

## 1. Distributed Processing

Distributed databases result from combining two main concepts: databases and computer networks. Databases introduce *data independence*, meaning independent data administration (as opposed to previous efforts where each application managed its own data). This was achieved by centralizing access to data and, consequently, facilitate their control. Computer networks, however, promote the opposite of centralization by linking different nodes (e.g., computers) that can be spread over a (potentially) large geographical area. The two concepts may seem contradictory, but it is important to recognize that they are not: the main objective of a database is integration, not centralization. Even though DBMSs have traditionally achieved integration through centralization, and the two concepts have come to be closely related, note that one concept does not demand the other.

Distributed database systems, like centralized ones, seek to provide data integration. However, distributed systems handle this issue from a distributed point of view. In other words, data are no longer assumed to be stored at a single node.

In the following sections, we introduce the fundamentals of distributed processing and how they specifically apply to distributed databases, highlighting both benefits and drawbacks. Next, we present a tentative classification of distributed DBMSs and a reference architecture. Subsequent sections focus on the two approaches for implementing distributed database systems: namely, top-down (where the distributed system is designed from scratch) and bottom-up (where the distributed system is built on top of existing databases).

Traditionally, it has been hard to define distributed processing. For example, the difference between distributed systems and some kinds of parallel systems is rather vague. A precise definition of a distributed computing system is the following:

“A distributed computing system is a number of autonomous processing elements (referred to as a computing device that can execute a program on its own), not necessarily homogeneous, that are interconnected by a computer network and that cooperate in performing their assigned tasks.”

Tamer Özsu, M.; Valduriez, P. (2011). *Principles of Distributed Database Systems*. New York: Springer. (pg. 2).

First of all, we must clearly state *what* can be distributed when autonomous elements cooperate to perform an assigned task. Four main things can be distributed: the processing logic, functionality, data and control. The processing logic (in other words, the code execution) is assumed to be distributed according to the above definition. For example, a query could be split up in small pieces and be executed by different elements (nodes) in parallel. If functionality is distributed, it means that some functions are delegated to some elements. For example, some elements might answer queries and others might store data. If data are distributed, some elements are responsible for storing certain pieces of data. For example, one node might store product data and another might store customer data. Finally, we can also distribute task execution control. For example, we can have one node controlling the other nodes' tasks (e.g., one node is responsible for ensuring consistency when gathering the pieces of the output produced by certain elements), or the node may distribute the tasks. Note that most combinations also make sense, as for example, a system where some nodes are responsible for storing data and others for controlling the execution of distributed queries.

Distributed processing has proved to be helpful in three main scenarios. Firstly, distributed processing is a natural fit for today's organizations, which are geographically distributed all over the world. More and more applications do follow this model, as can be easily perceived by considering web-based applications, e-commerce, news-on-demand or even new paradigms such as e-science and knowledge networks. Consequently, it might even be found that a centralized system is not possible in certain scenarios. This is more and more a typical scenario in the case of certain country data privacy laws (such as the Spanish LOPD, *Ley Orgánica de Protección de Datos*), which do not allow local user data to be stored outside of the country (accordingly, e-companies operating in Spain must store data related to Spanish users in servers located in Spain). For these cases, a distributed system also happens to be more efficient and provide better performance, as data can be placed where they are needed to minimize communication overheads. In this way, distributed systems benefit from this situation, also known as *data locality*.

Secondly, distributed systems are known to be more reliable and responsive. Indeed, if one element fails or is unavailable, it does not mean that the whole system will also fail or will be unavailable (whenever the system is able to cope with element failures). For example, an e-commerce platform like Amazon cannot be down for hours, as it would translate directly to a massive loss of money.

Finally, distributed systems facilitate data sharing, and also the sustained autonomy of the different elements involved. This is important, for example, for knowledge networks. In relation to this, scalability and support for ad hoc

**Note**

In the context of distributed databases, element and node are used interchangeably to refer to the different elements that are interconnected and cooperating in the distributed database; such usage will be seen throughout this chapter.

**Beyond the Exabyte**

As of today, we can already find the first databases storing more than 1 Exabyte of data, mainly in the e-science field. Furthermore, the first company databases storing a Petabyte of data are also a reality. In 2009, Facebook and Yahoo! announced the first Petabyte-scale data warehouses.

growth needs are also easier than with centralized systems. This is an important issue, for example, for start-ups and companies that expect a potentially large growth ratio.

Although all the above is true, the main need for distributed processing today lies in the current role played by very large databases. Indeed, distributed processing is understood as the best way to handle large-scale data management problems, being considered as a direct application of the divide-and-conquer approach. If the necessary software and hardware needed for a very large application are beyond current capabilities of centralized systems, distributed systems can provide the solution by breaking up the problem in small logic pieces, and working in a distributed manner. Importantly, this is the principle behind well-known concepts such as grid computing (adopted by big companies such as IBM or Oracle) or the growing *cloud data management* concept. Indeed, consider the following definition of cloud computing:

“Cloud computing encompasses on demand, reliable services provided over the Internet with easy access to virtually infinite computing, storage and networking resources.”

Tamer Özsu, M.; Valduriez, P. (2011). *Principles of Distributed Database Systems*. New York: Springer. (pg. 744).

In order to virtually provide *infinite* resources, cloud computing relies on distributed processing. Specifically, the big challenge behind cloud data management is to be able to deal with large numbers of distributed resources over the network, all of which together provide this virtually infinite access to resources. In essence, this is the very same problem that classic distributed systems face. The difference, though, is that cloud computing functions at a larger scale and is service-oriented. Thus, specific data storage, data management and parallel data processing techniques have been developed, most of them beyond relational theory. Such systems are beyond the scope of this module, where we will focus on classical, mostly relational or relational-like solutions.

In today's world, the main need for distributed processing as compared to centralized systems is to be able to manage very large data repositories.

### 1.1. Distributed Database Systems

Distributed database systems appeared as a solution to the increasing need to efficiently handle and administer large data repositories:

#### Grid Computing

The primary relational database vendors adopted grid computing as a distributed solution for managing very large databases. For example, the *g* in Oracle 10g and 11g versions stands for grid.

#### Grid Computing vs. Cloud Computing

There is much controversy today about the boundaries between these two concepts. Although there is no clear consensus, one may say that the two concepts tackle the same problem from different perspectives. Grid computing is used within an organization (and is therefore more reliable and secure because it uses the organization's resources), whereas cloud computing, in principle, has a public connotation (i.e., it is provided as an outsourced service through someone else's resources) and therefore follows more of a service-oriented paradigm.

#### Service-Oriented Paradigm

The new service-oriented paradigm claims to outsource certain complex tasks of the organization in a flexible, elastic and adaptive manner. Examples are infrastructure (network or hardware), platform (software packages that might help to develop applications, such as DBMSs), software (user-ready applications) or even business logics, which the provider administers and the user just exploits.

“A distributed database (DDB from now on) is a collection of multiple, logically interrelated databases (known as nodes or sites) distributed over a computer network. A distributed database management system (DDBMS from now on) is thus, the software system that permits the management of the distributed database and makes the distribution transparent to the users.”

Tamer Özsu, M.; Valduriez, P. (2011). *Principles of Distributed Database Systems*. New York: Springer. (pg. 3)

**Note**

The term DDBMS is often incorrectly used to refer both to the distributed database and the distributed database management system.

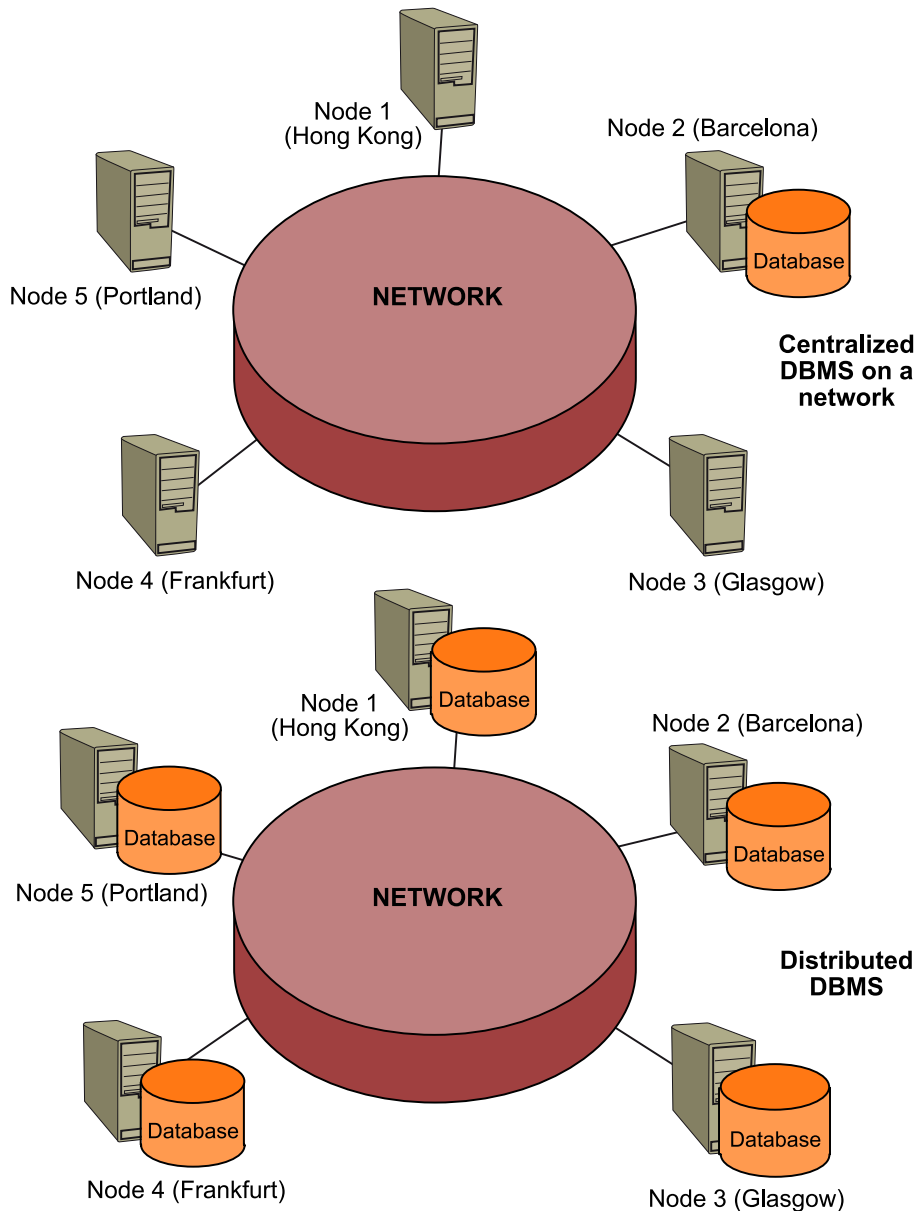
Pay special attention to three main concepts in this definition: “logically interrelated”, “distributed over a computer network” and “distribution is transparent to the user”:

- The quality of being logically interrelated stresses the fact that, like a DB, a DDB is more than a simple collection of files. Files should be somehow structured and a common access interface should be provided. For this reason, the physical distribution of data (e.g., data partitioning and replication) does matter, and it becomes one of the most conflictive aspects of DDBMS.
- In close connection with the previous point, note that data may be distributed over large geographical areas; however, there may also be the case where the distributed data are in the very same room. The only constraint imposed is that the communication between nodes is done through a computer network instead of shared memory or disk space. Note that this excludes parallel databases, since they do not meet this requirement.
- Another common mistake is to think that, despite communicating through a computer network, the database resides in only one node. If this were so, it would not be very different from a centralized database and would not pose new challenges. On the contrary, DDBMSs are a different matter, providing an environment where data are distributed among a number of sites (see Fig. 1).
- Finally, making distribution transparent to the user is, indeed, a huge accomplishment with many implications. Transparency refers to separation of a system’s higher-level semantics from lower-level implementation issues. Thus, the system must hide the implementation details. For example, the user must be able to execute distributed queries without knowing where data are physically stored; hence, data distribution and replication must be an internal issue for the DDBMS. Similarly, the DDBMS must ensure safety properties at every moment. Examples are the ACID transaction properties, which are obviously affected by distribution; dealing with update transactions, which must guarantee data consistency when replication happens (i.e., synchronization between copies); and coping with node failures to guarantee system availability.

