Chapter 1

The Worlds of Database Systems

Databases today are essential to every business. They are used to maintain internal records, to present data to customers and clients on the World-Wide-Web, and to support many other commercial processes. 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 the medicinal properties of proteins, along with many other scientists.

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. The capabilities that a DBMS provides the user are:

1. *Persistent storage.* Like a file system, a DBMS supports the storage of very large amounts of data that exists independently of any processes that are using the data. However, the DBMS goes far beyond the file system in providing flexibility, such as data structures that support efficient access to very large amounts of data.

2. *Programming interface.* A DBMS allows the user or an application program to access and modify data through a powerful query language. Again, the advantage of a DBMS over a file system is the flexibility to manipulate stored data in much more complex ways than the reading and writing of files.

3. *Transaction management.* A DBMS supports concurrent access to data, i.e., simultaneous access by many distinct processes (called "transac-

tions") at once. To avoid some of the undesirable consequences of simultaneous access, the DBMS supports *isolation*, the appearance that transactions execute one-at-a-time, and *atomicity*, the requirement that transactions execute either completely or not at all. A DBMS also supports *durability*, the ability to recover from failures or errors of many types.

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 *schema* (logical structure of the data), using a specialized language called a *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 gigabytes or more — over a long period of time, keeping it secure from accident or unauthorized use and allowing efficient access to the data for queries and database modifications.

4. Control access to data from many users at once, without allowing the actions of one user to affect other users and without allowing simultaneous accesses to corrupt the data accidentally.

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. Finally, file systems do not satisfy (4). When 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. Here are some of these applications.

Airline Reservations Systems

In this type of system, the items of data include:

1. Reservations by a single customer on a single flight, including such information as assigned seat or meal preference.

2. Information about flights — the airports they fly from and to, their departure and arrival times, or the aircraft flown, for example.

3. Information about ticket prices, requirements, and availability.

Typical queries ask for flights leaving around a certain time from one given city to another, what seats are available, and at what prices. Typical data modifications include the booking of a flight for a customer, assigning a seat, or indicating a meal preference. Many agents will be accessing parts of the data at any given time. The DBMS must allow such concurrent accesses, prevent problems such as two agents assigning the same seat simultaneously, and protect against loss of records if the system suddenly fails.

Banking Systems

Data items include names and addresses of customers, accounts, loans, and their balances, and the connection between customers and their accounts and loans, e.g., who has signature authority over which accounts. Queries for account balances are common, but far more common are modifications representing a single payment from, or deposit to, an account.

As with the airline reservation system, we expect that many tellers and customers (through ATM machines or the Web) will be querying and modifying the bank's data at once. It is vital that simultaneous accesses to an account not cause the effect of a transaction to be lost. Failures cannot be tolerated. For example, once the money has been ejected from an ATM machine, the bank must record the debit, even if the power immediately fails. On the other hand, it is not permissible for the bank to record the debit and then not deliver the money if the power fails. The proper way to handle this operation is far from obvious and can be regarded as one of the significant achievements in DBMS architecture.

Corporate Records

Many early applications concerned corporate records, such as a record of each sale, information about accounts payable and receivable, or information about employees — their names, addresses, salary, benefit options, tax status, and so on. Queries include the printing of reports such as accounts receivable or employees' weekly paychecks. Each sale, purchase, bill, receipt, employee hired, fired, or promoted, and so on, results in a modification to the database.

The early DBMS's, evolving from file systems, encouraged the user 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 user of earlier database systems, the user 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, starting with the basic relational concepts in Chapter 3. SQL ("Structured Query Language"), the most important query language based on the relational model, will be covered starting in Chapter 6. However, a brief introduction to relations will give the reader a hint of the simplicity of the model, and an SQL sample will suggest how the relational model promotes queries written at a very high level, avoiding details of "navigation" through the database.

Example 1.1: Relations are tables. Their columns are headed by *attributes*, which describe the entries in the column. For instance, a relation named `Accounts`, recording bank accounts, their balance, and type might look like:

accountNo	*balance*	*type*
12345	1000.00	savings
67890	2846.92	checking
...	...	...

[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.

Heading the columns are the three attributes: `accountNo`, `balance`, and `type`. Below the attributes are the rows, or *tuples*. Here we show two tuples of the relation explicitly, and the dots below them suggest that there would be many more tuples, one for each account at the bank. The first tuple says that account number 12345 has a balance of one thousand dollars, and it is a savings account. The second tuple says that account 67890 is a checking account with $2846.92.

