2

Architecture and Categorization of DBMSs

◈

Chapter Objectives

In this chapter, you will learn to:

- identify the key components of a DBMS architecture;
- understand how these components work together for data storage, processing, and management;
- categorize DBMSs based upon data model, degree of simultaneous access, architecture, and usage.

Opening Scenario

To kick-start its business, Sober purchased the customer database of Mellow Cab, a firm that recently stepped out of the taxi business. Unfortunately, the database has been handed over in a legacy CODASYL format that Sober is not familiar with. Sober also needs a new database to store transaction details whenever passengers book either a ride-hailing or ride-sharing service. Other data (e.g., multimedia) are an option they are interested in. Sober wants to continuously store the location of its taxis and periodically review hot-spot pick-up and drop-off locations. Sober is looking at ways to manage all these data sources in the optimal way.

As discussed in Chapter 1, a DBMS supports the creation, usage, and maintenance of a database. It consists of several modules, each with their specific functionality, that work together according to a predefined architecture. In this chapter, we zoom into this internal architecture and provide a categorization of DBMSs along various dimensions. The overview of the chapter is straightforward. We start by discussing the components that together make up a DBMS. Next, we provide a classification of DBMSs in terms of data model, degree of simultaneous access, architecture, and usage.

2.1 Architecture of a DBMS

As discussed before, a DBMS needs to support various types of data management-related activities, such as querying and storage. It also must provide interfaces to its environment. To achieve both of these goals, a DBMS is composed of various interacting modules that together make up the **database management system architecture**. Figure 2.1 shows an overview of the key components of a DBMS architecture. We review each component in more detail in what follows.

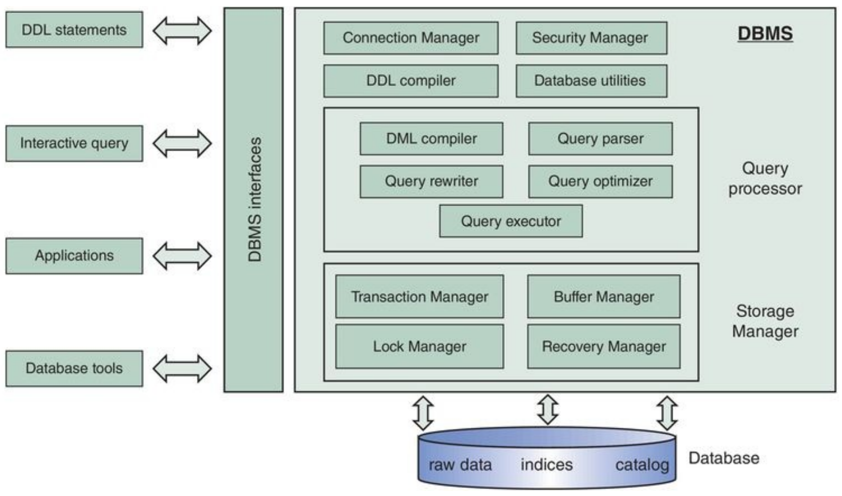


Figure 2.1 Architecture of a database management system (DBMS).

Figure 2.1 is by no means exhaustive. Depending upon the vendor and implementation, some components may be left out and others added. On the left, you can see various ways of interacting with the DBMS. **DDL statements** create data definitions that are stored in the catalog. **Interactive queries** are typically executed from a front-end tool, such as a command-line interface, simple graphical user interface, or forms-based interface. Applications interact with the DBMS using **embedded DML statements**. Finally, the *database administrator*

(DBA) can use various database tools to maintain or fine-tune the DBMS. To facilitate all these usages, the DBMS provides various interfaces that invoke its components. The most important components are: the connection manager; the security manager; the DDL compiler; various database utilities; the query processor; and the storage manager. The query processor consists of a DML compiler, query parser, query rewriter, query optimizer, and query executor. The storage manager includes a transaction manager, buffer manager, lock manager, and recovery manager. All these components interact in various ways depending upon which database task is executed. The database itself contains the raw data or database state and the catalog with the database model and other metadata, including the indexes that are part of the internal data model providing quick access to the data. In the rest of this section we discuss each component more extensively.

