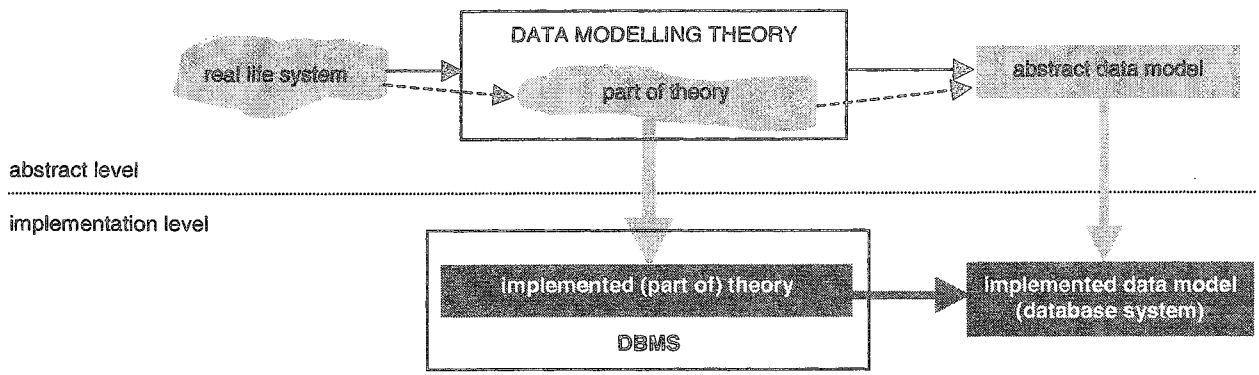

Figure 19: Data modelling and database development

The relational model is not the only data modelling theory. Two other well known models are: the *hierarchical model* and the *network model*. Compared to them, the relational model is relatively new. However, it has established itself rapidly as the "standard" theory for data modelling and, nowadays, almost all commercial systems implement it (in one form or another).

Nevertheless, new theories, that address some of the relational model's drawbacks, gradually become more and more popular. Amongst them, the *Object Oriented Model*, the *Extended Relational* (or Object Relational) *Model* and the *Deductive* (or Logic) *Model* are the most representative. They are discussed in the last chapter of the second volume of this study guide.

Learning outcomes

On completion of this chapter you should be able to:

- discuss the limitations of the file-based approach;
- describe the way the database approach overcomes the limitations of the file based approach;

- define and explain what is it meant by a *database system* and a *database management system (DBMS)*;
- describe the three level ANSI/SPARC architecture of a database system;
- discuss the schemas and mappings corresponding to the three level ANSI/SPARC architecture;
- discuss the concept of "data independence";
- explain the role of each of the components of a database system - data, hardware, software and people;
- present the most important features of a DBMS;
- discuss the advantages and disadvantages of database systems;
- discuss possible approaches to distributed database systems;
- explain what a data modelling theory (or data model) is and define its place within the context of database systems.

References

ANSI (American National Standards Institute), 1975. ANSI/X3/SPARC Study Group Data Base Management Systems. Interim Report, FDT. ACM SIGMOD Bulletin, 7(2).