Database Management Systems

Models and Functional Architecture

Alberto Abelló Gamazo

PID_00179807

UOC
Universitat Oberta de Catalunya

Index

Introduction

This module of the *Database Architecture* subject introduces the contents of the course in two sections.

Firstly, we will see some problems associated to the relational model; there is a well-known lack of semantics in the classical relational model; moreover, in some applications, there is a mismatch between their data and the rigid structure imposed by a relational schema. To address these problems, two trends have appeared in recent years (*Object-Orientation* and *XML*, respectively).

Secondly, we will see a functional architecture of a DBMS. A functional architecture is one representing the theoretical components of the system as well as their interaction. Concrete implementations do not necessarily follow such an architecture, since they are more concerned with performance, while a functional architecture is defined for teaching and learning purposes.

Objectives

The main objective of this module is to introduce relational database management systems. Specifically:

1. Recognise the importance and usefulness of relational extensions.
2. Enumerate the components of a functional architecture of a DBMS.
3. Explain the role of each functional component of a DBMS: Query manager, Execution manager, Scheduler, and Data manager.

1. Relational Extensions

The relational model was defined by Edward F. Codd in 1970, while he was working at IBM labs.

Bibliography

Edward F. Codd (1970). "A relational model for large shared data banks". *Communications of ACM* (13(6), pp. 377-487).

To understand the relational model, it is important to know how databases were managed at that time. Thus, we will briefly see how data was stored in the 60s. Then, we will overview the relational model to appreciate its contributions to the state of the art. Finally, we will analyse the flaws of the relational model from the semantics and rigidity of schema viewpoints.

See also

This will give rise to the relational extensions explained in the module "Relational Extensions".

1.1. Pre-relational Models

In the 60s, data was stored in file systems organised into records with fields. Each application had to access many independent files. Without any kind of control, in applications with large quantities of data being constantly updated, inconsistencies between different files (or even records in the very same file) were really likely.

Thus, in the 70s, the market dealt with this using hierarchical and network data systems. These kinds of systems went a step further than independent files by managing pointers between instances. The hierarchical model allowed the definition of a tree of elements, while the network model (also known as CODASYL) allowed one child to have many parents.

CODASYL

Committee on Data Systems and Languages was a consortium of companies that, among other things, defined the COBOL programming language.

1.2. Relational Model

During the 70s, the theoretical background of the relational model was developed (based on solid mathematical foundations, i.e. sets theory) and in the early 80s the first prototypes appeared. The idea behind this was to address two different problems: maintenance of pointers and low-level procedural query languages.

Relational prototypes

The first relational prototype was System R (later DB2), and later Oracle and Ingres.

Before the relational model, pointers were physical addresses. This means that moving data from one disk position to another resulted in either an inconsistency or a cascade modification. Primary keys and foreign keys solved this problem, since they are logical pointers (i.e. independents of where data is actually stored).

From another point of view, the development of applications was time-consuming and error-prone because programmers had to deal with low-level interfaces. Thus, the efficiency of the data access was absolutely dependent on the ability of the programmer and his/her knowledge of the physical characteristics of the storage. Therefore, the main contribution of the relational model was a declarative language. SQL was defined in 1986 and standardised in 1989. From here on, the required data had only to be declared and it was the responsibility of the DBMS to find the best way to retrieve it. This resulted in a simplification of the code and real savings in development and maintenance time.

Declarative vs Procedural

In a declarative language you state what you want, while in a procedural language (like C, C++, Java, etc.) you have to state how to obtain it.

Table 1. Comparison of components at different levels

Concept	Instance	Characteristic
Relation	Tuple	Attribute
Table	Row	Column
File	Record	Field

From a theoretical point of view, data in the relational model is stored in relations (usually associated with a given concept). Each relation contains tuples (corresponding to the instances of the concept) and these are composed by attributes (showing the characteristics of each instance). In the implementation of the model, relations are represented by tables, which contain rows that have different values (one per column defined in the schema of the table). Below this, the implementations store tables in files that contain records of fields. The correspondences of these terms are summarised in Table 1.

1.3. Object-oriented Extension

In the early 90s, the object-oriented paradigm was the main trend in programming languages. This widened the gap between them and databases (the former being procedural and the latter, declarative). The theoretical background of the foundations of the relational model included several normal forms that a good design had to follow. The first of these normal forms stated that the value of an attribute had to be atomic. This explicitly forbids the possibility of storing objects, since these can contain nested structures (i.e. they are not atomic).

Object-oriented programming language

Smalltalk appeared in 1972, C ++ in 1983, Delphi in 1986, Java in 1995 and C# in 2001.