2.1.1 Connection and Security Manager

The **connection manager** provides facilities to set-up a database connection. It can be set-up locally or through a network, the latter being more common. It verifies the logon credentials, such as user name and password, and returns a connection handle. A database connection can run either as a single process or as a thread within a process. Remember, a thread represents an execution path within a process and represents the smallest unit of processor scheduling. Multiple threads can run within a process and share common resources such as memory. The security manager verifies whether a user has the right privileges to execute the database actions required. For example, some users can have read access while others have write access to certain parts of the data. The security manager retrieves these privileges from the catalog.

2.1.2 DDL Compiler

The **DDL compiler** compiles the data definitions specified in DDL. Ideally, the DBMS should provide three DDLs: one for the internal data model; one for the logical data model; and one for the external data model. Most implementations, however, have a single DDL with three different sets of instructions. This is the case for most relational databases that use SQL as their DDL. The DDL compiler first parses the DDL definitions and checks their syntactical correctness. It then translates the data definitions to an internal format and generates errors if required. Upon successful compilation, it registers the data definitions in the catalog, where they can be used by all the other components of the DBMS.

Connections

Chapter 7 discusses how SQL can be used to define a logical and external data model in a relational environment. Chapter 13 reviews how SQL can be used to define an internal data model.

2.1.3 Query Processor

The **query processor** is one of the most important parts of a DBMS. It assists in the execution of database queries such as retrieval of data, insertion of data, update of data, and removal of data from the database. Although most DBMS vendors have their own proprietary query processor, it usually includes a DML compiler, query parser, query rewriter, query optimizer, and query executor.

2.1.3.1 DML Compiler

The **DML compiler** compiles the data manipulation statements specified in DML. Before we explain its functioning, we need to elaborate on the different types of DML. As discussed in Chapter 1, DML stands for data manipulation language. It provides a set of constructs to select, insert, update, and delete data.

The first data manipulation languages developed were predominantly **procedural DML**. These DML statements explicitly specified how to navigate in the database to locate and modify the data. They usually started by positioning on one specific record or data instance and navigating to other records using memory pointers. Procedural DML is also called **record-at-a-time DML**. DBMSs with procedural DML had no query processor. In other words, the application developer had to explicitly define the query optimization and execution him/herself. To write efficient queries, the developer had to know all the details of the DBMS. This is not a preferred implementation since it complicates the efficiency, transparency, and maintenance of the database applications. Unfortunately, many firms are still struggling with procedural DML applications due to the legacy DBMSs still in use.

Declarative DML is a more efficient implementation. Here, the DML statements specify which data should be retrieved or what changes should be made, rather than how this should be done. The DBMS then autonomously determines the physical execution in terms of access path and navigational strategy. In other words, the DBMS hides the implementation details from the application developer, which facilitates the development of database applications. Declarative DML is usually **set-at-a-time DML**, whereby sets of records or data instances can be retrieved at once and provided to the application. Only the selection criteria are provided to the DBMS; depending on the actual database state, zero, one, or many records will qualify. A popular example of declarative DML is SQL, which we discuss extensively in Chapter 7.

Many applications work with data stored in a database. To access a database and work with it, DML statements will be directly embedded in the host language. The host language is the general-purpose programming language that contains the (non-database related) application logic. Obviously, both host language and DML should be able to successfully interact and exchange data.

