Chapter 1

The Worlds of Database Systems

Databases today are essential to every business. Whenever you visit a major Web site — Google, Yahoo!, Amazon.com, or thousands of smaller sites that provide information — there is a database behind the scenes serving up the information you request. Corporations maintain all their important records in databases. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators of the human genome, and by biochemists exploring properties of proteins, among many other scientific activities.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a *database management system*, or *DBMS*, or more colloquially a "database system." A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely. These systems are among the most complex types of software available. In this book, we shall learn how to design databases, how to write programs in the various languages associated with a DBMS, and how to implement the DBMS itself.

1.1 The Evolution of Database Systems

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

1.1.1 Early Database Management Systems

The first commercial database management systems appeared in the late 1960's. These systems evolved from file systems, which provide some of item (3) above; file systems store data over a long period of time, and they allow the storage of large amounts of data. However, file systems do not generally guarantee that data cannot be lost if it is not backed up, and they don't support efficient access to data items whose location in a particular file is not known.

Further, file systems do not directly support item (2), a query language for the data in files. Their support for (1) — a schema for the data — is limited to the creation of directory structures for files. Item (4) is not always supported by file systems; you can lose data that has not been backed up. Finally, file systems do not satisfy (5). While they allow concurrent access to files by several users or processes, a file system generally will not prevent situations such as two users modifying the same file at about the same time, so the changes made by one user fail to appear in the file.

The first important applications of DBMS's were ones where data was composed of many small items, and many queries or modifications were made. Examples of these applications are:

1. Banking systems: maintaining accounts and making sure that system failures do not cause money to disappear.

2. Airline reservation systems: these, like banking systems, require assurance that data will not be lost, and they must accept very large volumes of small actions by customers.

3. Corporate record keeping: employment and tax records, inventories, sales records, and a great variety of other types of information, much of it critical.

The early DBMS's required the programmer to visualize data much as it was stored. These database systems used several different data models for

describing the structure of the information in a database, chief among them the "hierarchical" or tree-based model and the graph-based "network" model. The latter was standardized in the late 1960's through a report of CODASYL (Committee on Data Systems and Languages).[1]

A problem with these early models and systems was that they did not support high-level query languages. For example, the CODASYL query language had statements that allowed the user to jump from data element to data element, through a graph of pointers among these elements. There was considerable effort needed to write such programs, even for very simple queries.

1.1.2 Relational Database Systems

Following a famous paper written by Ted Codd in 1970,[2] database systems changed significantly. Codd proposed that database systems should present the user with a view of data organized as tables called *relations*. Behind the scenes, there might be a complex data structure that allowed rapid response to a variety of queries. But, unlike the programmers for earlier database systems, the programmer of a relational system would not be concerned with the storage structure. Queries could be expressed in a very high-level language, which greatly increased the efficiency of database programmers. We shall cover the relational model of database systems throughout most of this book. SQL ("Structured Query Language"), the most important query language based on the relational model, is covered extensively.

By 1990, relational database systems were the norm. Yet the database field continues to evolve, and new issues and approaches to the management of data surface regularly. Object-oriented features have infilrated the relational model. Some of the largest databases are organized rather differently from those using relational methodology. In the balance of this section, we shall consider some of the modern trends in database systems.

1.1.3 Smaller and Smaller Systems

Originally, DBMS's were large, expensive software systems running on large computers. The size was necessary, because to store a gigabyte of data required a large computer system. Today, hundreds of gigabytes fit on a single disk, and it is quite feasible to run a DBMS on a personal computer. Thus, database systems based on the relational model have become available for even very small machines, and they are beginning to appear as a common tool for computer applications, much as spreadsheets and word processors did before them.

Another important trend is the use of documents, often tagged using XML (eXtensible Modeling Language). Large collections of small documents can

[1] *CODASYL Data Base Task Group April 1971 Report*, ACM, New York.

[2] Codd, E. F., "A relational model for large shared data banks," *Comm. ACM*, **13**:6, pp. 377–387, 1970.

serve as a database, and the methods of querying and manipulating them are different from those used in relational systems.

1.1.4 Bigger and Bigger Systems

On the other hand, a gigabyte is not that much data any more. Corporate databases routinely store terabytes (10^{12} bytes). Yet there are many databases that store petabytes (10^{15} bytes) of data and serve it all to users. Some important examples:

1. Google holds petabytes of data gleaned from its crawl of the Web. This data is not held in a traditional DBMS, but in specialized structures optimized for search-engine queries.