Moreover, object-oriented programming advocated the encapsulation of behaviour and data under the same structure. The maturity of relational DBMSs (RDBMSs) at the time allowed for proposals to move the behaviour of the objects to the DBMS, where it would be closer to the data. On the one hand, this would be better for maintenance and extensibility, and, on the other, it would be more efficient, since we would not have to extract data from the server to modify it and move it back to the server side again.

1st Normal Form

A relation is in 1NF if and only if each and every attribute in the relation is atomic, i.e. no attribute is itself a relation or can be decomposed into smaller pieces of information.

Finally, another criticism of the relational model was its lack of semantics. While contemporary conceptual models (e.g. Extended Entity-Relationship, or later UML) allowed for many kinds of relationships (e.g. Associations, Generalisation/Specialisation, Aggregation, etc.), the relational model only provided meaningless foreign keys pointing to primary keys.

All these improvements (i.e. non-atomic values, object definition associating behaviour to data structures, and semantic relationships) and others were added to standard SQL in 1999.

1.4. XML Extension

The emergence of the Internet generated another problem in the late 90s: data from external sources also became available to the company. Until then, all data was generated inside and under the control of the IT department of the company. Therefore, the applications and systems knew exactly what to expect. It was always possible to define the schema of data.

Nevertheless, when the source of data is not under our control, we need to be prepared for the unexpected. It was clear that some tuples could have attributes that others would not. Moreover, it was quite common to find special attributes only in some tuples, or attributes that were no longer provided without warning, or the other way round new attributes were suddenly added to some tuples.

In this context, the rigid schema of a relation does not seem appropriate to store data. Something more flexible is needed.

XML was added to standard SQL in 2003.

2. Functional architecture of a DBMS

In this section we analyse the different components of a DBMS. A component is a piece of software in charge of a given functionality. It is in this sense that we introduce a functional architecture (i.e. the organisation and interaction of the components taking care of the different functionalities of the DBMS).

Figure 1 shows that we have to manage persistent as well as volatile storage systems. This implies the implementation of certain recovery mechanisms to guarantee that both storage systems are always synchronised and data is not lost in the event of system or media failures. On top of this, the presence of concurrent users forces the scheduling of the different operations to avoid interferences between them. Both functionalities are necessary to guarantee that certain sets of operations of a user in a session can be managed together into a transaction. However, as explained before, the most important components of a DBMS are the query manager and the execution manager. The former provides many functionalities, namely view resolution, security management, constraint checking and query optimisation. It generates a query plan that the execution manager implements.

We must take into account that the persistent as well as the volatile storage may be distributed (if so, different fragments of data can be located at different sites). Also, many processors could be available (in which case, the query optimiser has to take into account the possibility of parallelising the execution). Obviously, this would affect certain parts of this architecture, mainly depending on the heterogeneities we find in the characteristics of the storage systems we use and the semantics of the data stored.

Parallel and distributed computing dramatically influences the different components of this architecture.

See also

We will not go into detail in this introductory module, but we will see this in different modules devoted to each component.

2.1. Query Manager

The query manager is the component of the DBMS in charge of transforming a declarative query into an ordered set of steps (i.e. procedural description). This transformation is even more difficult taking into account that, following the

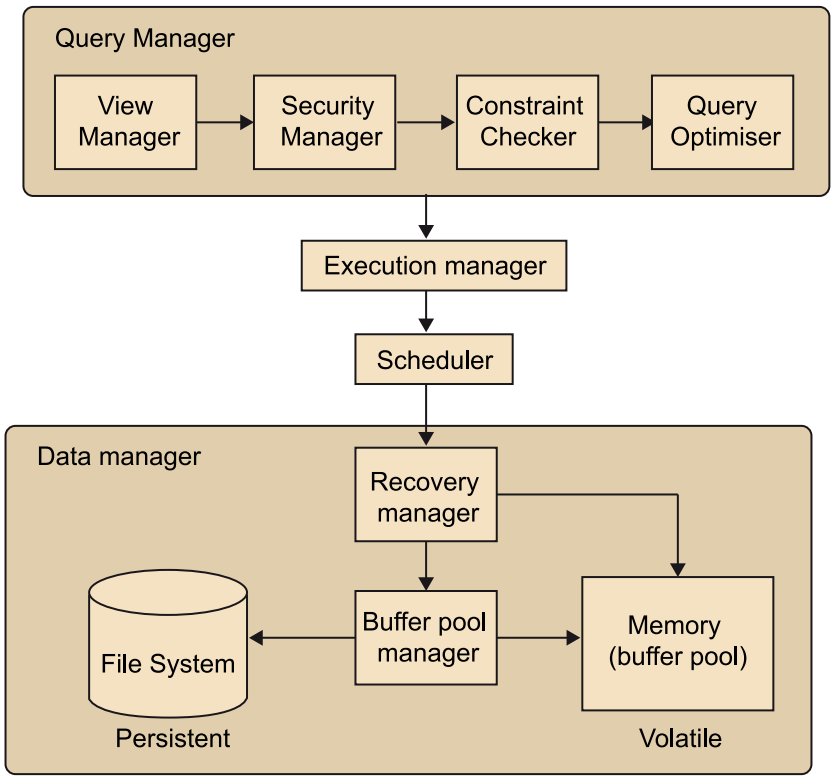
(1)This comes from different groups of people giving different meanings to the same words.