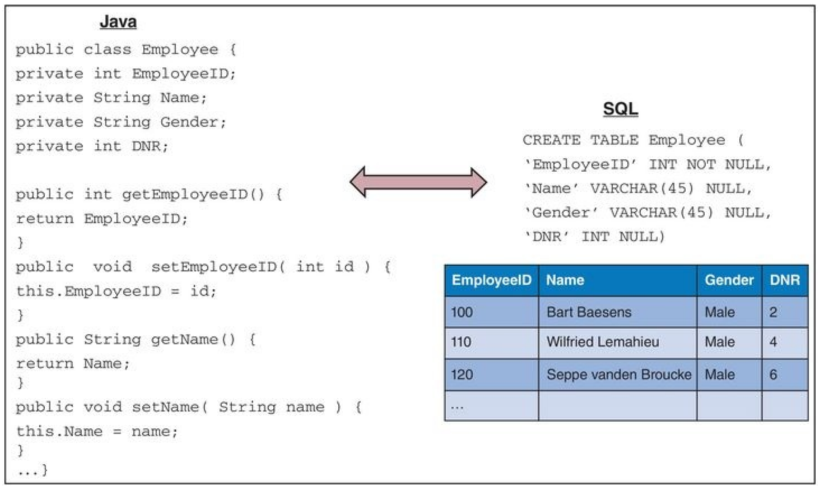
As an example, think about a Java application that needs to retrieve employee data from a database. It can do this by using SQL, which is one of the most popular querying languages used in the industry nowadays. In the following Java program, the SQL DML statements are highlighted in bold face.

```
import java.sql.*;
public class JDBCExample1 {
public static void main(String[] args) {
try {
  System.out.println("Loading JDBC driver…");
  Class.forName("com.mysql.jdbc.Driver");
  System.out.println("JDBC driver loaded!");
} catch (ClassNotFoundException e) {
  throw new RuntimeException(e);
}
String url = "jdbc:mysql://localhost:3306/employeeschema";
String username = "root";
String password = "mypassword123";
String query = "select E.Name, D.DName " +
"from employee E, department D " +
"where E.DNR = D.DNR;";
Connection connection = null;
Statement stmt = null;
try {
  System.out.println("Connecting to database");
  connection = DriverManager.getConnection(url, username,
password);
  System.out.println("MySQL Database connected!");
  stmt = connection.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  while (rs.next()) {
    System.out.print(rs.getString(1));
    System.out.print("");
    System.out.println(rs.getString(2));
  }
  stmt.close();
} catch (SQLException e) {
  System.out.println(e.toString());
} finally {
  System.out.println("Closing the connection.");
  if (connection != null) {
  try {
  connection.close();
} catch (SQLException ignore) {}}}}
```

Without going into any language or syntax specifics, this Java application first initiates a database connection with a given username and password. Next, the application executes an SQL query that asks for the employee names together with their department names. It then iterates through the results, whereby at each step the employee name and corresponding department name are displayed on the screen.

Embedding DML statements into a host language is not as straightforward as it may at first seem. The data structures of the DBMS and the DML may differ from the data structures of the host language. In our example, we used Java, which is an object-oriented host language, and combined it with MySQL, which is a relational DBMS using SQL DML. The mapping between object-oriented and relational concepts is often called the impedance mismatch problem. It can be solved in various ways. First, we can choose a host language and DBMS with comparable data structures. In other words, we combine Java with an object-oriented DBMS, which allows transparent retrieval and storage of data. As an alternative, we could also opt to use middleware to map the data structures from the DBMS to the host language and vice versa. Both options have their pros and cons and are discussed more extensively in Chapter 8.

Figure 2.2 shows the impedance mismatch problem. On the left, we have a Java class Employee with characteristics such as EmployeeID, Name, Gender, and DNR (which is the department number). It also has “getter” and “setter” methods to implement the object-oriented principle of information hiding. To the right, we have the corresponding SQL DDL that essentially stores information in a tabular format.



```
public class Employee {
private int EmployeeID;
private String Name;
private String Gender;
private int DNR;

public int getEmployeeID() {
return EmployeeID;
}
public  void  setEmployeeID( int id ) {
this.EmployeeID = id;
}
public String getName() {
return Name;
}
public void setName( String name ) {
this.Name = name;
}
...}
```

```
CREATE TABLE Employee (
'EmployeeID' INT NOT NULL,
'Name' VARCHAR(45) NULL,
'Gender' VARCHAR(45) NULL,
'DNR' INT NULL)
```

EmployeeID	Name	Gender	DNR
100	Bart Baesens	Male	2
110	Wilfried Lemahieu	Male	4
120	Seppe vanden Broucke	Male	6
...			

Figure 2.2 The impedance mismatch problem.

The DML compiler starts by extracting the DML statements from the host language. It then closely collaborates with the query parser, query rewriter, query optimizer, and query executor for executing the DML statements. Errors are generated and reported if necessary.

Connections

Chapter 5 introduces hierarchical and CODASYL data models which both assume procedural, record-at-a-time DML. Chapter 7 reviews SQL, which is declarative, set-at-time DML.

2.1.3.2 Query Parser and Query Rewriter

The query parser parses the query into an *internal representation format* that can then be further evaluated by the system. It checks the query for syntactical and semantical correctness. To do so, it uses the catalog to verify whether the data concepts referred to are properly defined there, and to see whether the integrity rules have been respected. Again, errors are generated and reported if necessary.