2. Satellites send down petabytes of information for storage in specialized systems.

3. A picture is actually worth way more than a thousand words. You can store 1000 words in five or six thousand bytes. Storing a picture typically takes much more space. Repositories such as Flickr store millions of pictures and support search of those pictures. Even a database like Amazon's has millions of pictures of products to serve.

4. And if still pictures consume space, movies consume much more. An hour of video requires at least a gigabyte. Sites such as YouTube hold hundreds of thousands, or millions, of movies and make them available easily.

5. Peer-to-peer file-sharing systems use large networks of conventional computers to store and distribute data of various kinds. Although each node in the network may only store a few hundred gigabytes, together the database they embody is enormous.

1.1.5 Information Integration

To a great extent, the old problem of building and maintaining databases has become one of *information integration*: joining the information contained in many related databases into a whole. For example, a large company has many divisions. Each division may have built its own database of products or employee records independently of other divisions. Perhaps some of these divisions used to be independent companies, which naturally had their own way of doing things. These divisions may use different DBMS's and different structures for information. They may use different terms to mean the same thing or the same term to mean different things. To make matters worse, the existence of legacy applications using each of these databases makes it almost impossible to scrap them, ever.

As a result, it has become necessary with increasing frequency to build structures on top of existing databases, with the goal of integrating the information

distributed among them. One popular approach is the creation of *data warehouses*, where information from many legacy databases is copied periodically, with the appropriate translation, to a central database. Another approach is the implementation of a mediator, or "middleware," whose function is to support an integrated model of the data of the various databases, while translating between this model and the actual models used by each database.

1.2 Overview of a Database Management System

In Fig. 1.1 we see an outline of a complete DBMS. Single boxes represent system components, while double boxes represent in-memory data structures. The solid lines indicate control and data flow, while dashed lines indicate data flow only. Since the diagram is complicated, we shall consider the details in several stages. First, at the top, we suggest that there are two distinct sources of commands to the DBMS:

1. Conventional users and application programs that ask for data or modify data.
2. A *database administrator*: a person or persons responsible for the structure or *schema* of the database.

1.2.1 Data-Definition Language Commands

The second kind of command is the simpler to process, and we show its trail beginning at the upper right side of Fig. 1.1. For example, the database administrator, or *DBA*, for a university registrar's database might decide that there should be a table or relation with columns for a student, a course the student has taken, and a grade for that student in that course. The DBA might also decide that the only allowable grades are A, B, C, D, and F. This structure and constraint information is all part of the schema of the database. It is shown in Fig. 1.1 as entered by the DBA, who needs special authority to execute schema-altering commands, since these can have profound effects on the database. These schema-altering data-definition language (DDL) commands are parsed by a DDL processor and passed to the execution engine, which then goes through the index/file/record manager to alter the *metadata*, that is, the schema information for the database.

1.2.2 Overview of Query Processing

The great majority of interactions with the DBMS follow the path on the left side of Fig. 1.1. A user or an application program initiates some action, using the data-manipulation language (DML). This command does not affect the schema of the database, but may affect the content of the database (if the