ANSI/SPARC architecture, DBMSs must provide views to deal with semantic relativism[1]. It is also relevant to note that security and constraints have to be taken into account.

ANSI/SPARC

The Standards Planning and Requirements Committee of the American National Standards Institute defined a three-level architecture in 1975 to abstract users from physical storage, which is still used in today's DBMSs.

Figure 1. Functional architecture of a DBMS



2.1.1. View Manager

From the point of view of the end user, there is no difference in querying data in a view or in a table. However, dealing with views is not easy and we will not go into detail about it in this course. Nevertheless, we will take a brief look at the difficulties it poses in this section.

First of all, we must take into account that data in the view may be physically stored (i.e. *materialised*) or not, which poses new difficulties. If data in the view is not physically stored, in order to transform the query over the views into a query over the source tables, we must substitute the view name in the user query by its definition (this is known as *view expansion*). In some cases, it is more efficient to instruct the DBMS to calculate the view result and store it while waiting for the queries. However, if we do this, we have to be able to transform an arbitrary query over the tables into a query over the available materialised views (this is known as *query rewriting*), which is somewhat contrary to view expansion (in the sense that we have to identify the view definition in the user query and substitute it with the view name). If we are able

to rewrite a query, we still have to decide whether it is worth rewriting it or not, i.e. check if the redundant data can be used to improve the performance of the query (this is known as *answering queries using views*).

Finally, updating data in the presence of views is also more difficult. Firstly, we would like to allow users to express not only queries but also updates in terms of views (this is known as *update through views*), which is only possible in few cases. Secondly, if views are materialised, changes in the sources are potentially propagated to the views (this is known as *view updating*).

2.1.2. Security Manager

Ethics as well as legal issues raise the need to control access to data. We cannot allow any user to query or modify all the data in our database. This component defines user privileges and, once it has done this, validates user statements by checking whether or not they are allowed to perform the action in question.

See also

We will go into the implementation details of this component in the "Security" module.

2.1.3. Constraint Checker

Another important aspect is guaranteeing the integrity of the data. The database designer defines constraints over the schema that must be subsequently enforced so that user modifications of data do not violate it. Although it is not evident, its theoretical background is closely related to that of view management.

Note

We will not elaborate on the problems related to constraint checking implementation during this course.

2.1.4. Query Optimiser

Clearly, the most important and complex part of the query manager is the optimiser, because it seeks to find the best way to execute user statements. Thus, its behaviour will have a direct impact on the performance of the system. Remember that it has three components, namely semantic, syntactic and physical optimisers.

See also

In the "Distributed queries optimisation" module, we will see how parallelism and distribution of data affect this component.

2.2. Execution Manager

The query optimiser breaks down the query into a set of atomic operations (mostly those of relational algebra). It is the task of this component to coordinate the step-by-step execution of these elements. In distributed environments, this component is also responsible for assigning the execution of each operation to a given site.

2.3. Scheduler

As we know, many users (up to tens or hundreds of thousands) can work concurrently on a database. In this case, it is quite likely that they will want to access not only the same table, but exactly the same column and row. If so, they could interfere with each other's task. The DBMS must provide certain mechanisms to deal with this problem. Generally, the way this is done is by restricting the execution order of the reads, writes, commits and aborts of the different users. On receiving a command, the scheduler can pass it directly to the data manager, queue it and wait for the appropriate time to execute it, or cancel it permanently (which aborts its transaction). The most basic and commonly used mechanism to avoid interferences is Shared-eXclusive locking.

See also

In the "Transaction models and Concurrency control" module we will explain more advanced centralised and distributed mechanisms (namely *Multi-granule locking*, *Multiversion*, and *Timestamping*).