The **query rewriter** optimizes the query, independently of the current database state. It simplifies it using a set of predefined rules and heuristics that are DBMS-specific. In a relational database management system, nested queries might be reformulated or flattened to join queries. We discuss both types of queries more extensively in Chapter 7.

2.1.3.3 Query Optimizer

The **query optimizer** is a very important component of the query processor. It optimizes the query based upon the current database state. It can make use of predefined indexes that are part of the internal data model and provide quick access to the data. The query optimizer comes up with various query execution plans and evaluates their cost (in terms of resources required) by aggregating the estimated number of input/output operations, the plan’s estimated CPU processing cost and the plan’s estimated execution time into the total estimated response time. A good execution plan should have a low response time. It is important to note that the response time is estimated and not exact. The estimates are made using catalog information combined with statistical inference procedures. Empirical distributions of the data are calculated and summarized by their means, standard deviations, etc. Coming up with accurate estimates is crucial in a good query optimizer. Finding an optimal execution path is essentially a classical search or optimization problem whereby techniques such as dynamic programming can be used. As already mentioned, the implementation of the query optimizer depends upon the type of DBMS and the vendor, and is a key competitive asset.

2.1.3.4 Query Executor

The result of the query optimization procedure is a final execution plan which is then handed over to the query executor. The **query executor** takes care of the actual execution by calling on the storage manager to retrieve the data requested.

2.1.4 Storage Manager

The **storage manager** governs physical file access and as such supervises the correct and efficient storage of data. It consists of a transaction manager, buffer manager, lock manager, and recovery manager. Let’s zoom in for more detail.

2.1.4.1 Transaction Manager

The **transaction manager** supervises the execution of database transactions. Remember, a database transaction is a sequence of read/write operations considered to be an atomic unit. The transaction manager creates a schedule with interleaved read/write operations to improve overall efficiency and execution performance. It also guarantees the atomicity, consistency, isolation and durability or ACID properties in a multi-user environment (see Chapter 1). The transaction manager will “commit” a transaction upon successful execution, so the effects can be made permanent, and “rollback” a transaction upon unsuccessful execution, so any inconsistent or bad data can be avoided.

2.1.4.2 Buffer Manager

The **buffer manager** is responsible for managing the buffer memory of the DBMS. This is part of the internal memory, which the DBMS checks first when data need to be retrieved. Retrieving data from the buffer is significantly faster than retrieving them from external disk-based storage. The buffer manager is responsible for intelligently caching the data in the buffer for speedy access. It needs to continuously monitor the buffer and decide which content should be removed and which should be added. If data in the buffer have been updated, it must also synchronize the corresponding physical file(s) on disk to make sure updates are made persistent and are not lost. A simple buffering strategy is based upon data locality that states that data recently retrieved are likely to be retrieved again. Another strategy uses the 20/80 law, which implies that 80% of the transactions read or write only 20% of the data. When the buffer is full, the buffer manager needs to adopt a smart replacement strategy to decide which content should be removed. Furthermore, it must be able to serve multiple transactions simultaneously. Hence, it closely interacts with the lock manager to provide concurrency control support.

2.1.4.3 Lock Manager

The **lock manager** is an essential component for providing concurrency control, which ensures data integrity at all times. Before a transaction can read or write a database object, it must acquire a lock which specifies what types of data operations the transaction can carry out. Two common types of locks are read and write locks. A **read lock** allows a transaction to read a database object, whereas a **write lock** allows a transaction to update it. To enforce transaction atomicity and consistency, a locked database object may prevent other transactions from using it, hence avoiding conflicts between transactions that involve the same data. The lock manager is responsible for assigning, releasing, and recording locks in the catalog. It makes use of a *locking protocol*, which describes the locking rules, and a lock table with the lock information.

2.1.4.4 Recovery Manager

The **recovery manager** supervises the correct execution of database transactions. It keeps track of all database operations in a logfile, and will be called upon to undo actions of aborted transactions or during crash recovery.

Connections

Chapter 14 elaborates further on the activities of the transaction, buffer, lock, and recovery managers.