**ACID properties**

ACID stands for the Atomicity, Consistency, Isolation and Durability properties of transactions in a DBMS.

Figure 1. Illustration of a central database on a network and a real DDBMS environment.



## 1.2. Characteristics of Distributed Database Systems

In the previous section we have briefly discussed the main implications of the DDBMS definition. In the next two sections, we thoroughly discuss all these implications. First, we focus on the traditional benefits attributed to DDBMSs, which have been summarized as three main fundamentals: distribution transparency, improved performance and easier system expansion.

### 1.2.1. Distribution Transparency

To hide distribution from users, a DDBMS must guarantee data independence and transparency with regard to the network, fragmentation and replication. To better introduce all these transparency levels, we will begin with an example to illustrate the concepts involved. Consider a company working in 5 dif-

ferent cities: Hong Kong, Portland, Barcelona, Glasgow and Frankfurt. They run IT projects in each city and keep a database about their employees, projects and certain other data as follows:

### Running example

employee(id, name, certificate, birthDate, nationality)

project(pno, name, city, country, budget, category, productivityRatio, income)

salary(certificate, amount)

assigned(employeeId, projectNo, position, duration)

The two first refer to information about employees and projects, whereas the third records the salary to be paid according to the employee's academic certificate (therefore, *employee(certificate)* is a foreign key to *salary(certificate)*). The final item keeps track of which employees are assigned to each project. Thus, it has a composite primary key (*employeeID* and *projectNo*, both foreign keys to the corresponding source table –or relation<sup>1</sup>– attributes). This is a traditional example where a company naturally distributes their business model over a large geographical area (we assume each project is only held in a given city). For this reason, it seems reasonable that each city (node) only keeps data about its employees, projects and assignments. Accordingly, in Barcelona they will only store those tuples related to employees and projects undertaken in that city. This is known as *fragmentation*. Furthermore, there may be some tables that are duplicated at other sites. For example, the salary table seems a good candidate to be stored at every node. This is known as *replication*.

<sup>(1)</sup>Note we will use *table* and *relation* interchangeably throughout this module.

Now, let us introduce the different transparency levels a DDBMS should provide:

- **Data independence:** This is fundamental to any form of transparency, and is also common to centralized DBMSs. Basically, data definition occurs at two different levels: logical (schema definition) and physical. Logical data independence refers to user applications' immunity to changes in the logical structure (i.e., the schema) of the database, whereas physical data independence, on the other hand, hides storage details from the user.
- **Network transparency:** In a centralized database, the only resource to be shielded is data. In a DDBMS, however, there is a second resource to be likewise protected: the network. Preferably, the user should be protected from the operation details of the network, even hiding its existence whenever possible. Two kinds of transparency are identified at this level: *location* and *naming* transparency. The former underlines the fact that any task performed is independent of both the location and system where the operation must be performed. The latter refers to the fact that each object has a unique name in the database. In the absence of this, the user is required to embed the location name with the object name. Location transparency is needed in order to have naming transparency. In our example, nam-

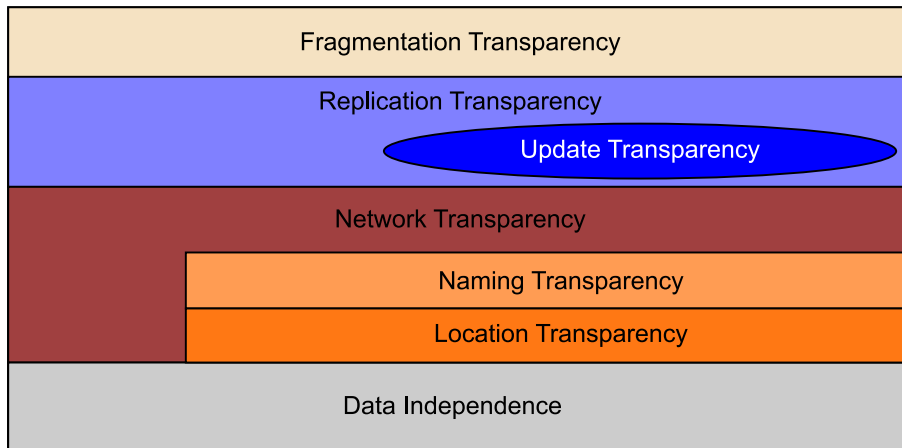
ing transparency means that we can refer to the *employee* table, instead of *Barcelona:employee* (which refers to the fragment in Barcelona).

- **Replication transparency:** As briefly discussed at the beginning of this section, it may be advantageous to replicate data over different nodes. The main reason to do so is performance, since we may avoid network overhead by accessing data locally (i.e., replication increases locality of references) and may use replicas to perform certain actions in a more efficient way, using other copies. Furthermore, it also provides robustness: if one node fails, we still have the other copies to access. However, the more we replicate, the more difficult it becomes to deal with **update transactions** and to keep all replicas consistent. Ideally, all these issues should be transparent to the users, and they should act as if a single copy of data were available (thus, as if one object were available instead of many). In our example, if the *salary* table is replicated at each node, the DDBMS itself will be responsible for maintaining consistency between the replicas and for choosing which replicas to use for each action. All these issues are dealt with in detail in section 4.
- **Fragmentation transparency:** In our example we discussed the usefulness of having different fragments for each relation. Thus, each fragment would be a different object. Again, the main reason for fragmentation is reliability, availability and performance, where it can be seen as a way to diminish the negative aspects of replication. When relations are fragmented, we face the problem of handling queries over relations to be executed over relation fragments. This typically entails a translation from the *global query* into *fragment queries*. If this transparency level is provided, the translation is performed by the DDBMS. For example, if we want to see all the employees in our supposed company, the user will query all the tuples in the employee relation. The DDBMS will then be responsible for breaking this global query into fragment queries to be posed at each node, and to compose the global result from the partial results obtained. To do so, the DDBMS must know the criteria used to fragment the relation, but this will be further explained in section 4.

#### Update transparency

The update transparency level refers to whether synchronization of replicas falls to the DBA (database administrator) or is automatically performed by the system. It is important to note that, unlike other transparency levels, update transparency mainly affects DBAs.

Figure 2. Dependency between the different levels of transparency.



It is important to note that all these transparency levels are incremental, as depicted in figure 2.

Distribution transparency implies fragmentation, replication, and network transparency, as well as data independence.

From the user point of view, full distribution transparency (also known as query transparency) is obviously appealing, but in practice, it is well-known that full transparency is hard to achieve:

“Applications coded with transparent access to geographically distributed databases have: poor manageability, poor modularity and poor message performance.” Gray, J. (1989).

*Transparency in its place - the case against transparent access to geographically distributed data.* Cupertino: Technical Report TR89.1, Tandem Computers, Inc. (pg. 11).

In short, this statement claims that full transparency makes the management of distributed data very difficult (in the sense that many bottlenecks are introduced, see section 1.3. for a better understanding of this claim). For this reason, it is widely accepted at this time that data independence and network transparency are a must, but replication and/or fragmentation transparency might be relaxed to boost performance.

### 1.2.2. Improved Performance

This claim is closely related to data locality. Consider the two scenarios depicted in figure 1. If a single centralized database is available over the network, every other node accessing the data must do so through the network. This does not happen in the second scenario, where:

- Each node handles its own portion of the database, decreasing the competition (in the sense of blocking) for CPU and I/O operations.



- Localization reduces the number of remote accesses to data, which in turn, implies less propagation delays.

This is the reason why most DDBMSs are designed to maximize data locality and reduce resource contention and communication overhead. As discussed in section 4, data locality can only be obtained by means of replication and fragmentation.

Besides data locality, DDBMSs can also benefit from both inter-query and intra-query parallelism. Inter-query parallelism is the ability to execute multiple queries at the same time, while intra-query is achieved by breaking up each single query and executing its pieces in parallel at different sites, accessing different parts of the distributed database. Distributed query processing, however, falls outside the objectives of this module.

These issues help explain why DDBMSs are considered today to be a solution for managing large-scale data repositories, since data locality (i.e., replication and fragmentation), together with parallelism, are seen as an efficient way to put a divide-and-conquer approach into practice.

### 1.2.3. Easier System Expansion

In a DDBMS, it is much easier to accommodate increasing database sizes. Indeed, expansion can be dealt by simply adding new processing and storage capacity to the network. This has many implications, but we focus on two of them. Firstly, it is much cheaper to run a bunch of *smaller* interconnected computers than the corresponding centralized version. Secondly, adding new nodes (e.g., computers), rather than adding new hardware or resources in a centralized system, brings other benefits such as not needing to stop the system in order to expand it. This feature is important for the so-called 24/7 systems (always available).

Together with the previous point, this makes it much easier to elastically increase the dimensions of our system to improve performance, and to apply the divide-and-conquer approach previously discussed.

## 1.3. Challenges Associated with Distributed Database Systems

The previous section discusses the main benefits of DDBMS. However, there is a price to pay for such nice properties. In this section, we focus on the challenges that a DDBMS must overcome. Briefly, they can be reduced to design issues that arise when building our DDBMS.

**1) Designing the Distributed Database:** We have already discussed that placement of data is important to exploit data locality and parallelism and to minimize the communication overhead. For these purposes, data replication and fragmentation play a crucial role. Indeed, we can decide to fully replicate our

#### Computer networks

Despite availability of high-speed and high-capacity networks, network latency is still an issue that recommends against moving large amounts of data around distributed environments, in favor of exploiting the data locality. For example, in some satellite connections, latency can be extended by up to 1 second.

#### Cloud computing

This principle is exploited to the limit in cloud computing, which is able to provide a seemingly infinite amount of CPU and storage facilities.

database or to not replicate it at all, or we can achieve any desired degree of replication in between. Furthermore, we also must decide how to fragment the database and where to place these fragments. Unfortunately, finding an answer to such problems demands expensive algorithms, as this is an NP-hard problem in nature. Consequently in most real-life cases we must rely on heuristics that lead to partial optimums.

A closely related problem is to decide how to manage the distributed catalog. A catalog contains information (e.g., descriptions and locations) about data items in the database. Where to place the catalog is a similar question as where to place data. We can opt for either a global catalog or one local to each site. It can be distributed or centralized, and also replicated or not replicated.

**2) Distributed Query Processing:** The principles behind query processing are basically the same as for a centralized DBMS and thus are cost-based. However, in this case we must deal with additional parameters such as distribution of data (fragments and replicas), communication costs and lack of sufficient, locally available information for estimating costs of alternative execution plans. Unfortunately, this is an NP-hard problem in essence, leading again to the use of heuristics in most real cases.

**3) Distributed Concurrency Control:** Concurrency control entails the synchronization of accesses to the distributed resources, so that database integrity is preserved. This problem is somewhat different from the centralized case, as we must also consider the consistency of several copies of the same data. Optimistic and pessimistic solutions can be used to address this topic.

**4) Reliability of DDBMSs:** Two of the main characteristics of a DDBS are availability and reliability. However, they do not come for free. The DDBMS is responsible for keeping DDB integrity even when some sites fail or become unavailable. If the whole system were to fail, the DDBMS is also responsible for starting up the system and reestablishing a consistent state.

**5) Security Issues:** Security is an important factor in any DBMS. In DDBMSs, security issues must be revisited by considering the presence of a network.

In this module, we will focus on the design issue (either bottom-up or top-down), which is thoroughly addressed in sections 4 and 5.