2.4. Data Manager

As we know, memory storage is much faster than disk storage (up to hundreds of thousands of times faster). However, it is volatile and much more expensive. Being expensive means that its size is limited and it can only contain a small part of the database. Moreover, being volatile means that switching off the server or simply rebooting it would cause us to lose our data. It is the task of the data manager to take advantage of both kinds of storage while smoothing out their weaknesses.

See also

We will explain this in detail in the "Data Management" module.

2.4.1. Buffer Pool Manager

The data manager has a component in charge of moving data from disk to memory (i.e. *fetch*) and from memory to disk (i.e. *flush*) to meet the requests it receives from other components.

Data have to be moved from disk to memory to make it accessible to other components of the DBMS. Sooner or later, if the data have not been modified, they are simply removed from memory to make room for other data. However, when they are modified, data have to be moved from memory to disk in order to avoid losing that modification.

The simplest way to manage this is known as write through. Unfortunately, this is quite inefficient, because the disk (which is a slow component) becomes a bottleneck for the whole system. It would be much more efficient to leave a chunk of data in memory waiting for several modifications and then to make all of the persistent at the same time with a single disk write. Thus, we will have a buffer pool to keep the data temporarily in the memory when expecting lots of modifications in a short period of time.

Write through

This means that any modification of data is immediately sent to disk, which guarantees that nothing is lost in the event of a power failure.

Waiting for lots of modifications before writing data to disk is worth it, because the transfer unit between memory and disk is a block (not a single byte at a time), and it is likely that several modifications of the data will be contained in the same block over a short period of time.

Block size

The default block size in Oracle 10g is 8Kb.

2.4.2. Recovery Manager

Waiting for many modifications before writing data to disk may result in losing certain user requests in the event of power failure. Imagine the following situation: a user executes a statement modifying tuple t_1, the system confirms the execution, but does not write the data to disk as it is waiting for other modifications of tuples in the same block; unfortunately, there is a power failure before the next modifications arrive. In the event of system failure, all the system would have is what was stored on the disk. If the DBMS did not implement some sort of safety mechanism, the modification of t_1 would be lost. It is the task of this component to avoid such a loss. Moreover, this component also has the task of undoing all changes made during a rolled-back transaction. As with other components, doing this in a distributed environment is much more difficult than in a centralised one.

Note that the interface to the buffer pool manager is always through the recovery manager (i.e. no components but this can gain direct access to it).

We must not only foresee system failures, but also media failures. For example, in the event of a disk failure, it will also be the responsibility of this component to provide the means to recover all data in it. Remember that the "Durability" property of centralised ACID transactions states that once a modification is committed it cannot be lost under any circumstances.