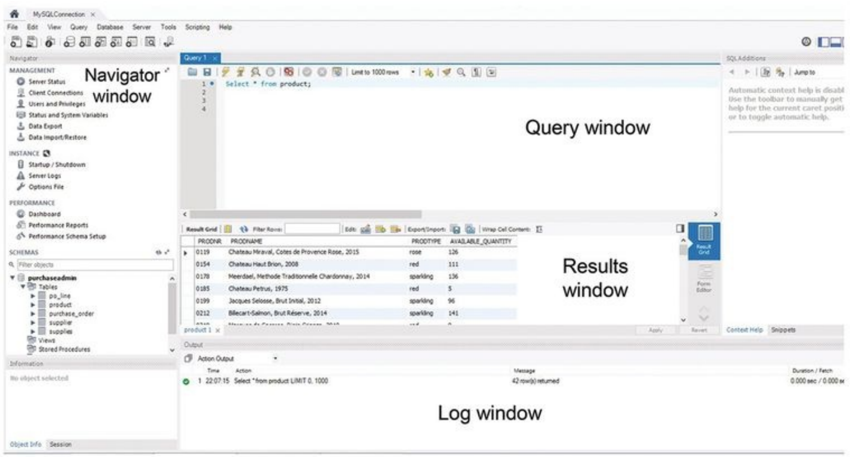
2.1.5 DBMS Utilities

Besides the components we discussed before, a DBMS also comes with various utilities. A **loading utility** supports the loading of the database with information from a variety of sources, such as another DBMS, text files, Excel files, etc. A **reorganization utility** automatically reorganizes the data for improved performance. **Performance monitoring utilities** report various key performance indicators (KPIs), such as storage space consumed, query response times, and transaction throughput rates to monitor the performance of a DBMS. **User management utilities** support the creation of user groups or accounts, and the assignment of privileges to them. Finally, a **backup and recovery utility** is typically included.

2.1.6 DBMS Interfaces

A DBMS needs to interact with various parties, such as a database designer, a database administrator, an application, or even an end-user. To facilitate this communication, it provides various **user interfaces** such as a *web-based interface*, a *stand-alone query language interface*, a *command-line interface*, a *forms-based interface*, a *graphical user interface*, a *natural language interface*, an *application programming interface* (API), an *admin interface*, and a *network interface*.

Figure 2.3 shows an example of the MySQL Workbench interface. You can see the navigator window with the management, instance, performance, and schemas section. The query window provides an editor to write SQL queries. In our case, we wrote a simple SQL query to ask for all information from the product table. The results window displays the results of the execution of the query. The log window provides a log with actions and possible errors.



```
Select * from product;
```

	PRODNR	PRODNAME	PRODTYPE	AVAILABLE_QUANTITY
▶	0119	Chateau Miraval, Cotes de Provence Rose, 2015	rose	126
	0154	Chateau Haut Brion, 2008	red	111
	0178	Meerdael, Methode Traditionnelle Chardonnay, 2014	sparkling	136
	0185	Chateau Petrus, 1975	red	5
	0199	Jacques Selosse, Brut Initial, 2012	sparkling	96
	0212	Billecart-Salmon, Brut Réserve, 2014	sparkling	141

```
   Time      Action                                Message              Duration / Fetch
1  22:07:15  Select * from product LIMIT 0, 1000   42 row(s) returned   0.000 sec / 0.000 se
```

Figure 2.3 MySQL interface.

Retention Questions

- What are the key components of a DBMS?
- What is the difference between procedural and declarative DML?
- Give some examples of DBMS utilities and interfaces.

2.2 Categorization of DBMSs

Given the proliferation of DBMSs available, in this section we introduce a categorization according to various criteria. We discuss categorization of DBMSs based upon data model, simultaneous access, architecture, and usage. Note that our categorization is not to be interpreted in an exhaustive or exclusive way. It can thus be that a DBMS falls into multiple categories simultaneously. Other categories may also be considered.

2.2.1 Categorization Based on Data Model

Throughout the past decades, various types of data models have been introduced for building conceptual and logical data models. We briefly summarize them here and provide more detail in later chapters.

2.2.1.1 Hierarchical DBMSs

Hierarchical DBMSs were one of the first DBMS types developed, and adopt a tree-like data model. The DML is procedural and record-oriented. No query processor is included. The definitions of the logical and internal data model are intertwined, which is not desirable from a usability, efficiency, or maintenance perspective. Popular examples are IMS from IBM and the Registry in Microsoft Windows.