## 2. Brief History and (Tentative) Classification of Distributed DBMSs

Distributed DBMSs started in the early 1970s as research projects, and did not reach an acceptable level of maturity until the end of the 1970s. System R and Ingres were the pioneer DDBMSs and were developed in parallel. The initial research on this topic was mainly motivated by the need to manage data for large (and thus, naturally distributed) organizations. These systems assumed slow communications between nodes and the DBA needed to instruct the DBMS where to place data. However, the user could query data in a transparent way (i.e., he or she was provided with query transparency). The first commercial systems, though, did not hit the market until the 1980s and they did not really succeed until the late 1990s, with the arrival of fast wide area networks. Until then, networks were too slow to really exploit a DDBMS system. Thus, during those years, DDBMSs were set aside and most efforts were dedicated to parallel databases, which benefited from computer clusters and local area networks.

With faster networks, geographically distributed databases were developed, but it was not until the arrival of the Internet that DDBMSs emerged as first-class citizens. The Internet, however, brought many other challenges, such as the need to integrate data from many pre-existing databases. As a consequence, heterogeneous databases blossomed with the arrival of the new century. It is significant that traditionally distributed databases assume a single DDBMS and a single logical database, whereas heterogeneous DDBMSs combine several autonomous databases using different DBMSs and different schemas.

The foundations of distributed database management systems were already established back in the 1970s. However, we had to wait until fast wide area networks appeared and, especially for the Internet breakthrough, in order to observe the real impact of DDBMSs on the market.

There are many different ways to classify DDBMSs, but a traditional classification focuses on the degree of autonomy and heterogeneity of each node in the system. In this classification, systems can be classified as mainly homogeneous or heterogeneous distributed systems:

### Note

Obviously, large organizations in the 70s were several orders of magnitude smaller than current ones. However, the amount of available memory was also considerably smaller and networks were substantially slower, which essentially brought up the same problem about how to efficiently distribute data (maximizing data locality).

**1) Homogeneous DDBMSs:** Regular distributed databases are homogeneous and one distributed DBMS manages all data. Distributed database design involves creating the schema in a top-down fashion as for a conventional central database. Furthermore, the DBA is responsible for specifying how data should be distributed over nodes. Consequently, nodes have no autonomy at all.

**2) Heterogeneous DDBMSs:** Heterogeneous systems, however, must deal with the inherent heterogeneity of the pre-existing nodes, and they are usually designed in a bottom-up fashion. Designing heterogeneous distributed databases implies dealing with data integration and heterogeneities of every kind, as we must overcome the fact that the same or similar data may be represented in different ways in each participating database. In addition, note that nodes in a DDBMS might also act as isolate databases in addition to their participant role in the DDBMS.

Although the distinction between heterogeneous and homogeneous DDBMSs has come to be accepted, there is much controversy about how to classify heterogeneous DDBMSs. In this module we provide a tentative classification based on how the existing databases are interconnected and how they are intended to cooperate in order to build the heterogeneous DDBMS. This classification thus depends on how these databases are coupled and mainly refers to the negotiation process carried out among the pre-existing nodes in order to build up the DDBMS. A strict cooperation protocol results in less autonomy for the DDBMS nodes (e.g., they cannot decide to rollback a transaction if that is against the cooperation agreement), whereas a relaxed cooperation protocol results in more autonomy within the DDBMS. Accordingly, the classification presented below elaborates on the degree of autonomy provided.

Tightly coupled federated databases are the most restrictive with regard to autonomy. This approach is meant to tightly interconnect the nodes, and it therefore reduces their autonomy. As presented below, this degree of cooperation is achieved by providing a common global integration schema. At the opposite end, we talk about multi-databases, which provide a fair degree of autonomy. Multi-databases rely on a very weak interconnection between nodes and no global integration schema is provided. Between these two extremes there are a myriad of possibilities in the middle ground, where partial integration schemas are provided. These kinds of systems are known as loosely coupled federated databases. Note that any solution providing either a partial or global integration schema is known as a federated database (loosely or tightly coupled, respectively), whereas multi-databases do not provide integration schema at all. From here on we will focus on the role played by the integration schema (if any):

- Tightly coupled federated databases require the definition of a global integration schema that contains mappings to all participating database schemas. The federated database becomes a central server management system on top of the participating autonomous databases. Note that such

a low degree of autonomy can make entering or leaving the federation become problematic for the participants.

- As the number of databases to integrate increases, it becomes very difficult or impossible to define a global integration schema over the large number of autonomous databases. Multi-databases provide no global conceptual schema and instead a *multi-database* query language allows specification of queries that search through many participating databases. Consequently, a node can easily enter or leave the DDBMS.
- Loosely coupled federated databases provide a middle ground between a single integration schema (tightly coupled federated databases) and no schema at all (multi-databases). Instead, the designer can define views that combine and reconcile data from different data sources. You can think of these views as similar to relational views. Such views, however, require a query language that can express queries over several databases, i.e., a multi-database query language. Thus, on one hand they provide a certain degree of integration (views) and, on the other hand, they use an ad hoc query language to overcome heterogeneities.

Table 1: Comparison of distributed databases

	<b>Autonomy</b>	<b>Central Schema</b>	<b>Query Transparency</b>	<b>Update Transparency</b>
<i>Homogeneous DBs</i>	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>
<i>Tightly Coupled Federated DBs</i>	<i>Low</i>	<i>Yes</i>	<i>Yes</i>	<i>Limited</i>
<i>Loosely Coupled Federated DBs</i>	<i>Medium</i>	<i>No</i>	<i>Yes</i>	<i>Limited</i>
<i>Multi-databases</i>	<i>High</i>	<i>No</i>	<i>No</i>	<i>No</i>

Table 1 elaborates on the discussion initiated above:

- **Autonomy:** This column focuses on the autonomy of nodes in the DDBMS. As previously discussed, nodes in homogeneous databases are not autonomous by definition, whereas heterogeneous solutions are built on top of (more or less) autonomous databases.
- **Central Schema:** This column focuses on the presence of a central schema in the system. Homogeneous databases and tightly coupled federated databases rely on central schemas to tackle the integration issue, whereas the rest of heterogeneous solutions relax this constraint to deal with large numbers of nodes.
- **Query Transparency:** Query transparency refers to whether distribution of data is reflected in how users pose queries –in other words, if, from the user perspective, it gives the impression of a single database. Query transparency is primarily achieved when a single schema is provided: as views

in loosely coupled federated databases, as a global schema for tightly coupled federated databases and as a single logical schema for homogeneous distributed databases.

- **Update Transparency:** This transparency level only affects DBAs; it refers to how updates are processed internally and if distribution is taken into account.

In general, it is difficult to achieve consistent updating for heterogeneous databases since the nodes are autonomous and the integration layer may not have access to the local transaction managers. However, tightly and loosely coupled federated databases may partially achieve it. In the first case, some updates in the global integration schema might be propagated to the underlying databases. In the second case, they define views over data, and the problem is reduced to the update through views problem (and thus, limited to it).

Note this is not an exhaustive classification of distributed systems. The new generation of distributed systems, those adopted in solutions such as cloud computing, would hardly fit in these definitions. On the contrary, this classification aims at producing a clear taxonomy about classical, mainly relational, solutions.

Distributed database management systems are usually divided into homogeneous and heterogeneous systems. Homogeneous DDBMSs were naturally designed from scratch to be distributed, whereas heterogeneous DDBMSs integrate in a distributed fashion several heterogeneous, autonomous and preexisting sources.

In what follows, the reader can find a detailed discussion of the main challenges that have arisen for both homogeneous and heterogeneous databases. First, section 3 discusses reference architectures for the different types of distributed systems we just discussed in this section. Later, we focus on how to design homogeneous databases (section 4), and finally we elaborate on data integration issues and a more detailed discussion of heterogeneous distributed databases (section 5).

### 3. Distributed DBMS Architectures

In this section we discuss system architectures to handle distributed DBMSs. Using the classification introduced in the previous section, we distinguish between architectures for homogeneous and heterogeneous systems.

#### 3.1. Architectures for Homogeneous DDBMSs

We introduce this discussion at two different levels. First, we elaborate on how to extend ANSI/SPARC schema architecture for distributed database systems. Then, we will focus on a generic component architecture for centralized DBMSs and how to extend it to cope with the challenges posed by DDBMSs.

It is important not to confuse a schema architecture with a component architecture. While the ANSI/SPARC schema architecture focuses on user classes and roles and how they view/access data, the latter focuses mainly on the functional layers and components of a DBMS.

#### ANSI/SPARC Architecture

Issued in 1977 by the American National Standards Institute (ANSI), this architecture was intended to be a reference that shows which elements and interfaces must be implemented and discussed with regard to DBMSs.

##### 3.1.1. The ANSI/SPARC Schema Architecture

Figure 3. The ANSI/SPARC Schema Architecture

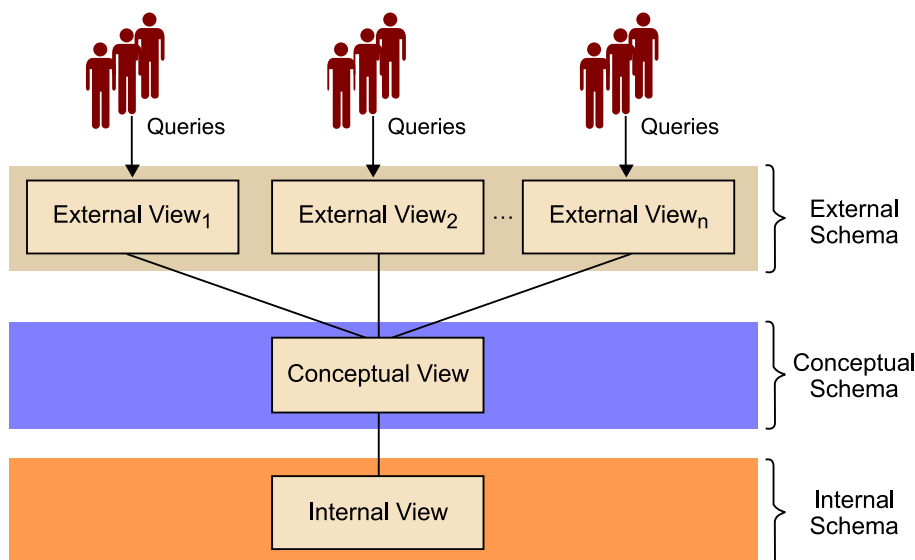


Figure 3 outlines the ANSI/SPARC architecture. Three views<sup>2</sup> of data are depicted. The external view shows how users see the database. It can be understood as a *window* over the database (i.e., they can only access and see that portion of the database); it is adapted to the needs of each kind of user, taking into consideration only data and relationships of interest to them. Several users may share the same view of the database, and the collection of different user views makes up the external schema. Next, we find the conceptual schema, which captures how the real world is conceived by the organization. At this level, the

<sup>(2)</sup>View, in this sense, must be understood as a way to show or see data according to a given schema.

universe of discourse needed for day-to-day operations is modeled in a conceptual view. The conceptual schema, representing the organization's view of the world, is what interests developers who will model their applications on top of that. Finally, the internal view deals with the physical definition and organization of data. Thus, it represents the system view, which is responsible for locating data on different storage devices and using the corresponding access structures to retrieve data. The internal schema is what interests the DBA who will want to tune it up to boost performance.

For example, re-consider schema 2 as our database conceptual schema (previously introduced as a running example in section 1.2.). From this schema we can define several external views, for example, one for the general manager and another for project leaders. The first one would contain all the relations contained in the conceptual schema, whereas the second one would only contain *employee*, *project* and *assigned* relations. The reason is that project leaders are supposed to focus on managing projects, whereas the manager will also be interested in other orthogonal aspects such as salaries. Besides these two schemas, each relation in the conceptual schema will be physically implemented and the DBA is responsible for creating indexes, clusters or any other physical structure to improve data access. This is what we know as the database internal schema.