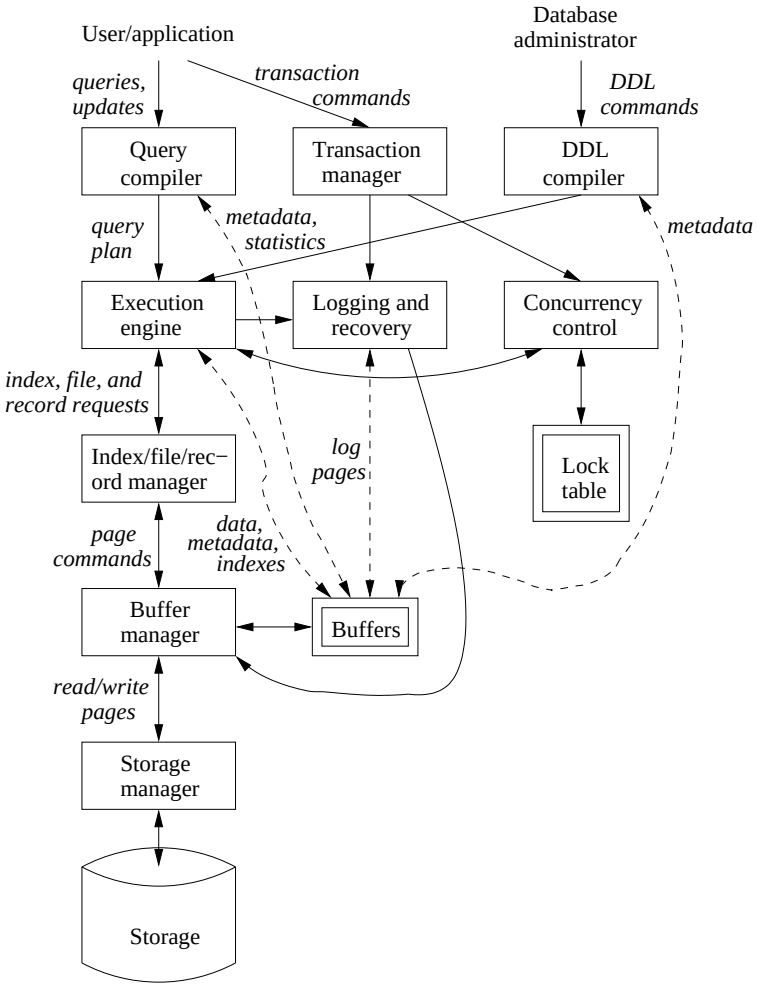

Figure 1.1: Database management system components

action is a modification command) or will extract data from the database (if the action is a query). DML statements are handled by two separate subsystems, as follows.

Answering the Query

The query is parsed and optimized by a *query compiler*. The resulting *query plan*, or sequence of actions the DBMS will perform to answer the query, is passed to the *execution engine*. The execution engine issues a sequence of requests for small pieces of data, typically records or tuples of a relation, to a resource manager that knows about *data files* (holding relations), the format and size of records in those files, and *index files*, which help find elements of data files quickly.

The requests for data are passed to the *buffer manager*. The buffer manager's task is to bring appropriate portions of the data from secondary storage (disk) where it is kept permanently, to the main-memory buffers. Normally, the page or "disk block" is the unit of transfer between buffers and disk.

The buffer manager communicates with a storage manager to get data from disk. The storage manager might involve operating-system commands, but more typically, the DBMS issues commands directly to the disk controller.

Transaction Processing

Queries and other DML actions are grouped into *transactions*, which are units that must be executed atomically and in isolation from one another. Any query or modification action can be a transaction by itself. In addition, the execution of transactions must be *durable*, meaning that the effect of any completed transaction must be preserved even if the system fails in some way right after completion of the transaction. We divide the transaction processor into two major parts:

1. A *concurrency-control manager*, or *scheduler*, responsible for assuring atomicity and isolation of transactions, and

2. A *logging and recovery manager*, responsible for the durability of transactions.

1.2.3 Storage and Buffer Management

The data of a database normally resides in secondary storage; in today's computer systems "secondary storage" generally means magnetic disk. However, to perform any useful operation on data, that data must be in main memory. It is the job of the *storage manager* to control the placement of data on disk and its movement between disk and main memory.

In a simple database system, the storage manager might be nothing more than the file system of the underlying operating system. However, for efficiency

purposes, DBMS's normally control storage on the disk directly, at least under some circumstances. The *storage manager* keeps track of the location of files on the disk and obtains the block or blocks containing a file on request from the buffer manager.

The *buffer manager* is responsible for partitioning the available main memory into *buffers*, which are page-sized regions into which disk blocks can be transferred. Thus, all DBMS components that need information from the disk will interact with the buffers and the buffer manager, either directly or through the execution engine. The kinds of information that various components may need include:

1. *Data*: the contents of the database itself.
2. *Metadata*: the database schema that describes the structure of, and constraints on, the database.
3. *Log Records*: information about recent changes to the database; these support durability of the database.
4. *Statistics*: information gathered and stored by the DBMS about data properties such as the sizes of, and values in, various relations or other components of the database.
5. *Indexes*: data structures that support efficient access to the data.