2.2.1.2 Network DBMSs

Network DBMSs use a network data model, which is more flexible than a tree-like data model. One of the most popular types are CODASYL DBMSs, which implement the CODASYL data model. Again, the DML is procedural and record-oriented, and no query processor is available. Consequently, the definitions of the logical and internal data models are also intertwined. Popular examples are CA-IDMS from Computer Associates, UDS from Siemens Nixdorf, DMS 1100 from Unisys, and Image from HP. Both hierarchical and CODASYL DBMSs are legacy database software.

2.2.1.3 Relational DBMSs

Relational DBMSs (RDBMSs) use the relational data model and are the most popular in the industry. They typically use SQL for both DDL and DML operations. SQL is declarative and set oriented. A query processor is provided to optimize and execute the database queries. Data independence is available thanks to a strict separation between the logical and internal data model. This makes it very attractive to develop powerful database applications. Popular examples are MySQL, which is open-source and maintained by Oracle, the Oracle DBMS also provided by Oracle, DB2 from IBM, and Microsoft SQL Server from Microsoft.

2.2.1.4 Object-Oriented DBMSs

Object-oriented DBMSs (OODBMSs) are based upon the object-oriented data model. An object encapsulates both data (also called variables) and functionality (also called methods). When combining an OODBMS with an object-oriented programming language (e.g., Java, Python), there is no impedance mismatch since the objects can be transparently stored and retrieved from the database. Examples of OODBMSs are db4o, which is an open-source OODBMS maintained by Versant, Caché from Intersystems, and GemStone/S from GemTalk Systems. OODBMSs are not very popular in the industry, beyond some niche markets, due to their complexity.

2.2.1.5 Object-Relational/Extended Relational DBMSs

An **object-relational DBMS (ORDBMS)**, also commonly called an **extended relational DBMS (ERDBMS)**, uses a relational model extended with object-oriented concepts, such as user-defined types, user-defined functions, collections, inheritance, and behavior. Hence, an ORDBMS/ERDBMS shares characteristics with both an RDBMS and an OODBMS. As with pure relational

DBMSs, the DML is SQL, which is declarative and set oriented. A query processor is available for query optimization. Most relational DBMSs such as Oracle, DB2, and Microsoft SQL Server incorporate object-relational extensions.

2.2.1.6 XML DBMSs

XML DBMSs use the XML data model to store data. XML is a data representation standard. Here you can see an example of an XML fragment.

```
<employee>
      <firstname>Bart</firstname>
      <lastname>Baesens</lastname>
      <address>
          <street>Naamsestraat</street>
          <number>69</number>
          <zipcode>3000</zipcode>
          <city>Leuven</city>
          <country>Belgium</country>
      </address>
      <gender>Male</gender>
</employee>
```

You can see we have various tags, such as employee, firstname, lastname, etc. The address tag is further subdivided into street, number, zip code, city, and country tags. It is important that every <tag> is properly closed with a </tag>. An XML specification essentially represents data in a hierarchical way. Figure 2.4 shows the tree corresponding to our XML specification.

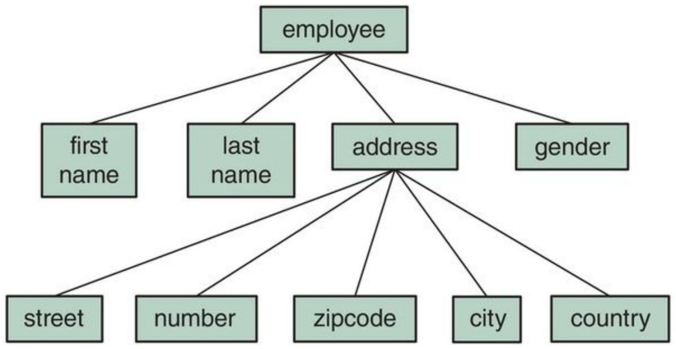


Figure 2.4 Tree-based XML representation.

XML is a very popular standard to exchange data between various applications. Native XML DBMSs (e.g., BaseX, eXist) store XML data by using the logical, intrinsic structure of the XML document. More specifically, they map the hierarchical or tree structure of an XML document to a physical storage structure. XML-enabled DBMSs (e.g., Oracle, IBM DB2) are existing RDBMSs or ORDBMSs that are extended with facilities to store XML data and structured data in an integrated and transparent way. Both types of DBMSs also provide facilities to query XML data.