Suppose we wanted to know the balance of account 67890. We could ask this query in SQL as follows:

```
SELECT balance
FROM Accounts
WHERE accountNo = 67890;
```

For another example, we could ask for the savings accounts with negative balances by:

```
SELECT accountNo
FROM Accounts
WHERE type = 'savings' AND balance < 0;
```

We do not expect that these two examples are enough to make the reader an expert SQL programmer, but they should convey the high-level nature of the SQL "select-from-where" statement. In principle, they ask the DBMS to

1. Examine all the tuples of the relation `Accounts` mentioned in the `FROM` clause,

2. Pick out those tuples that satisfy some criterion indicated in the `WHERE` clause, and

3. Produce as an answer certain attributes of those tuples, as indicated in the `SELECT` clause.

In practice, the system must "optimize" the query and find an efficient way to answer the query, even though the relations involved in the query may be very large. □

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. 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, many 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.

1.1.4 Bigger and Bigger Systems

On the other hand, a gigabyte isn't much data. Corporate databases often occupy hundreds of gigabytes. Further, as storage becomes cheaper people find new reasons to store greater amounts of data. For example, retail chains often store *terabytes* (a terabyte is 1000 gigabytes, or 10^{12} bytes) of information recording the history of every sale made over a long period of time (for planning inventory; we shall have more to say about this matter in Section 1.1.7).

Further, databases no longer focus on storing simple data items such as integers or short character strings. They can store images, audio, video, and many other kinds of data that take comparatively huge amounts of space. For instance, an hour of video consumes about a gigabyte. Databases storing images from satellites can involve *petabytes* (1000 terabytes, or 10^{15} bytes) of data.

Handling such large databases required several technological advances. For example, databases of modest size are today stored on arrays of disks, which are called *secondary storage devices* (compared to main memory, which is "primary" storage). One could even argue that what distinguishes database systems from other software is, more than anything else, the fact that database systems routinely assume data is too big to fit in main memory and must be located primarily on disk at all times. The following two trends allow database systems to deal with larger amounts of data, faster.

Tertiary Storage

The largest databases today require more than disks. Several kinds of *tertiary storage devices* have been developed. Tertiary devices, perhaps storing a terabyte each, require much more time to access a given item than does a disk. While typical disks can access any item in 10-20 milliseconds, a tertiary device may take several seconds. Tertiary storage devices involve transporting an object, upon which the desired data item is stored, to a reading device. This movement is performed by a robotic conveyance of some sort.

For example, compact disks (CD's) or digital versatile disks (DVD's) may be the storage medium in a tertiary device. An arm mounted on a track goes to a particular disk, picks it up, carries it to a reader, and loads the disk into the reader.

Parallel Computing

The ability to store enormous volumes of data is important, but it would be of little use if we could not access large amounts of that data quickly. Thus, very large databases also require speed enhancers. One important speedup is

through index structures, which we shall mention in Section 1.2.2 and cover extensively in Chapter 13. Another way to process more data in a given time is to use parallelism. This parallelism manifests itself in various ways.

For example, since the rate at which data can be read from a given disk is fairly low, a few megabytes per second, we can speed processing if we use many disks and read them in parallel (even if the data originates on tertiary storage, it is "cached" on disks before being accessed by the DBMS). These disks may be part of an organized parallel machine, or they may be components of a distributed system, in which many machines, each responsible for a part of the database, communicate over a high-speed network when needed.

Of course, the ability to move data quickly, like the ability to store large amounts of data, does not by itself guarantee that queries can be answered quickly. We still need to use algorithms that break queries up in ways that allow parallel computers or networks of distributed computers to make effective use of all the resources. Thus, parallel and distributed management of very large databases remains an active area of research and development; we consider some of its important ideas in Section 15.9.

1.1.5 Client-Server and Multi-Tier Architectures

Many varieties of modern software use a *client-server* architecture, in which requests by one process (the *client*) are sent to another process (the *server*) for execution. Database systems are no exception, and it has become increasingly common to divide the work of a DBMS into a server process and one or more client processes.

In the simplest client-server architecture, the entire DBMS is a server, except for the query interfaces that interact with the user and send queries or other commands across to the server. For example, relational systems generally use the SQL language for representing requests from the client to the server. The database server then sends the answer, in the form of a table or relation, back to the client. The relationship between client and server can get more complex, especially when answers are extremely large. We shall have more to say about this matter in Section 1.1.6.

There is also a trend to put more work in the client, since the server will be a bottleneck if there are many simultaneous database users. In the recent proliferation of system architectures in which databases are used to provide dynamically-generated content for Web sites, the two-tier (client-server) architecture gives way to three (or even more) tiers. The DBMS continues to act as a server, but its client is typically an *application server*, which manages connections to the database, transactions, authorization, and other aspects. Application servers in turn have clients such as Web servers, which support end-users or other applications.