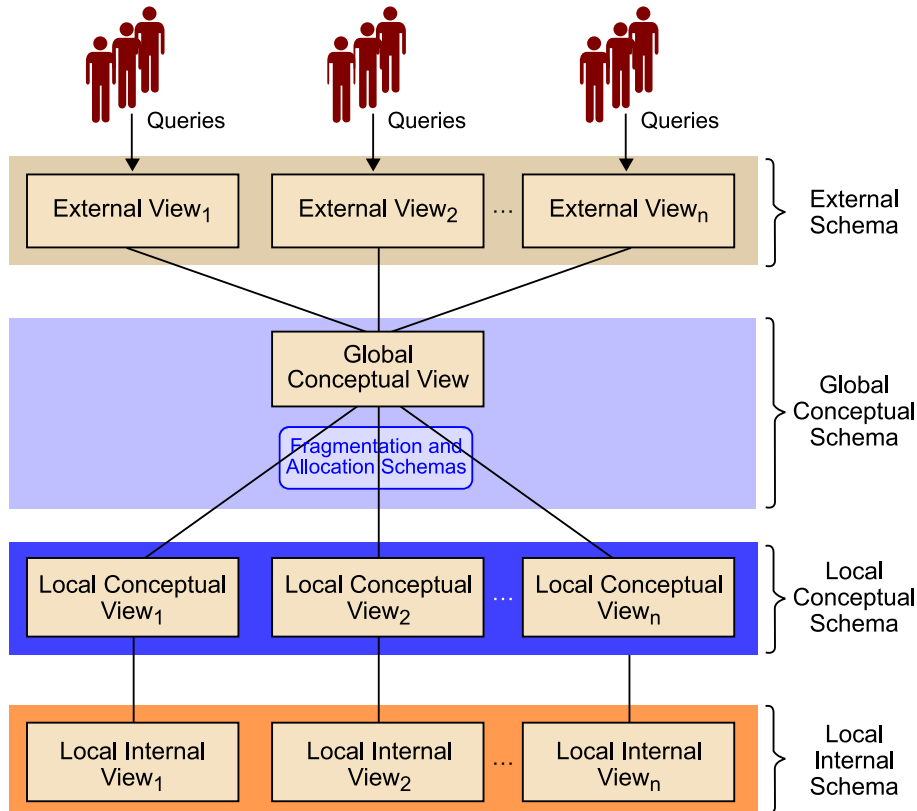
These schemas are related by means of mappings which specify how a definition at one level can be obtained from a definition at another level. These three levels are the basis for data independence; the separation between the external and conceptual schemas provide logical data independence, whereas the separation between the conceptual and internal schemas provide physical data independence. Such mappings are stored in the database catalog.

However, ANSI/SPARC does not consider distribution, and must be adapted in order to provide distribution transparency. As shown in figure 4, in the presence of distribution, a global conceptual schema is needed in order to work with a single logical database (i.e., the conceptual view of the organization). This conceptual schema corresponds to the very same idea behind the original ANSI/SPARC architecture, but the crucial difference is that the database is composed of several nodes (instead of just one) and this general schema provides for data independence as well as network, replication and fragmentation independence (thus providing distribution transparency). Furthermore, every node also has one local conceptual schema and one internal schema. Again, there must be mappings between the external schema and the global conceptual schema, and between the global and local conceptual schemas, in order to translate a global definition in a set of local definitions. Such mappings are stored in the global catalog. Out of all these, the specific mappings between the global and local conceptual views are known as *fragmentation* and *allocation* schemas.



Finally, note that in this extended architecture we can talk about the global unique view (which hides distribution and where each data object has a unique name), and local views (which are aware of naming and distribution).

Figure 4. Extending the ANSI/SPARC Schema Architecture for DDBMSs



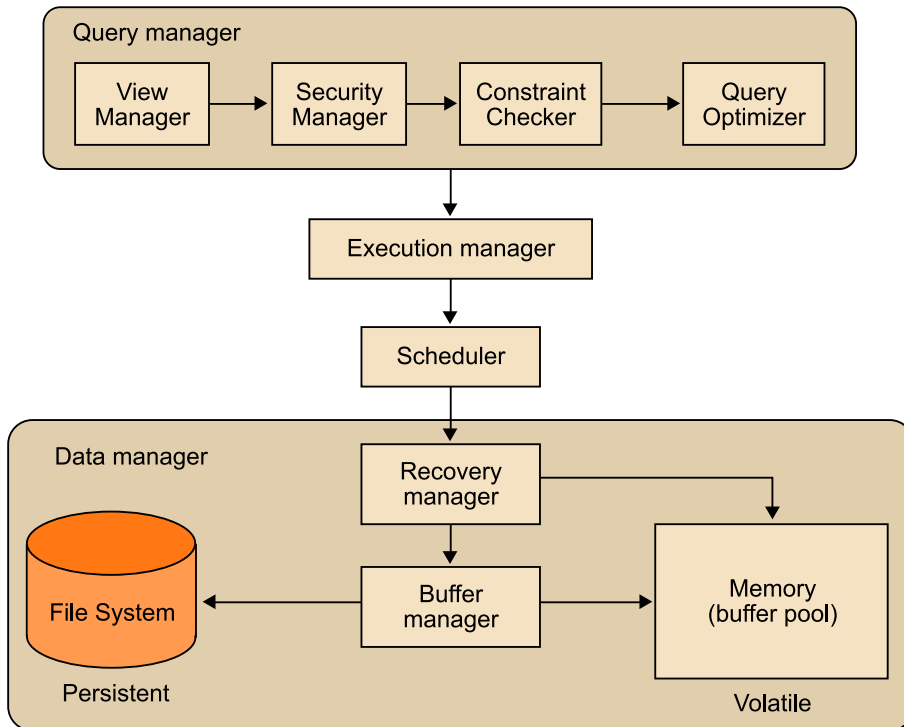
#### Note

Be aware that local catalogs are still needed to solve mappings between the local conceptual schema and the internal schema.

### 3.1.2. Extending a Centralized DBMS Functional Architecture

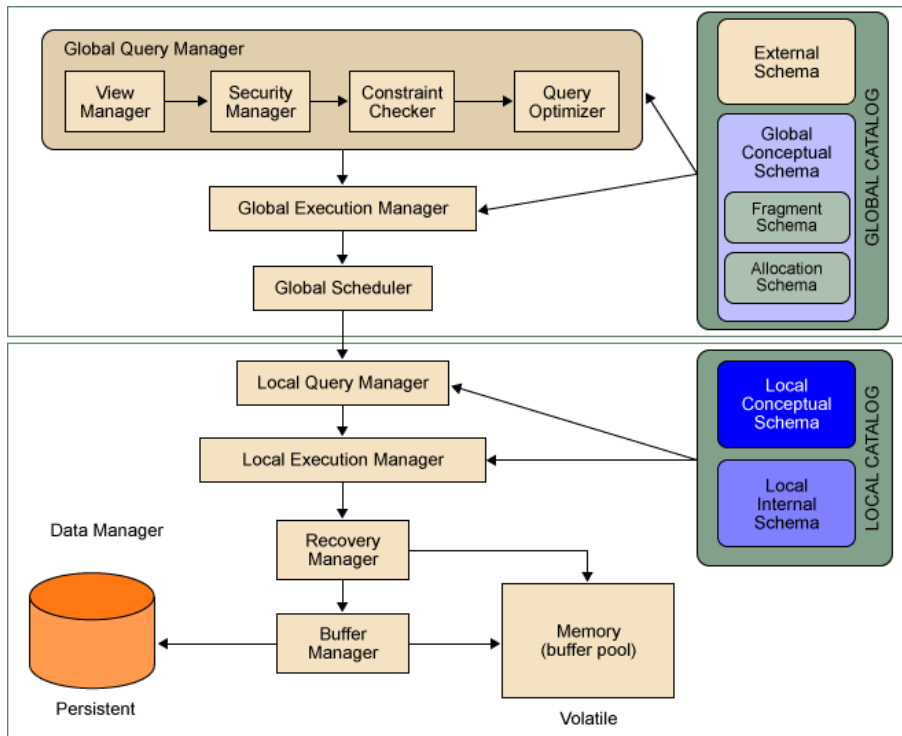
A DDBMS is made up of a set of components to successfully manage the logical database. When running on a computer, DBMSs interact with applications through their interface layer at the highest level of abstraction, whereas they communicate at the lowest level of abstraction with the operating system, through their communication layer. In between, a large number of components interact to form the DBMS as a whole. Figure 5 outlines the main components of a centralized DBMS.

Figure 5. Modular Architecture of a Centralized DBMS



Users issue queries over the database through the interface layer and, subsequently, these queries reach the query manager component. The query manager contains the view manager (a query is posed according to the external schema in ANSI/SPARC architecture, and the view manager must rewrite the input query in terms of the conceptual schema), the security manager (responsible for performing the corresponding authorization checks), the constraint checker (responsible for guaranteeing that integrity constraints are preserved) and the query's semantic, syntactic and physical optimizers, responsible for, respectively, performing semantic optimizations (i.e., transforming the input query into an equivalent one of a lower cost), generating the syntactic tree (in terms of relational algebra operations) and the eventual physical access plan. Next, the execution manager decides where to execute what (in case more than one CPU is available) and in which order (for example, a subquery that must be solved prior to solving the outer query). Then, for each executed operation, it is also responsible for passing the result to the next operation. The scheduler deals with the problem of keeping the database in a consistent state even when concurrent accesses and failures occur. In short, it preserves the consistency and isolation properties of transactions (C and I of the ACID acronym). It sits on top of the recovery manager that is responsible for preserving the atomicity and durability properties of transactions (A and D of the ACID acronym). The recovery manager, in turn, sits on top of the buffer manager, responsible for bringing data to the main memory from secondary storage. Thus, the buffer manager communicates with the operating system.

Figure 6. Modular Architecture of a DDBMS



In a DDBMS there are several refinements to be done. As depicted in figure 6, there are two well-differentiated stages. The first one corresponds to modules cooperating at the global level, as discussed by the ANSI/SPARC architecture, whereas the second one corresponds to modules cooperating at the local level. In the former, the data flow is transformed and mapped to the lower layers by dealing with a single view of the database (once the external schemas are resolved). In the latter, distribution transparency is no longer provided and the system must deal with distribution, replication and partitioning. More specifically:

- The global query manager contains the view manager, the security manager, the constraint checker and the query's semantic and syntactic optimizers. All of these behave as in a centralized DBMS, except the syntactic optimizer that is extended to consider data location (by querying the global data catalog). Finally, the physical optimizer is replaced, at the global level, by the global execution manager. This new extension is responsible for exploiting the metadata stored in the global catalog, for deciding issues such as which node executes what (according to replicas and fragments available and communication cost over the network) and for certain execution strategies that are relevant for distributed query execution, such as minimizing the size of intermediate results to be sent over the network (e.g., deciding among join strategies) and exploiting parallelism. Finally, it inserts communication primitives in the execution plan.
- The global scheduler then receives the global execution plan produced in the previous module and distributes tasks between the available sites. It will be responsible for building up the final result from all the subqueries

that were executed in a distributed context. The global scheduler also applies distributed algorithms to guarantee the ACID transaction properties.

- Next, each node receives its local execution plan and, by querying its catalog, generates a local access plan by means of the local query manager. The local query manager behaves similarly to a centralized query manager and, among other duties, decides which data structures must be used to optimize data retrieval. Subsequently, the data flow goes through the recovery and buffer managers, as in a centralized DBMS.

### 3.2. Architectures for Heterogeneous DDBMSs

Among the various heterogeneous DDB systems, we can find different levels of coupling, from more loosely coupled systems to more tightly coupled ones, according to the classification proposed by Sheth and Larson (1990), based on the level of autonomy of the component DBs. In loosely coupled systems, each component DB handles system input and output, as well as the data schema and data exchange with the other DBs, while maintaining its own autonomy. By contrast, tightly coupled systems require a negotiation, in detriment of their own autonomy, to achieve a global schema for all DDB users, in addition to a functional architecture to deal with global accesses.