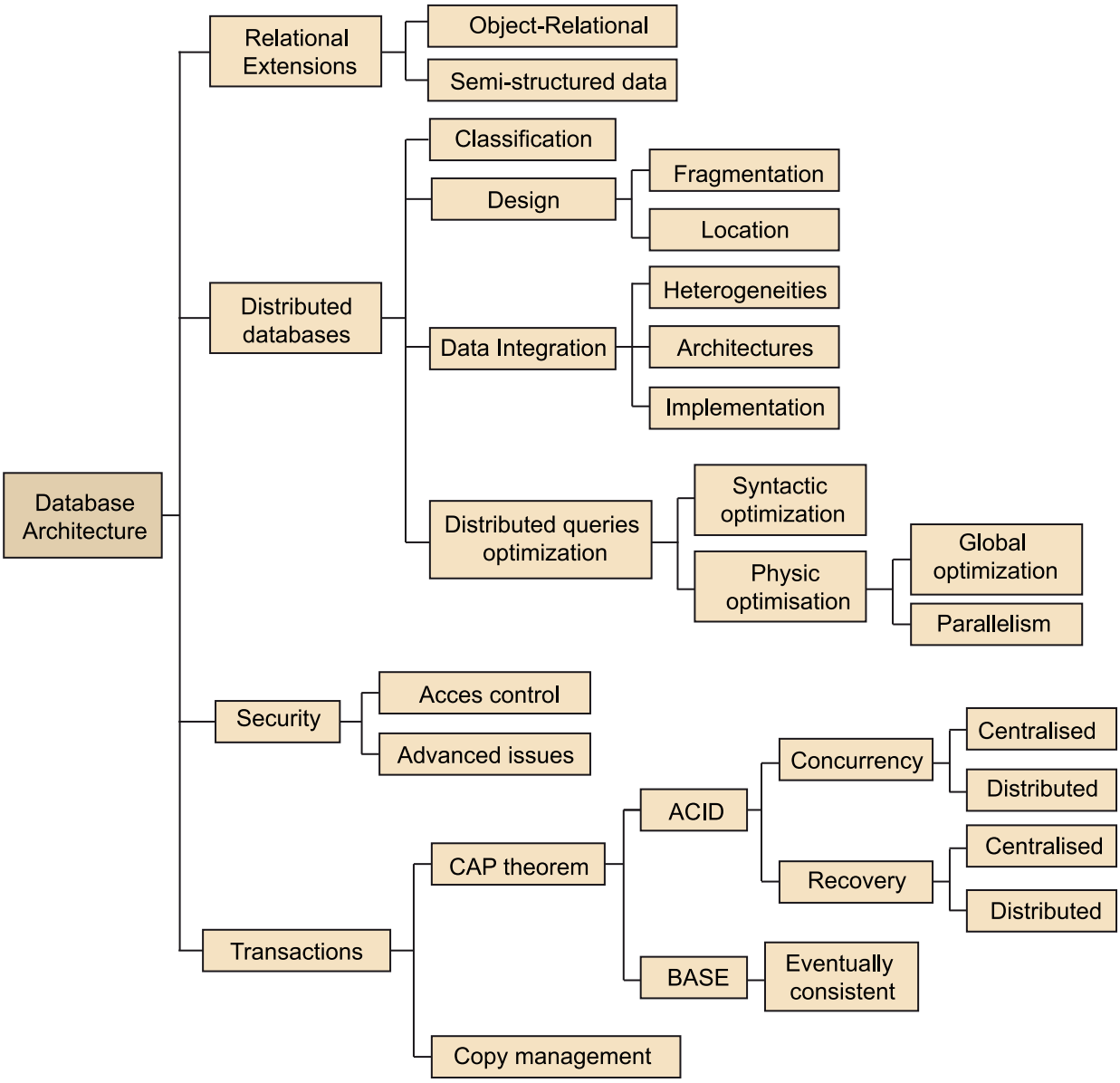
Summary

In this introductory module, we have briefly reviewed the history of databases, starting from file systems, and going throughout pre-relational systems, relational systems and post-relational extensions (i.e. object and semi-structured storage). We have seen the main reasons for each of these changes: inconsistent redundant data being stored independently, physical storage dependence and time-consuming data access, complex data storage with associated behaviour, and the existence of highly incomplete or unstructured data.

We then outlined the different functionalities required in a DBMS: firstly, query processing (including views, access control, constraint checking and, last but not least, optimisation) decides on the best way to access the requested data; the resulting access plan is then passed through the execution manager, which allows us to handle and coordinate a set of operations as a unit; the scheduler subsequently reorganises the operations to avoid interferences between users and, finally, the data manager helps to prevent disk bottlenecks by efficiently moving data from disk to memory and vice versa, guaranteeing that no data will be lost in the event of system failure.

Conceptual map

The following conceptual map illustrates the contents of this subject. Note that it reflects the structure of the contents as opposed to that of the modules.


Database Architecture
Relational Extensions
Object-Relational
Semi-structured data
Distributed databases
Classification
Design
Fragmentation
Location
Data Integration
Heterogeneities
Architectures
Implementation
Distributed queries optimization
Syntactic optimization
Physic optimisation
Global optimization
Parallelism
Security
Acces control
Advanced issues
Transactions
CAP theorem
ACID
Concurrency
Centralised
Distributed
Recovery
Centralised
Distributed
BASE
Eventually consistent
Copy management

Self-evaluation

1. What is the theoretical limitation to storing objects in a relational database?

2. Name the four components of the query manager.

3. What makes it possible to improve the performance of a write-through mechanism for the transfer of data from memory to disk?

4. Would the recovery manager still be necessary in a DBMS implementing write-through to transfer data from memory to disk?

Answer key

Self-evaluation

1. 1NF.

2. View manager, Security manager, Constraint checker and Query optimiser.

3. The fact that the transfer unit is blocks as opposed to bytes and the locality of the modifications (i.e. several modifications being located in the same block within a short period of time).

4. Yes, because we still have to undo changes made by aborted transactions.

Glossary

DBMS Database Management System

IT Information Technology

O-O Object-Oriented

RDBMS Relational DBMS

SQL Structured Query Language

XML eXtensible Markup Language

Bibliography

Garcia-Molina, H; Ullman, J. D.; Widom, J. (2009). *Database systems* (second edition). Pearson/Prentice Hall.

Bernstein, P. A.; Hadzilacos, V.; Goodman, N. (1987). *Concurrency control and recovery in database systems*. Addison-Wesley.