1.1.6 Multimedia Data

Another important trend in database systems is the inclusion of multimedia data. By "multimedia" we mean information that represents a signal of some sort. Common forms of multimedia data include video, audio, radar signals, satellite images, and documents or pictures in various encodings. These forms have in common that they are much larger than the earlier forms of data — integers, character strings of fixed length, and so on — and of vastly varying sizes.

The storage of multimedia data has forced DBMS's to expand in several ways. For example, the operations that one performs on multimedia data are not the simple ones suitable for traditional data forms. Thus, while one might search a bank database for accounts that have a negative balance, comparing each balance with the real number 0.0, it is not feasible to search a database of pictures for those that show a face that "looks like" a particular image.

To allow users to create and use complex data operations such as image-processing, DBMS's have had to incorporate the ability of users to introduce functions of their own choosing. Often, the object-oriented approach is used for such extensions, even in relational systems, which are then dubbed "object-relational." We shall take up object-oriented database programming in various places, including Chapters 4 and 9.

The size of multimedia objects also forces the DBMS to modify the storage manager so that objects or tuples of a gigabyte or more can be accommodated. Among the many problems that such large elements present is the delivery of answers to queries. In a conventional, relational database, an answer is a set of tuples. These tuples would be delivered to the client by the database server as a whole.

However, suppose the answer to a query is a video clip a gigabyte long. It is not feasible for the server to deliver the gigabyte to the client as a whole. For one reason it takes too long and will prevent the server from handling other requests. For another, the client may want only a small part of the film clip, but doesn't have a way to ask for exactly what it wants without seeing the initial portion of the clip. For a third reason, even if the client wants the whole clip, perhaps in order to play it on a screen, it is sufficient to deliver the clip at a fixed rate over the course of an hour (the amount of time it takes to play a gigabyte of compressed video). Thus, the storage system of a DBMS supporting multimedia data has to be prepared to deliver answers in an interactive mode, passing a piece of the answer to the client on request or at a fixed rate.

1.1.7 Information Integration

As information becomes ever more essential in our work and play, we find that existing information resources are being used in many new ways. For instance, consider a company that wants to provide on-line catalogs for all its products, so that people can use the World Wide Web to browse its products and place on-

line orders. A large company has many divisions. Each division may have built its own database of products independently of other divisions. These divisions may use different DBMS's, different structures for information, perhaps even different terms to mean the same thing or the same term to mean different things.

Example 1.2: Imagine a company with several divisions that manufacture disks. One division's catalog might represent rotation rate in revolutions per second, another in revolutions per minute. Another might have neglected to represent rotation speed at all. A division manufacturing floppy disks might refer to them as "disks," while a division manufacturing hard disks might call *them* "disks" as well. The number of tracks on a disk might be referred to as "tracks" in one division, but "cylinders" in another. □

Central control is not always the answer. Divisions may have invested large amounts of money in their database long before information integration across divisions was recognized as a problem. A division may have been an independent company, recently acquired. For these or other reasons, these so-called *legacy databases* cannot be replaced easily. Thus, the company must build some structure on top of the legacy databases to present to customers a unified view of products across the company.

One popular approach is the creation of *data warehouses*, where information from many legacy databases is copied, with the appropriate translation, to a central database. As the legacy databases change, the warehouse is updated, but not necessarily instantaneously updated. A common scheme is for the warehouse to be reconstructed each night, when the legacy databases are likely to be less busy.

The legacy databases are thus able to continue serving the purposes for which they were created. New functions, such as providing an on-line catalog service through the Web, are done at the data warehouse. We also see data warehouses serving needs for planning and analysis. For example, company analysts may run queries against the warehouse looking for sales trends, in order to better plan inventory and production. *Data mining*, the search for interesting and unusual patterns in data, has also been enabled by the construction of data warehouses, and there are claims of enhanced sales through exploitation of patterns discovered in this way. These and other issues of information integration are discussed in Chapter 20.

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 *DDL commands* ("DDL" stands for "data-definition language") 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 that does not affect the schema of the database, but may affect the content of the database (if the action is a modification command) or will extract data from the database (if the action is a query). Remember from Section 1.1 that the language in which these commands are expressed is called a data-manipulation language (*DML*) or somewhat colloquially a query language. There are many data-manipulation languages available, but SQL, which was mentioned in Example 1.1, is by far the most commonly used. 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