Tightly coupled systems typically adopt a five-level schema architecture allowing a global schema to be obtained for the heterogeneous DDB. During the operation of heterogeneous DDBs, it is fairly common to use a functional architecture based on mediators and wrappers, with the advantage that this permits different degrees of coupling. Moreover, modern peer-to-peer systems are now in use, and a reference functional architecture for these is also available. Since peer-to-peer systems allow different degrees of coupling, it would seem reasonable to implement more tightly coupled systems with mediators and wrappers, while more loosely coupled systems would be implemented using peer-to-peer systems. The following sections will cover the three types of architecture we have identified here.

A.P. Sheth; J.A. Larson (1990). "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases". *ACM Computing Surveys* (Vol. 22, No. 3, September).

#### 3.2.1. Five-Level Schema Architecture

The reference schema architecture for tightly coupled heterogeneous DDBs is based on the ANSI/SPARC three-level schema architecture (see Figure 3), comparable to the architecture for homogeneous DDBs (see Figure 4).

Given the need to create a heterogeneous DDB from different heterogeneous DBs, the process used is known as bottom-up design. Instead of creating a logical design from scratch, as in the case of homogeneous DDBs, it begins with the local designs of the DBs that will form the new DDB.

Figure 7. Tightly coupled heterogeneous DDB reference architecture

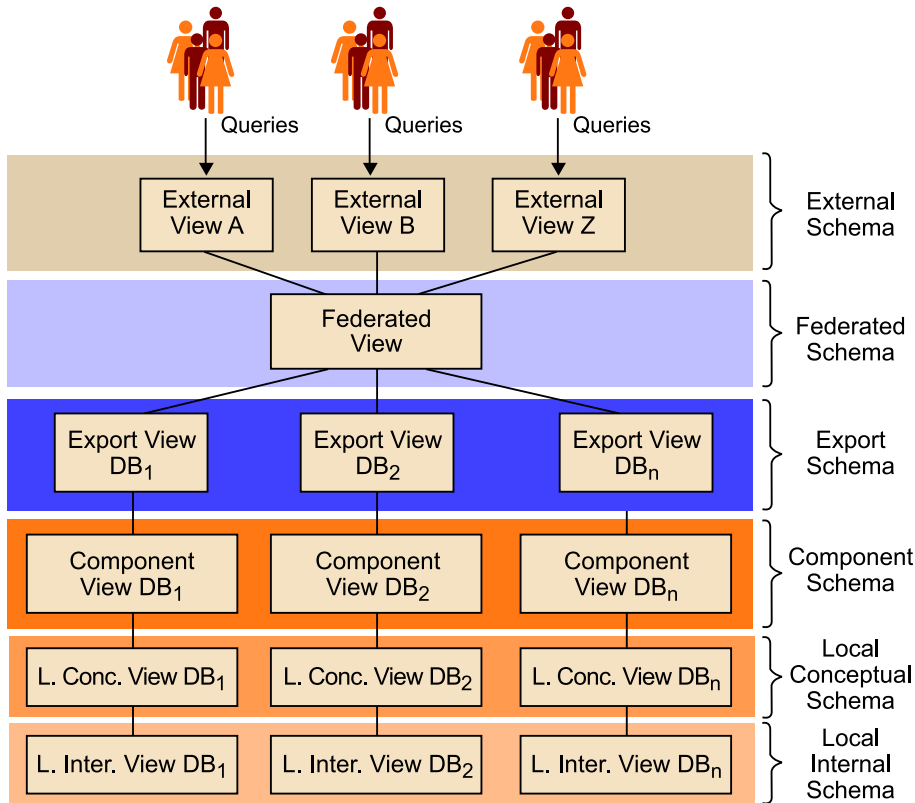


Figure 7 shows the internal schema and the conceptual schema – as the local internal schema and local conceptual schema – of each component DB. The first additional schema level, compared to the DDB architecture shown in Figure 4, is the level of the component schema. The purpose of this schema level is to solve the syntactic heterogeneity of the data model of the component DBs. Just as the conceptual schema is expressed as being dependent on the local DBMS, the component schema is expressed with the canonical data model chosen as the data model of the heterogeneous system. This schema level presents all local schemas expressed according to the same data model.

The second additional schema level, the export schema level, defines the subset of data from the component database that will be shared through the heterogeneous DDB. Like the component schema, the export schema is also expressed in terms of the canonical data model.

The purpose of the federated schema (see Figure 7), which acts as a global schema (see Figure 4), is to serve the heterogeneous DDB in resolving all possible semantic heterogeneities among the different export schemas of the component DBs. In section 5, we discuss various aspects related to heterogeneities. The federated schema is expressed in terms of the canonical data model.

As with the other schema architectures, external schemas are defined from the federated schema. Users of the heterogeneous DDB will make global access based on the external schemas. There are two options for expressing external schemas. Firstly, external schemas can be expressed through the canonical da-

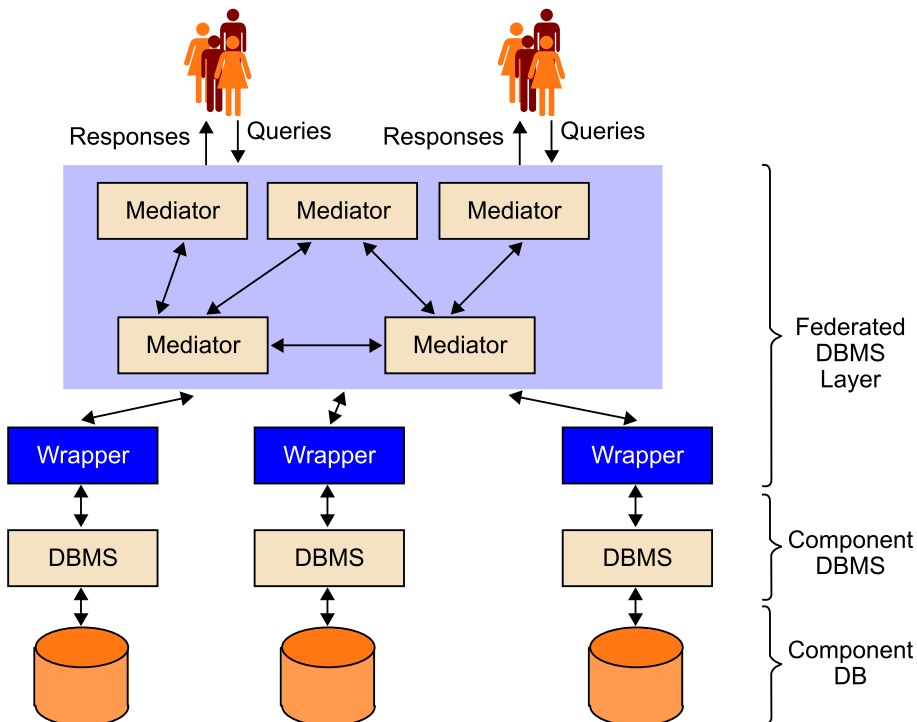
ta model of the heterogeneous system (monolingual architecture). Thus, users may need to address the system differently according to whether the access is local or global. Secondly, each external schema can be expressed according to the data model of the component DB to which the user belongs. In this case, there will be a multilingual architecture allowing the user to work in the same way both locally and globally.

The mappings that relate the elements at different levels of the reference architecture schema of heterogeneous DDBs are just as important as they are in the architecture of homogeneous DDBs.

### 3.2.2. Wrapper-Mediator Functional Architecture

From a functional perspective, the main difference between heterogeneous and homogeneous DDBs are the autonomous DBMSs that exist in each DB comprising a heterogeneous DDB. Heterogeneous DDBs are obtained from a software layer that works above the component DBMS (as shown in the functional reference architecture in Figure 8) and that enables to access to the global users to the different DBs. This layer acts as though it were just another application sending queries and receiving responses.

Figure 8. Wrapper-mediator functional architecture



The most popular implementation of functional architecture for heterogeneous DDBs is based on mediators and wrappers, as shown in Figure 8.

In Wiederhold (1992), a mediator is defined as “a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications.”

Each mediator performs a given function through a clearly defined interface. Thus, the heterogeneous DDBMS layer is implemented through a mediator – or through a hierarchy of these – that deals with the access requests of heterogeneous DDB users, covering all the functionality of the global manager.

Mediators typically work using a data model and a common interface language. Thus, wrappers are used to resolve the potential heterogeneities arising from the different component DBMSs. Each wrapper is obtained from a mapping between a view of a component DBMS and the view of the mediator. We will look more closely at mediators and wrappers in section 5.

### 3.2.3. Peer-to-Peer Functional Architecture

Over time, several types of peer-to-peer (P2P) systems have appeared, all with different aims. In this section, we will deal with modern P2P systems, which focus on data exchange and are characterized by:

- Mass distribution, since they are composed of thousands of nodes (peers) with a wide geographical distribution and have the possibility of forming groups in certain places.
- The inherent heterogeneity of the peers and their total individual autonomy.
- System volatility, since each peer is usually a personal computer that enters and leaves the system at will, making data management more complex.

To ensure the data management features of these systems, we must deal with data location, query processing, data integration and data consistency.

Figure 9 shows a functional reference architecture for a peer participating in a P2P data exchange system<sup>3</sup>. The most important point about the proposed architecture is the separation of its functionality into three main components: an interface used to send access requests, a data management layer that processes queries and information from the metadata catalog and a P2P infrastructure, which comprises the sub-layer of the P2P network and the P2P network *per se*. Depending on the functionality of the P2P system, one or more of these components may not exist, or they may be combined or implemented in specialized peers.

#### Bibliography

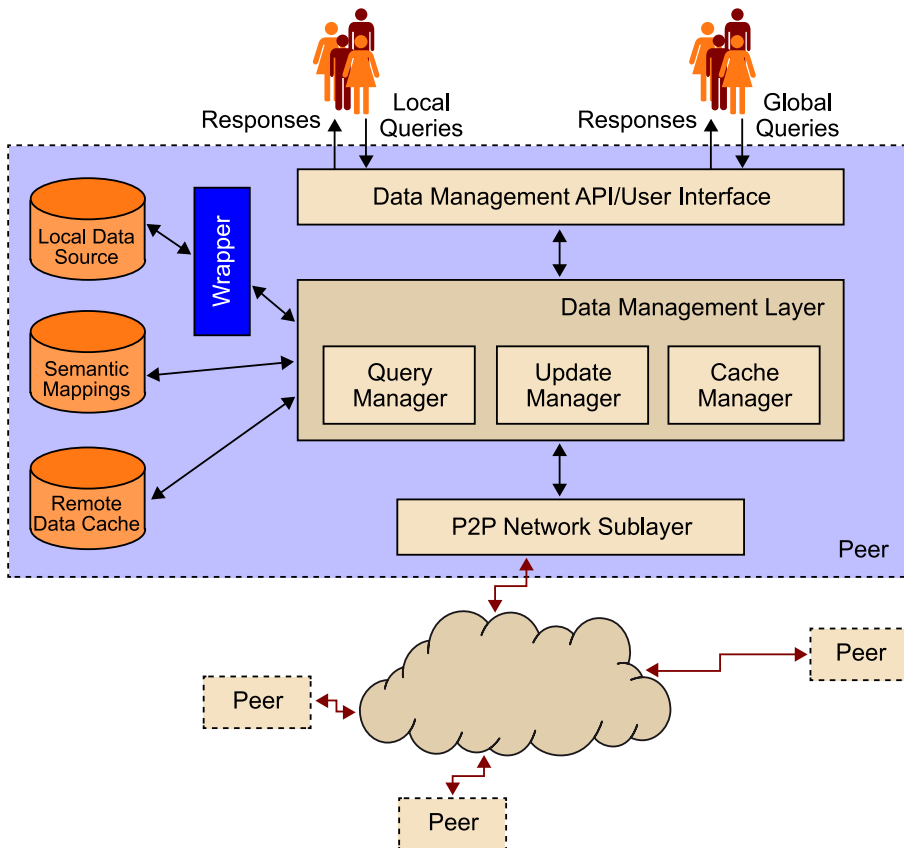
Wiederhold, G. (1992). Mediators in the architecture of future information systems. *Computer*, 25(3):38–49.