2.2.1.7 NoSQL DBMSs

Finally, the last few years brought us a realm of new database technologies targeted at storing big and unstructured data. These are often referred to using the umbrella term **not-only SQL (NoSQL)** databases with popular examples such as Apache Hadoop or Neo4j. As we explain in Chapter 11, NoSQL databases can be classified according to data model into key–value stores, tuple, or document stores, column-oriented databases, and graph databases. However, even within such subcategories, the heterogeneity of the members is quite high. The common denominator of all NoSQL databases is that they attempt to make up for some shortcomings of relational DBMSs in terms of scalability and the ability to cope with irregular or highly volatile data structures.

Connections

Chapter 5 reviews both hierarchical and network DBMSs. Chapters 6 and 7 discuss relational DBMSs. Object-oriented DBMSs are covered in Chapter 8, whereas Chapter 9 reviews object-relational DBMSs. XML DBMSs are introduced in Chapter 10. Chapter 11 discusses NoSQL DBMSs.

2.2.2 Categorization Based on Degree of Simultaneous Access

DBMSs can also be categorized based upon the degree of **simultaneous access**. In a **single-user system**, only one user at a time is allowed to work with the DBMS. This is not desirable in a networked environment. **Multi-user systems** allow multiple users to simultaneously interact with the database in a distributed environment, as illustrated in Figure 2.5 where three clients are being served by three server instances or threads.

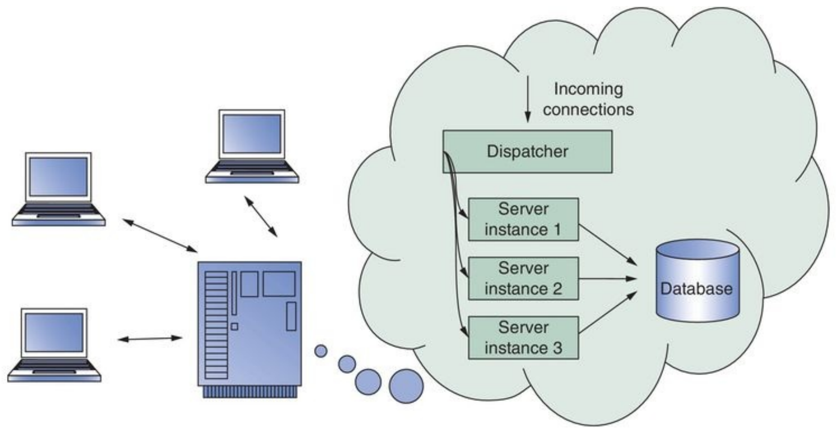


Figure 2.5 Simultaneous access to a DBMS.

To do so successfully, the DBMS should support multi-threading and provide facilities for concurrency control. A dispatcher component then typically distributes the incoming database requests among server instances or threads.

2.2.3 Categorization Based on Architecture

The architectural development of DBMSs is similar to that of computer systems in general. In a **centralized DBMS architecture**, the data are maintained on a centralized host, e.g., a mainframe system. All queries will then have to be processed by this single host.

In a **client–server DBMS architecture**, active clients request services from passive servers. A *fat client variant* stores more processing functionality on the client, whereas a *fat server variant* puts more on the server.

The **n-tier DBMS architecture** is a straightforward extension of the client–server architecture. A popular example is a client with GUI (graphical user interface) functionality, an application server with the various applications, a database server with the DBMS and database, and a web server for the web-based access. The communication between these various servers is then handled by middleware.

In a **cloud DBMS architecture**, the DBMS and database are hosted by a third-party cloud provider. The data can then be distributed across multiple computers in a network. Although this is sometimes a cost-effective solution, depending on the context it can perform less efficiently in terms of processing queries or other database transactions. Popular examples are the Apache Cassandra project and Google's BigTable.

A **federated DBMS** is a DBMS that provides a uniform interface to multiple underlying data sources such as other DBMSs, file systems, document management systems, etc. By doing so, it hides the underlying storage details (in particular the distribution and possible heterogeneity of data formats and data management functionality) to facilitate data access.

An **in-memory DBMS** stores all data in internal memory instead of slower external storage such as disk-based storage. It is often used for real-time purposes, such as in Telco or defense applications. Periodic snapshots to external storage can be taken to support data persistence. A popular example of an in-memory DBMS is SAP’s Hana product.

2.2.4 Categorization Based on Usage

DBMSs can also be categorized based on usage. In what follows, we discuss operational versus strategic usage, Big Data and analytics, multimedia DBMSs, spatial DBMSs, sensor DBMSs, mobile DBMSs, and open-source DBMSs.