1.2.4 Transaction Processing

It is normal to group one or more database operations into a *transaction*, which is a unit of work that must be executed atomically and in apparent isolation from other transactions. In addition, a DBMS offers the guarantee of durability: that the work of a completed transaction will never be lost. The *transaction manager* therefore accepts *transaction commands* from an application, which tell the transaction manager when transactions begin and end, as well as information about the expectations of the application (some may not wish to require atomicity, for example). The transaction processor performs the following tasks:

1. *Logging*: In order to assure durability, every change in the database is logged separately on disk. The *log manager* follows one of several policies designed to assure that no matter when a system failure or "crash" occurs, a *recovery manager* will be able to examine the log of changes and restore the database to some consistent state. The log manager initially writes the log in buffers and negotiates with the buffer manager to make sure that buffers are written to disk (where data can survive a crash) at appropriate times.
2. *Concurrency control*: Transactions must appear to execute in isolation. But in most systems, there will in truth be many transactions executing

The ACID Properties of Transactions

Properly implemented transactions are commonly said to meet the "ACID test," where:

- "A" stands for "atomicity," the all-or-nothing execution of transactions.
- "I" stands for "isolation," the fact that each transaction must appear to be executed as if no other transaction is executing at the same time.
- "D" stands for "durability," the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

The remaining letter, "C," stands for "consistency." That is, all databases have consistency constraints, or expectations about relationships among data elements (e.g., account balances may not be negative after a transaction finishes). Transactions are expected to preserve the consistency of the database.

at once. Thus, the scheduler (concurrency-control manager) must assure that the individual actions of multiple transactions are executed in such an order that the net effect is the same as if the transactions had in fact executed in their entirety, one-at-a-time. A typical scheduler does its work by maintaining *locks* on certain pieces of the database. These locks prevent two transactions from accessing the same piece of data in ways that interact badly. Locks are generally stored in a main-memory *lock table*, as suggested by Fig. 1.1. The scheduler affects the execution of queries and other database operations by forbidding the execution engine from accessing locked parts of the database.

3. *Deadlock resolution*: As transactions compete for resources through the locks that the scheduler grants, they can get into a situation where none can proceed because each needs something another transaction has. The transaction manager has the responsibility to intervene and cancel ("rollback" or "abort") one or more transactions to let the others proceed.

1.2.5 The Query Processor

The portion of the DBMS that most affects the performance that the user sees is the *query processor*. In Fig. 1.1 the query processor is represented by two components:

1. The *query compiler*, which translates the query into an internal form called a *query plan*. The latter is a sequence of operations to be performed on the data. Often the operations in a query plan are implementations of "relational algebra" operations, which are discussed in Section 2.4. The query compiler consists of three major units:

   (a) A *query parser*, which builds a tree structure from the textual form of the query.

   (b) A *query preprocessor*, which performs semantic checks on the query (e.g., making sure all relations mentioned by the query actually exist), and performing some tree transformations to turn the parse tree into a tree of algebraic operators representing the initial query plan.

   (c) A *query optimizer*, which transforms the initial query plan into the best available sequence of operations on the actual data.

   The query compiler uses metadata and statistics about the data to decide which sequence of operations is likely to be the fastest. For example, the existence of an *index*, which is a specialized data structure that facilitates access to data, given values for one or more components of that data, can make one plan much faster than another.

2. The *execution engine*, which has the responsibility for executing each of the steps in the chosen query plan. The execution engine interacts with most of the other components of the DBMS, either directly or through the buffers. It must get the data from the database into buffers in order to manipulate that data. It needs to interact with the scheduler to avoid accessing data that is locked, and with the log manager to make sure that all database changes are properly logged.

1.3 Outline of Database-System Studies

We divide the study of databases into five parts. This section is an outline of what to expect in each of these units.

Part I: Relational Database Modeling

The relational model is essential for a study of database systems. After examining the basic concepts, we delve into the theory of relational databases. That study includes *functional dependencies*, a formal way of stating that one kind of data is uniquely determined by another. It also includes *normalization*, the process whereby functional dependencies and other formal dependencies are used to improve the design of a relational database.

We also consider high-level design notations. These mechanisms include the Entity-Relationship (E/R) model, Unified Modeling Language (UML), and Object Definition Language (ODL). Their purpose is to allow informal exploration of design issues before we implement the design using a relational DBMS.

Part II: Relational Database Programming

We then take up the matter of how relational databases are queried and modified. After an introduction to abstract programming languages based on algebra and logic (Relational Algebra and Datalog, respectively), we turn our attention to the standard language for relational databases: SQL. We study both the basics and important special topics, including constraint specifications and triggers (active database elements), indexes and other structures to enhance performance, forming SQL into transactions, and security and privacy of data in SQL.