<sup>(3)</sup>Two examples of data exchange P2P systems are Gnutella <<http://www.gnutelliums.com>> and Kazaa <<http://www.kazaa.com>>.

Access queries, both those referring to local data and those referring to global data, are transmitted through the user interface or through a data management API<sup>4</sup>, and sent to the data management layer. The layer that receives the query has a query manager responsible for its execution.

<sup>(4)</sup>An API is an Application Programming Interface, whose function is to allow the different software components to communicate with one another.

Figure 9. Peer reference architecture



All the tasks required for the query manager to execute the query depend on the system heterogeneity. If the peers are heterogeneous, the query manager must check the semantic mappings in order to identify the peers of the system storing relevant data to the query resolution. The query must then be reformulated in order each involved peer can understand the query. To communicate with the peers, the query manager invokes the services implemented by the P2P network sublayer. Given that some P2P systems have specialized peers for storing semantic mappings, the peer receiving the access query must contact the specialized peer in order to execute the query.

Moreover, if all peers in the P2P system had the same data schema, the query reformulation functionality would not be required, nor would it be necessary to store the semantic mappings.



Regardless of whether execution is coordinated by the peer receiving the query or a specialized node, when the peer receiving the query receives the responses from the set of involved peers, it has the option of storing the results locally using the cache manager, in order to speed up the execution of similar queries in the future.

The query manager is also responsible for executing the local part of each global query generated by another peer. In this case, the existence of a wrapper can resolve heterogeneities. In addition, when a data update is required, the update manager is responsible for coordinating the update among the peers that contain replicas of the data to be changed.

The P2P network infrastructure, regardless of how it is implemented, provides communication services to the data management layer. Note that the execution of queries is sensitive to the implementation of this infrastructure because it is an overlay network above a physical network (typically Internet). This overlay network can be pure (all peers are equal) or hybrid (some peers have special features). The set of peers can form an unstructured topology (there is no restriction on data location in the peers) or a structured topology (the data location is controlled to obtain greater scalability, compromising autonomy). The key to query execution is to see how the resources are indexed and how they will be searched. In unstructured topologies, indexes stored centrally or decentrally have been used. Structured topologies typically use a dynamic hash table, whereby applying the hash to the key, which is the identifier of an object, generates the identifier of the peer where data associated to the object are stored.

## 4. Distributed Database Design

In this section we focus on distributed database design from scratch (i.e., a top-down design). Thus, given a database and its workload, we look at how to allocate data in different sites so that we optimize a certain set of parameters (normally, minimizing resource consumption for query processing). To do so, two main issues must be addressed: how to fragment data and where to allocate them. The two issues are strongly interrelated.

### 4.1. Data Fragmentation

Data fragmentation deals with the problem of breaking relations into smaller pieces and thus, decreasing the working unit in the distributed system. There are many reasons to fragment, but the main idea is that a relation is not a good processing unit. For example, consider the case of a user accessing a database through a view defined as a selection over a relation  $R$ . Certainly, this user will never access any data instance out of this subset. In general, applications and users will only access a subset of the relations available in the database or, more specifically, a certain subset of some relations.

Without considering data fragmentation we have two options: either placing the relation at a single node (and thus, increasing the remote accesses and producing a potential bottleneck) or replicating it at every node where the relation might be needed. However, different subsets are naturally needed at different nodes and it makes complete sense, from a performance point of view, to collocate those fragments likely to be used jointly. Remember, this is what is known as data locality. As a result, the communication overhead through the network is minimized and we avoid unnecessary replication. Consider our running example (section 1.2.1). If we want to take advantage of data locality, we will be interested in fragmenting data and placing it at the corresponding node. As previously discussed, our example organization is geographically distributed and we are only interested in keeping data about projects and employees at the corresponding node (either Hong Kong, Barcelona, Portland, Frankfurt or Glasgow, see figure 1). Thus, out of all the projects, Barcelona's node will store projects undertaken in that city (or better put, coordinated from that city), as well as the assignments and employees working in these projects. The same will be true for each of the other nodes. In this case, the `project` relation should be fragmented according to the `city` attribute and according to its value, placing the tuple in one node or another. As for the `employee` and `assigned` relations, note that the fragmentation must be performed in a slightly different way. Each `employee` and `assigned` tuple will be collocated with its corresponding `project` tuple (thus, we should first fragment the `project` relation).

#### Fragmentation Vs. Replication

It is important to clearly distinguish between fragmenting relations and replicating fragments. While the first step relates to the problem of finding the proper working unit for the distributed system, the second relates to the allocation problem, and will be addressed later.

Although benefiting from data locality has been the traditional reason for fragmentation, another advantage has also appeared. Decomposing a relation into fragments allows different transactions to be executed concurrently and also increases system throughput. Importantly, when fragments are placed in different nodes, both of these aspects facilitate parallelism, which today has become the main approach for dealing with very large databases, as discussed in section 1.

Notwithstanding, fragmentation also entails certain difficulties. On the one hand, fragmenting a relation may lead to poorer performance when multiple fragments must be retrieved and manipulated at query time. This situation might happen when there are conflicting requirements that make it impossible to separate mutually exclusive fragments. On the other hand, integrity checking can become costly if two attributes with a dependency are split across fragments (even in some simple cases, depending on where fragments are allocated). For example, suppose that in our running example, the organization headquarters are in Hong Kong. From that location, they would need to periodically retrieve data about all the employees. According to our previous discussion, employees are spread across all the nodes according to what project they are currently working on. Consider the following query: `identify employees who have been working in at least three different cities`. We should retrieve all `employee`, `assigned` and `project` fragments to one node (suppose that the data is shipped to the node issuing the query; i.e., Hong Kong), then reconstruct the original relations and join them properly in order to answer this query. For this case, fragmentation is clearly a drawback; it is affecting the query answer time, since we are incurring in communication costs between nodes. It would seem that integrity checking could also become more complicated. Consider an employee `e` who has been living in Barcelona and Glasgow. Currently, he or she is working at project `pB` in Barcelona, but he or she was enlisted in project `pG` when working in Glasgow. Something as common as checking the foreign key relationship between `assigned` and `employee` becomes more difficult, because data about employee `e` is now placed in Barcelona, whereas the assignment tuple `e`, `pG` is in Glasgow. Again, we need to incur in communication overhead between nodes to check such constraints.

#### Fragmentation Vs. Partitioning

Partitioning refers to a relation that is broken into small pieces but they are not distributed over a network. Normally, this is done in order to benefit from parallelism and it is typical of parallel systems, but it can also be used to implement privacy.

#### Note

Minimizing distributed joins is a crucial aspect for distributed query processing.

Traditionally, data fragmentation has been useful to reflect the fact that applications and users might only be interested in accessing different subsets of the whole schema or even different subsets of the same relation. More recently, it has also been used in dealing with large databases, since fragmentation increases the degree of concurrency and facilitates parallelism. As a negative aspect, it might also increase distributed joins and make semantic data control more difficult.

Fragmenting a table is as simple as finding alternatives to decompose it into smaller tables. Clearly, we have two main approaches: horizontal and vertical fragmentation. In the first case, a selection predicate is used to create different fragments and, according to an attribute value, place each tuple in the corresponding fragment. For example, the `project` fragmentation proposed in our running example suits this category. There, the predicates to be used would be `city = NAME_CITY` (where `NAME_CITY` is each of the `city` attribute values: Barcelona, Hong Kong, Portland, Glasgow or Frankfurt). Each predicate will result in a fragment.

By contrast, in case of vertical fragmentation, different projections of the relation are carried out, each of them forming a different fragment. For example, consider once again the `project` relation. Suppose now that we want to store the `name`, `city` and `country` attributes at each node, whereas the other data will be stored at the Hong Kong head office. In this case, a vertical fragmentation of the relation could be carried out as follows:

$\Pi_{\text{project}}(\text{pno}, \text{name}, \text{city}, \text{country})$  and

$\Pi_{\text{project}}(\text{pno}, \text{budget}, \text{category}, \text{productivityRatio}, \text{income})$ .

The need to place the primary key at each vertical fragment will be properly justified later. But note that different requirements may lead to different fragmentation strategies over the same table.

As usual, a third alternative combines both options, and is known as hybrid fragmentation. Hybrid fragmentation is nothing other than nesting horizontal and vertical fragmentation strategies. In other words, further fragmenting fragments produced by a previous fragmentation strategy. For example, consider a combination of the two requirements discussed above: we want to geographically distribute data about projects, but only those attributes of interest at each node (the others will be stored in Hong Kong as discussed previously). Thus, we could perform a vertical fragmentation, namely  $\text{VF}_1 (\Pi_{\text{project}}(\text{pno}, \text{name}, \text{city}, \text{country}))$  and  $\text{VF}_2 (\Pi_{\text{project}}(\text{pno}, \text{budget}, \text{category}, \text{productivityRatio}, \text{income}))$ .  $\text{VF}_2$  will be placed at the Hong Kong node, whereas  $\text{VF}_1$  will be further fragmented by applying the horizontal fragmentation discussed earlier (i.e., `city = NAME_CITY`). As result, five fragments will be produced, each of them to be placed at the corresponding node.

In this document, the following subsections focus on horizontal and vertical fragmentation and discuss two main aspects: the degree of fragmentation we may achieve, and fragmentation rules for correctness. The first aspect considers to what extent fragmentation of a relation is desirable (in other words,

how many subsets we want to produce); the second aspect seeks to ensure the semantic correctness of the fragments. The latter is achieved by means of three properties:

- **Completeness:** Given a relation  $R$  and a set of fragments, any data item (either a tuple or a set of attributes) of  $R$  can be found in at least one fragment. Thus, data are not lost when fragmenting.
- **Disjointness:** Given a relation  $R$  and a set of fragments, any data item placed in a fragment cannot be found in any of the other fragments. Remember that data replication must be considered a posteriori, in the allocation stage, as a task independent of data fragmentation.
- **Reconstruction:** Given a relation  $R$  and a set of fragments, the original relation can always be reconstructed from the fragments by means of relational algebraic operators.

A fragmentation is correct if we can guarantee completeness, reconstruction and disjointness.

Hybrid fragmentation is not discussed further, because it corresponds to a nested combination of the two strategies, and is therefore correct if all the subsequent fragmentation strategies applied are correct.

#### 4.1.1. Horizontal Fragmentation

Horizontal fragmentation partitions a relation along its tuples. The way to define each fragment is by means of predicates (i.e., selections over any relation attribute). Table 2 illustrates a horizontal fragmentation<sup>5</sup> for the `project` relation in our running example. Here, each fragment contains a subset of the tuples of the relation.

<sup>(5)</sup>Horizontal fragmentation is also known as primary horizontal fragmentation.

Table 2. Horizontal Fragmentation

pno	name	city	country	budget	category	productivity Ratio	income
1	p1	Barcelona	Spain	40000	administration	0.1	3000
2	p2	Barcelona	Spain	10000	administration	0.5	36000
3	p3	Barcelona	Spain	5000	financial	0.8	1000
4	p4	Frankfurt	Germany	100000	tv	0.8	70000
5	p5	Glasgow	UK	450000	financial	0.6	240000
6	p6	Glasgow	UK	2000	financial	0.6	1000
7	p7	Glasgow	UK	1000	financial	0.3	2000
8	p8	Portland	USA	30000	culture	0.2	2000
9	p9	Hong Kong	China	7000	others	0.1	10000
10	p10	Hong Kong	China	10000	financial	0.9	40000

Formally, a relation  $R$  is horizontally fragmented in  $n$  fragments by means of a *fragmentation predicate*  $R_i = \sigma_{F_i}$ ,  $1 \leq i \leq n$  where  $F_i$  is the fragmentation predicate that defines fragment  $R_i$ . Typically, we represent each fragment by the predi-

cate used to create it. In our example:  $HF_1$ : city = Barcelona,  $HF_2$ : city = Hong Kong,  $HF_3$ : city = Portland,  $HF_4$ : city = Frankfurt and  $HF_5$ : city = Glasgow.