On-line transaction processing (OLTP) DBMSs focus on managing operational or transactional data. Think of a point-of-sale (POS) application in a supermarket, where data about each purchase transaction such as customer information, products purchased, prices paid, location of the purchase, and timing of the purchase need to be stored. In these settings, the database server must be able to process lots of simple transactions per unit of time. Also, the transactions are initiated in real-time, simultaneously, by many users and applications, hence the DBMS must have good support for processing a high volume of short, simple queries. **On-line analytical processing (OLAP) DBMSs** focus on using operational data for tactical or strategical decision-making. Here, a limited number of users formulates complex queries to analyze huge amounts of data. The DBMS should support the efficient processing of these complex queries, which often come in smaller volumes.

Big data and analytics are all around these days (see Chapters 19 and 20). IBM projects that we generate 2.5 quintillion bytes of data every day. This is a lot compared to traditional database applications. Hence, new database technologies have been introduced to efficiently cope with Big Data. NoSQL is one of these newer technologies. **NoSQL databases** abandon the well-known and popular relational database schema in favor of a more flexible, or even schema-less, database structure. This is especially handy to store unstructured information such as emails, text documents, Twitter tweets, Facebook posts, etc. One of their key advantages is that they also scale more easily in terms of storage capacity. We already mentioned four popular types of NoSQL database technologies, classified according to data model: key–value-based databases such as CouchDB; document-based databases such as MongoDB; column-based databases such as Cassandra; and graph-based databases such as Neo4j. We discuss these in more detail in Chapter 11.

Multimedia DBMSs allow for the storage of multimedia data such as text, images, audio, video, 3D games, CAD designs, etc. They should also provide content-based query facilities such as "find images of Bart" or "find images of people who look like Bart". Streaming facilities should also be included to stream multimedia output. These are very resource-intensive transactions that may require specific hardware support. Note that multimedia data are usually stored as a binary large object (BLOB), supported by most modern-day commercial DBMSs.

A **spatial DBMS** supports the storage and querying of spatial data. This could include both 2D objects (e.g., points, lines, and polygons) and 3D objects. Spatial operations such as calculating distances or relationships between objects (e.g., whether one object is contained within another, intersects with another, is detached from another, etc.) are provided. Spatial databases are a key building block of geographical information systems (GIS). Most commercial DBMS vendors offer facilities for spatial data management.

A **sensor DBMS** manages sensor data such as biometric data obtained from wearables, or telematics data which continuously record driving behavior. Ideally, it has facilities to formulate application-specific queries such as spatial–temporal queries that ask for the shortest path between two locations given the current state of the traffic. Most modern-day DBMSs provide support for storing sensor data.

Mobile DBMSs are the DBMSs running on smartphones, tablets, and other mobile devices. They should always be online, have a small footprint, and be able to deal with limited processing power, storage, and battery life. Depending upon the context, they could connect and synchronize to a central DBMS. Ideally, they should be capable of handling queries and support self-management without the intervention of a DBA. Some popular examples are: Oracle Lite, Sybase SQL Anywhere, Microsoft SQL Server Compact, SQLite, and IBM DB2 Everyplace.

Finally, **open-source DBMSs** are DBMSs for which the code is publicly available and can be extended by anyone. This has the advantage of having a large development community working on the product. They are very popular for small business settings and in developing countries where budgets are limited. Most of the open-source DBMSs can be obtained from www.sourceforge.net, which is a well-known website for open-source software. Some examples are: MySQL, which is a relational DBMS maintained by Oracle; PostgresSQL, which is also relational and maintained by the PostgresSQL Global Development Group; Twig, which is an object-oriented DBMS maintained by Google; and Perst, which is also an OODBMS maintained by McObject.

Drill Down

Spotify streams more than 24 million songs to more than 40 million users worldwide. It needed a database solution which ensures data availability at all times, even in the event of crashes or bugs. It turned to Apache Cassandra as the database technology of choice since its cloud-based architecture ensures high availability.

Drill Down

Gartner[1] estimates that by 2018 more than 70% of new applications will be developed using open-source DBMSs. This clearly illustrates that open-source solutions have significantly matured into viable and robust alternatives to their commercial counterparts.

Retention Questions

- How can DBMSs be categorized based on data model?
- How can DBMSs be categorized based on usage?