We also discuss how SQL is used in complete systems. It is typical to combine SQL with a conventional or *host* language and to pass data between the database and the conventional program via SQL calls. We discuss a number of ways to make this connection, including embedded SQL, Persistent Stored Modules (PSM), Call-Level Interface (CLI), Java Database Interconnectivity (JDBC), and PHP.

Part III: Semistructured Data Modeling and Programming

The pervasiveness of the Web has put a premium on the management of hierarchically structured data, because the standards for the Web are based on nested, tagged elements (*semistructured data*). We introduce XML and its schema-defining notations: Document Type Definitions (DTD) and XML Schema. We also examine three query languages for XML: XPATH, XQuery, and Extensible Stylesheet Language Transform (XSLT).

Part IV: Database System Implementation

We begin with a study of *storage management*: how disk-based storage can be organized to allow efficient access to data. We explain the commonly used B-tree, a balanced tree of disk blocks and other specialized schemes for managing multidimensional data.

We then turn our attention to *query processing*. There are two parts to this study. First, we need to learn *query execution*: the algorithms used to implement the operations from which queries are built. Since data is typically on disk, the algorithms are somewhat different from what one would expect were they to study the same problems but assuming that data were in main memory. The second step is *query compiling*. Here, we study how to select an efficient query plan from among all the possible ways in which a given query can be executed.

Then, we study *transaction processing*. There are several threads to follow. One concerns *logging*: maintaining reliable records of what the DBMS is doing, in order to allow *recovery* in the event of a crash. Another thread is *scheduling*: controlling the order of events in transactions to assure the ACID properties. We also consider how to deal with deadlocks, and the modifications to our algorithms that are needed when a transaction is *distributed* over many independent

sites.

Part V: Modern Database System Issues

In this part, we take up a number of the ways in which database-system technology is relevant beyond the realm of conventional, relational DBMS's. We consider how *search engines* work, and the specialized data structures that make their operation possible. We look at information integration, and methodologies for making databases share their data seamlessly. *Data mining* is a study that includes a number of interesting and important algorithms for processing large amounts of data in complex ways. *Data-stream systems* deal with data that arrives at the system continuously, and whose queries are answered continuously and in a timely fashion. *Peer-to-peer systems* present many challenges for management of distributed data held by independent hosts.

1.4 References for Chapter 1

Today, on-line searchable bibliographies cover essentially all recent papers concerning database systems. Thus, in this book, we shall not try to be exhaustive in our citations, but rather shall mention only the papers of historical importance and major secondary sources or useful surveys. A searchable index of database research papers was constructed by Michael Ley [5], and has recently been expanded to include references from many fields. Alf-Christian Achilles maintains a searchable directory of many indexes relevant to the database field [3].

While many prototype implementations of database systems contributed to the technology of the field, two of the most widely known are the System R project at IBM Almaden Research Center [4] and the INGRES project at Berkeley [7]. Each was an early relational system and helped establish this type of system as the dominant database technology. Many of the research papers that shaped the database field are found in [6].

The 2003 "Lowell report" [1] is the most recent in a series of reports on database-system research and directions. It also has references to earlier reports of this type.

You can find more about the theory of database systems than is covered here from [2] and [8].

1. S. Abiteboul et al., "The Lowell database research self-assessment," *Comm. ACM* **48**:5 (2005), pp. 111–118. `http://research.microsoft.com/~gray/lowell/LowellDatabaseResearchSelfAssessment.htm`

2. S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.

3. `http://liinwww.ira.uka.de/bibliography/Database` .

4. M. M. Astrahan et al., "System R: a relational approach to database management," *ACM Trans. on Database Systems* **1**:2, pp. 97–137, 1976.

5. `http://www.informatik.uni-trier.de/~ley/db/index.html` . A mirror site is found at `http://www.acm.org/sigmod/dblp/db/index.html` .

6. M. Stonebraker and J. M. Hellerstein (eds.), *Readings in Database Systems*, Morgan-Kaufmann, San Francisco, 1998.

7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Trans. on Database Systems* **1**:3, pp. 189–222, 1976.

8. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volumes I and II*, Computer Science Press, New York, 1988, 1989.