The first issue to address is when a DBA should horizontally fragment a given relation. In general, a distributed system benefits from horizontal fragmentation when it needs to mirror geographically distributed data (each node mainly accesses data related to itself), to facilitate recovery and parallelism, to reduce the depth of indexes (since each fragment has its own index, the number of indexes increases but their size is reduced) and to reduce contention. In our example, each fragment is obviously smaller than the whole relation (provided we know there is at least one tuple at each node). Thus, an index over the primary key (pno) would result in five different indexes (one per fragment), that are smaller in size (since each index will only contain entries for the tuples in that fragment). Contention<sup>6</sup> is clearly reduced, since several mutually exclusive users, who work on different nodes, can access different fragments simultaneously and no conflicts will be caused. Furthermore, queries over the whole relation can be resolved by means of parallelism.

<sup>(6)</sup> Contention occurs when we are denied access to a database resource because of conflicts with other users, normally related to concurrency control techniques.

Next, we must decide up to what extent we should fragment. Note that fragmentation can go from one extreme (no fragmentation at all) to the other (placing each tuple in a different node). Furthermore, we need to know which predicates (over which attributes) are of interest in our database. To address this issue we should check to see which predicates are used by the users (or applications) that access the database. A general rule of thumb claims that the top 20% of most active users produce 80% of the total accesses to the database. Thus, we should focus on these users to determine which predicates to consider in our analysis. For each of these predicates, we must perform the following steps:

- 1) Gather all the simple clauses used in any query predicate related to a given relation. From here on we assume that a predicate is an expression of the form  $attr \ \theta \ k$  (where  $attr$  is a relation attribute,  $\theta$  is one of the following operators  $\neq, \leq, \geq, <, >$  or  $=$  and  $k$  is a value in the domain  $attr$ ).
- 2) For each set of simple clauses over the same attribute, complete (according to clause operator semantics) by adding the missing *complementary* clauses (i.e., all the domain values for that attribute must be considered in at least one predicate).
- 3) For each set of simple clauses generated in the previous step, determine *relevant* sets of clauses. We can apply different criteria in order to find relevant sets. A typical approach would consider a set to be relevant if each clause of

this set produces a fragment to be accessed either by a user or an application. In other words, if it makes sense to use the clauses in that set to fragment the relation.

4) For all the clauses remaining from previous step, consider all the combinations among clauses over the same relation and put them in conjunctive normal form.

5) From the results of the previous step, prune out those complex predicates that are semantically meaningless. Rules based on reasoning in first-order logic can be used here to spot contradictory predicates.

6) Among the remaining predicates, select the fragmentation predicates. Finally, the designer could decide to tune up the obtained result.

#### Note

A formula in conjunctive normal form is a conjunction of clauses, where a clause is a disjunction of literals.

For example, consider again the `project` relation, which has not yet been fragmented. And consider the following to be the queries issued by the top 20% of most active users:

```
Q1: SELECT * FROM project WHERE city = 'Barcelona'
Q2: SELECT * FROM project WHERE city = 'Glasgow' AND budget > 10000
Q3: SELECT * FROM project WHERE city = 'Frankfurt'
```

Now, let us apply the previous algorithm step by step:

**Step 1:** First, we gather all clauses issued in any of these queries. Thus, we obtain: `city = 'Barcelona'`, `city = 'Glasgow'`, `city = 'Frankfurt'` and `budget > 10000`.

**Step 2:** There are two attributes involved in these clauses: `city` and `budget`. We know that the values in the `city` domain are {Barcelona, Hong Kong, Frankfurt, Glasgow and Portland}, and the `budget` value ranges from 2000 to 450000. According to the semantics of each operator, we complement the missing predicates as follows:

- For equalities, we need to add a predicate for each value in the domain. Thus, for the `city` attribute, we need to add `city = 'Portland'` and `city = 'Hong Kong'`.
- For ranges, we need to complete the range. In the case of the `budget` attribute, we need to add `budget <= 10000`.

**Step 3:** Now we need to know whether each predicate is of relevance to at least one node. Our organization is geographically distributed, so all predicates over the `city` attribute happen to be relevant. As for the `budget` attribute, we know that `budget > 10000` is of relevance at Glasgow, but suppose that `budget <= 10000` does not happen to be interesting for any of the other nodes. Thus, most nodes will not benefit from this fragmentation, and consequently, both predicates over `budget` are discarded.

**Step 4:** For the remaining predicates, we generate all the combinations. However, only predicates over the `city` attribute remain, and it does not make sense to combine them, as they are mutually exclusive (note that it is impossible for a tuple to have two different values for the same attribute). Just as an example of how to combine them, assume that some other nodes were interested in the `budget <= 10000` predicate and thus, the `budget` predicates would be present at this step. To combine them, keep in mind that the same combination should not contain two complementary predicates, so you should only combine predicates over different attributes (i.e., `city` and `budget`). As result, we would obtain ten predicates: for each `city` we will generate two predicates (one for budgets over 10000 and another for budgets below 10000). For example, `city = 'Barcelona' AND budget > 10000`, `city = 'Barcelona' AND budget <= 10000`, and similarly for the other cities.

**Step 5:** Of the complex predicates previously generated, discard any that are meaningless (from a semantical point of view). This can happen due to obvious contradictions (for example, `city = 'Barcelona' AND country = 'USA'`) or due to internal business rules. For the latter, suppose that Glasgow only runs financial projects (i.e., all the projects

in Glasgow satisfy `category = 'financial'`). Thus, if `city = 'Glasgow' AND category = 'administration'` had been generated, we would prune it at this point.

**Step 6:** In this example we would conclude that a fragmentation like the one depicted in table 2 is the most appropriate.

Once we have identified the fragmentation predicates (by using the previous algorithm or another alternative) we must guarantee their correctness. Given a set of fragmentation predicates, a horizontal fragmentation is correct if it satisfies the three properties presented in the previous section. Specifically:

- **Completeness:** The fragmentation predicates must ensure that every tuple is assigned to at least one fragment. As long as the fragmentation predicates are complete, the final fragmentation is also guaranteed to be complete.
- **Disjointness:** The fragmentation predicates must be mutually exclusive. In other words, one tuple must be placed in at most one fragment. This is usually referred as the minimality property for horizontal fragments.
- **Reconstruction:** The union of all the fragments must constitute the original relation. Thus,  $R = \bigcup R_i$ , where  $1 \leq i \leq n$ .

For example, the fragmentation strategy proposed in table 2 satisfies all of these: the fragmentation predicates are complete (as we have considered all values for the `city` attribute), disjoint (being an equality, we know that an attribute cannot take two different values) and it can be reconstructed by means of the union operator.

#### 4.1.2. Derived Horizontal Fragmentation

The previously discussed horizontal fragmentation (also known as primary horizontal fragmentation) only considers one relation at a time. However, any relation is related to other relations and it would seem that we may use such relationships intensively when querying a database schema. For example, suppose a relation  $R$  is related to a relation  $S$  through a many-to-one relationship not accepting NULL values in the  $S$ -end (implemented as a foreign key - primary key constraint). Suppose now that a frequent, well-known system transaction, whenever inserting data in  $R$ , is to first insert the corresponding tuple in  $S$ . Furthermore, suppose now that these relations are usually queried together by joining them through the primary key - foreign key relationship. Since horizontal fragmentation tries to maximize data locality, it seems rather clear that the two relations are candidates for being placed in the same node.

To apply this strategy we should identify an *owner* and a *member* relation. These role names are simply a question of notation (they try to highlight the fact that a member is a feature or a characteristic of an object and hence, somehow dependent on it), but they are meaningful because the owner decides how to



fragment the member. For example, we have already discussed that it would be interesting to fragment the `assigned` and `employee` relations according to `project`. In this case, `project` will be the owner and `assigned` would be the member, and we would join them through the `pno` (primary key) - `projectNo` (foreign key) relationship. Similarly, `assigned` would be the owner of `employee` when fragmenting it, by joining them through the `id` (primary key) - `employeeId` (foreign key) relationship.

As a general rule, derived horizontal fragmentation is of interest when the owner fragments need to be combined with member fragments through matching join keys –in other words, when the member relation is clearly dependent on the owner relation, according to the database queries. If this is the case, we proceed as follows. Let us suppose the owner relation is already fragmented in  $n$  fragments ( $S_i$ ) and we want to fragment the member relation  $R$  regarding the owner relation, by means of a relationship  $r$ . The derived horizontal fragmentation is defined as:  $R_i = R \bowtie S_i$ ,  $1 \leq i \leq n$ . Remember that  $\bowtie$  stands for a semijoin and thus, the result of this join will be those tuples in  $R$  for which there is at least one tuple in  $S_i$  with matching joining key (we are considering the joining attributes to be the attributes of  $R$  and  $S$  in  $r$ ).

Note that the owner and member relations may happen to be related by means of two or more relationships. In this case, we should apply the following criteria to decide among the available relationships:

- The fragmentation used most by users / applications (i.e., which subset is closer to what users and applications use),
- The fragmentation that maximizes parallel execution of the queries.

For example, in the first case we could consider query frequency, whereas distributed query processing and parallelism could be considered in the second case.

Finally, in order to consider a derived horizontal fragmentation to be complete and disjoint, two additional constraints must be considered with respect to those discussed for horizontal fragmentation:

- **Completeness:** The relationship used to semijoin the two relations must enforce the referential integrity constraint.
- **Disjointness:** The join attribute must be the owner's key.

Both primary horizontal fragmentation and derived horizontal fragmentation strategies aim at maximizing data locality. However, the first considers each relation per se, whereas the latter also considers the relationships between relations.

#### 4.1.3. Vertical Fragmentation

Vertical fragmentation partitions the table in smaller subsets by projecting some attributes of the relation in each fragment. Consider table 3 that illustrates a vertical fragmentation for the `project` relation in our running example. Each fragment contains a subset of the relation attributes, but notice that all of them contain the primary key. This will be justified later in this section.

Table 3. Vertical Fragmentation

pno	name	city	country	pno	budget	category	productivityRatio	income
1	p1	Barcelona	Spain	1	40000	administration	0.1	3000
2	p2	Barcelona	Spain	2	10000	administration	0.5	36000
3	p3	Barcelona	Spain	3	5000	financial	0.8	1000
4	p4	Frankfurt	Germany	4	100000	tv	0.8	70000
5	p5	Glasgow	UK	5	450000	financial	0.6	240000
6	p6	Glasgow	UK	6	2000	financial	0.6	1000
7	p7	Glasgow	UK	7	1000	financial	0.3	2000
8	p8	Portland	USA	8	30000	culture	0.2	2000
9	p9	Hong Kong	China	9	7000	others	0.1	10000
10	p10	Hong Kong	China	10	10000	financial	0.9	40000

Formally, a relation  $R$  is vertically fragmented in  $n$  fragments by means of projections:  $R_i = \pi_R(PK, A_j, \dots, A_k)$ ,  $1 \leq j, k \leq m$ .

Where  $m$  is the number of attributes in  $R$ , and  $PK$  is the primary key of  $R$ .

As in the horizontal fragmentation strategy, the first issue is to determine whether a vertical fragmentation suits our needs. Vertical fragmentation has traditionally been passed over in practice, since it often worsened insertions and update times of transactional systems (for years, the solution to any data storage problem). However, with the arrival of query-based systems, such as decisional systems (where the user is only allowed to query data), this kind of fragmentation arose as a powerful alternative for decreasing the number of attributes to be read from a table. This can be clearly seen with an extreme scenario. Consider a relation  $Z$  with 2000 attributes. On average, these attributes are stored in 4 bytes each. During our analytical tasks, we are interested in querying 1000000 tuples on average, but only 5 attributes out of the 2000 are involved in the query. For each query we read approximately  $100000 \times 2000 \times 4$  bytes ( $10^{12}$  bytes), when, in actuality, we only need to read  $100000 \times 5 \times 4$  ( $10^5$  bytes).

In general, vertical fragmentation improves the ratio of useful data read (i.e., we only read relevant attributes) and, similarly to horizontal fragmentation, it also reduces contention and facilitates recovery and parallelism. As disad-

#### Column-oriented databases

Vertical partitioning is taken to the extreme in column-oriented database management systems, which store data by column rather than by row. These systems have been shown to be extremely useful in supporting decisional systems, and today we can find successful commercial systems such as Vertica or Greenplum.

vantages, note that it increases the number of indexes (all of the same size), worsens updating/insertion time and, in principle, increases the space used by data (as the primary key is replicated at each fragment).

Deciding how to group attributes in each fragment is, moreover, not an easy issue. Today, we can benefit from well-known approaches like clustering or attribute splitting in order to group attributes likely to be read together. In this document, we focus on attribute grouping. Thus, we start by considering the whole relation and relevant queries posed on it (again, the rule of thumb can be used for this purpose) and, from these data, we compute the *affinity* between every pair of attributes. The affinity of two attributes regarding a set of queries is defined as follows:  $aff(A_i, A_j) = \sum_{k | use(q_k, A_i) \wedge use(q_k, A_j)} freq(q_k)$ .

Where  $use(q_k, A_i) = 1$  if  $q_k$  uses attribute  $A_i$  and  $freq(q_k)$  is the frequency of  $q_k$  in the system. Intuitively, this formula tells us the number of queries where these two attributes appear together, weighted by their frequency.

For example, consider the `project` relation and the following queries (frequency shown in brackets):

Q1: `SELECT SUM(budget) FROM projects WHERE category = CATEGORY_NAME` (30%)

Q2: `SELECT pno, name, city, country FROM projects` (20%)

Q3: `SELECT productivityRatio, income FROM projects WHERE budget < 10000` (10%)

Q4: `SELECT productivityRatio, income FROM projects WHERE budget > 20000 AND city = 'GLASGOW'` (1%)

Q5: `SELECT pno, country FROM projects WHERE city = CITY_NAME` (25%)

Q6: `SELECT DISTINCT(category) FROM projects WHERE income > 30000 AND budget < 20000` (14%)

The affinity between attributes can be represented in the following symmetrical matrix (note that the diagonal remains empty as it is meaningless):

	p	n	c	co	b	ca	pr	i
p	–	20	45	45	0	0	0	0
n	20	–	20	20	0	0	0	0
c	45	20	–	20	1	0	1	1
co	45	20	20	–	0	0	0	0
b	0	0	1	0	–	44	11	25
ca	0	0	0	0	44	–	0	14
pr	0	0	1	0	11	0	–	11
i	0	0	1	0	25	14	11	–

Where `pno` is abbreviated as `p`, `name` as `n`, `city` as `c`, `country` as `co`, `budget` as `b`, `category` as `ca`, `productivityRatio` as `pr` and `income` as `i`. Note that this matrix shows heavy dependencies between attributes. Indeed, by transitivity, we can decide to cluster a set of attributes with high affinity together in the same fragment. For example, `pno-city` and `pno-country` are the highest affinities. Furthermore, the `city-country`

pair also has a high affinity (20). Since none of these is related to any other attribute (except for `city`, but with very low affinity) it is quite clear these 3 attributes should be grouped together. Similarly, `budget`, `category` and `income` are closely related to each other. At this point, only `name` and `productivityRatio` have not been grouped. In the first case, it is clear that `name` should join the first group, as it has no relation to any attribute in the other group. As for `productivityRatio`, the situation is exactly the same but the other way round: high affinity with the second group and low affinity with the first.

Just as with any other fragmentation, vertical fragments produced must guarantee completeness, disjointness and reconstruction. Specifically:

- **Completeness:** The union of the projected attributes in each fragment must produce the original relation.
- **Disjointness:** A given attribute (except for the primary key) can be used to produce only one fragment. In other words, every attribute (but the primary key) appears in one and only one fragment.
- **Reconstruction:** To guarantee reconstruction, the primary key must be replicated in each fragment. Intuitively, this is needed to keep track of the original tuple and be able to reconstruct it by means of joins through the replicated *PK*. Thus,  $R = R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$ .

In our example, we have two fragments that can be represented by the set of attributes they project.  $VP_1: \{pno, name, city, country\}$  and  $VP_2: \{pno, budget, category, income, productivityRatio\}$ . This fragmentation is complete as both fragments contain all the attributes in the relation. They are also disjoint as no attribute is repeated. Finally, to guarantee reconstruction, we replicate the primary key (`pno`) in each fragment.

Summing up, horizontal fragmentation mirrors geographically distributed data and boosts data locality both for querying and inserts/updates. By contrast, vertical fragmentation better supports query-based systems, such as decisional systems, and drastically improves the ratio of useful data read (only relevant attributes are read out of the whole set of attributes of the relation).

#### 4.1.4. Fragmentation in Oracle

This section elaborates on a practical example of fragmenting relations. We use Oracle to show the kind of fragmentation strategies that are presently available:

##### 1) Horizontal fragmentation:

#### Note

The version used in this module is Oracle 10g.

Oracle allows two principal types of horizontal fragmentation, by list and by range. Both correspond to a primary horizontal fragmentation strategy. The first must be used in the presence of fragmentation predicates with equalities, while the second one allows ranges. For example, in the first case, consider the fragmentation proposed in table 2:

```
CREATE TABLE project (
  pno NUMBER(8,0) PRIMARY KEY,
  name VARCHAR(20),
  city VARCHAR(10),
  country VARCHAR(8),
  budget NUMBER(7,23),
  category VARCHAR(10),
  productivityRatio NUMBER(3,2),
  income NUMBER(7,2))
PARTITION BY LIST (city) (
  PARTITION node1 VALUES ('BARCELONA'),
  PARTITION node2 VALUES ('GLASGOW'),
  PARTITION node3 VALUES ('HONG KONG'),
  PARTITION node4 VALUES ('FRANKFURT'),
  PARTITION node5 VALUES ('PORTLAND'));
```

After the `LIST` keyword, in brackets, we must specify the fragmentation attribute, and before the `VALUES` keyword, the name of the fragment. Note that `NULL` values can also be used (but remember to drop the `'`, if so). Alternatively, if ranges must be used, the syntax looks like this:

```
CREATE TABLE project (...)
PARTITION BY RANGE (budget) (
  PARTITION node1 VALUES LESS THAN (10001),
  PARTITION node2 VALUES LESS THAN (450001));
```

In this way, values between 0 and 10000 would be stored at node1, whereas values between 10001 and 450000 would be stored at node2. Alternatively, Oracle also allows random creation of a number of fragments. In this case, a hash function is used internally to determine where each tuple must be placed:

```
CREATE TABLE project (...)
PARTITION BY HASH (pno) ( PARTITIONS 4; );
```

In this case, 4 different fragments will be created. Oracle will use the module function (%) over the `pno` value to determine where to place each tuple. It is advisable to use powers of ten to maximize dispersion. Finally, a kind of hybrid horizontal fragmentation is also allowed, but only nesting horizontal fragmentation strategies. Two options are available: range-hash and range-list strategies. Both follow the same notation. For example:

```
CREATE TABLE project (...)
PARTITION BY RANGE (budget)
SUBPARTITION BY HASH (pno) SUBPARTITIONS 4 (
  PARTITION node1 VALUES LESS THAN (10001),
  PARTITION node2 VALUES LESS THAN (450001));
```

In this example, each partition is divided into 4 subpartitions randomly (HASH); alternatively, we could choose the LIST keyword instead of HASH and specify a list of values to perform the subsequent fragmentation.

## 2) Vertical fragmentation:

Natively, Oracle only supports horizontal fragmentation (and as shown before, not even derived horizontal fragmentation). However, it is possible to perform vertical fragmentation by taking advantage of the object-relational features available in Oracle. We will assume the reader is familiar with such features.

Vertical fragmentation can be regarded as attribute grouping and can be implemented by means of *nested tables*. When a table type appears as the type of a column in a relational table, Oracle stores all of the nested table data in a single table (that can be placed at a different node). For example, consider the vertical fragmentation proposed in table 3:

```
CREATE TYPE info AS OBJECT (
  name VARCHAR(20),
  city VARCHAR(10),
  country VARCHAR(8));
CREATE TYPE analyticalItem AS OBJECT (
  budget NUMBER(7,23),
  category VARCHAR(10),
  productivityRatio NUMBER(3,2),
  income NUMBER(7,2));
CREATE TYPE generalInfo AS TABLE OF info;
CREATE TYPE analyticalInfo AS TABLE OF analyticalItems;
```

These CREATE TYPE create the objects needed to store each group of attributes (i.e., each fragment). Next, we create the table:

```
CREATE TABLE project (
  pno NUMBER(8,0) PRIMARY KEY,
  infoData generalInfo,
  analyticalData analyticalInfo);
NESTED TABLE infoData STORE AS infoDataNT;
NESTED TABLE analyticalData STORE AS analyticalDataNT;
```

This table contains two different nested tables, which can be placed at different nodes without any problem. In this way, we achieve vertical fragmentation.

## 4.2. Data Allocation

Once the database is fragmented, we must decide where to place each fragment. It should be noted that the same fragment might be placed in several nodes; thus, replication is an issue to be addressed at this point and not earlier. In general, replication must be used for reliability and efficiency of read-only queries. On the one hand, several copies guarantee that in the case one copy fails, we can still use the others. On the other hand, replication is also seen to improve data locality, allowing data to be accessed at the same node.

### Nested Tables

Object-relational features were introduced to support the object-oriented paradigm. In Oracle, as well as others, we can find nested tables, which were created to support collections.

Nevertheless, updating/inserting data in a replicated copy takes more time, and synchronizing such writings may not be trivial. As a result, consistency of copies may be affected. For this reason, the degree of replication must be a trade-off between performance and consistency.

At this point we are better able to introduce the problem of data allocation. Given a set of fragments and a set of sites on which a number of applications are running, we seek to allocate each fragment such that some optimization criterion is met (i.e., subject to certain constraints). Normally, the optimization criterion is defined along the lines of two features:

- **Minimal cost:** A function that results from computing the cost of storing each fragment  $F_i$  at a certain node  $N_i$ , the cost of querying  $F_i$  at  $N_i$ , the cost of updating each fragment  $F_i$  at all sites where it is replicated, and the cost of data communication. The allocation problem places each fragment at one or several sites in order to minimize this combined function.
- **Performance:** In this case, we aim either to optimize system response time (given a set of queries) or maximize throughput at each node.

The first question, however, is why all these notions are not considered simultaneously when dealing with the allocation problem. The reason is that this problem is known to be a NP-hard problem; the optimal solution depends on many factors, such as the location where each query originates, query processing strategies (e.g., join methods), network latency, etc. Furthermore, in a dynamic environment, the workload and access pattern may change, and all these statistics should always be available in order to determine the optimal solution. For all these reasons, this problem is typically simplified with certain assumptions (e.g., only communication cost is considered) and, typically, certain simplified cost models are built and an optimization algorithm may be adapted to solve it. Consequently, note that these optimization algorithms will always produce a sub-optimal solution.

#### Optimization algorithms

Game-theory and economics techniques have proved to be useful when looking for optimization algorithms to tackle the data allocation problem.

The data allocation problem is known to be a NP-hard problem produced by the large number of factors to be considered. In practice, simplified cost models (e.g., considering only communication cost over the network) and generic optimization algorithms are used to tackle this problem.