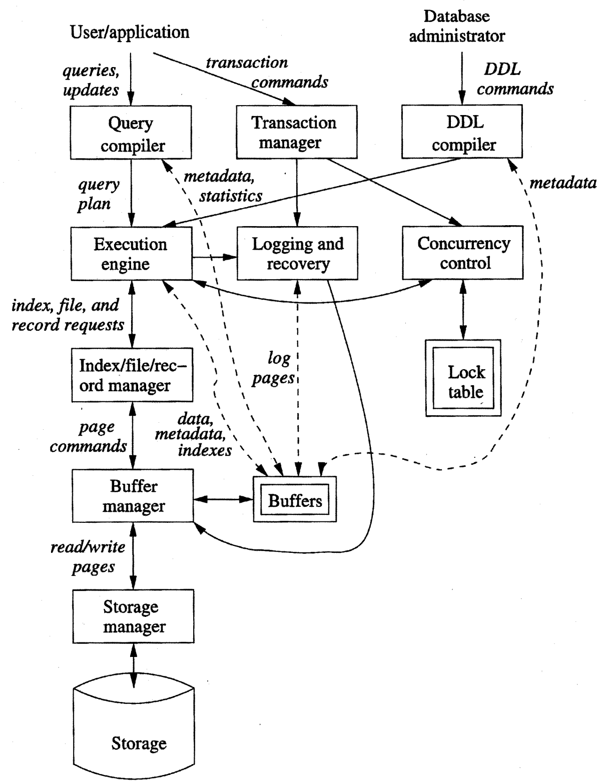

Figure 1.1: Database management system components

and size of records in those files, and *index files*, which help find elements of data files quickly.

The requests for data are translated into pages and these requests are passed to the *buffer manager*. We shall discuss the role of the buffer manager in Section 1.2.3, but briefly, its task is to bring appropriate portions of the data from secondary storage (disk, normally) where it is kept permanently, to 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. Often each query or modification action is 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.

We shall consider these components further in Section 1.2.4.

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. Recall that disks are generally divided into *disk blocks*, which are regions of contiguous storage containing a large number of bytes, perhaps 2^{12} or 2^{14} (about 4000 to 16,000 bytes).

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. *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.
4. *Indexes*: data structures that support efficient access to the data.

A more complete discussion of the buffer manager and its role appears in Section 15.7.

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 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

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). Transactions are expected to preserve the consistency of the database. We discuss the expression of consistency constraints in a database schema in Chapter 7, while Section 18.1 begins a discussion of how consistency is maintained by the DBMS.

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 5.2. 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

Ideas related to database systems can be divided into three broad categories:

1. *Design of databases.* How does one develop a useful database? What kinds of information go into the database? How is the information structured? What assumptions are made about types or values of data items? How do data items connect?
2. *Database programming.* How does one express queries and other operations on the database? How does one use other capabilities of a DBMS, such as transactions or constraints, in an application? How is database programming combined with conventional programming?
3. *Database system implementation.* How does one build a DBMS, including such matters as query processing, transaction processing and organizing storage for efficient access?

How Indexes Are Implemented

The reader may have learned in a course on data structures that a hash table is a very efficient way to build an index. Early DBMS's did use hash tables extensively. Today, the most common data structure is called a *B-tree*; the "B" stands for "balanced." A B-tree is a generalization of a balanced binary search tree. However, while each node of a binary tree has up to two children, the B-tree nodes have a large number of children. Given that B-trees normally reside on disk rather than in main memory, the B-tree is designed so that each node occupies a full disk block. Since typical systems use disk blocks on the order of 2^{12} bytes (4096 bytes), there can be hundreds of pointers to children in a single block of a B-tree. Thus, search of a B-tree rarely involves more than a few levels.

The true cost of disk operations generally is proportional to the number of disk blocks accessed. Thus, searches of a B-tree, which typically examine only a few disk blocks, are much more efficient than would be a binary-tree search, which typically visits nodes found on many different disk blocks. This distinction, between B-trees and binary search trees, is but one of many examples where the most appropriate data structure for data stored on disk is different from the data structures used for algorithms that run in main memory.

1.3.1 Database Design

Chapter 2 begins with a high-level notation for expressing database designs, called the ***entity-relationship model***. We introduce in Chapter 3 the relational model, which is the model used by the most widely adopted DBMS's, and which we touched upon briefly in Section 1.1.2. We show how to translate entity-relationship designs into relational designs, or "relational database schemas." Later, in Section 6.6, we show how to render relational database schemas formally in the data-definition portion of the SQL language.

Chapter 3 also introduces the reader to the notion of "dependencies," which are formally stated assumptions about relationships among tuples in a relation. Dependencies allow us to improve relational database designs, through a process known as "normalization" of relations.

In Chapter 4 we look at object-oriented approaches to database design. There, we cover the language ODL, which allows one to describe databases in a high-level, object-oriented fashion. We also look at ways in which object-oriented design has been combined with relational modeling, to yield the so-called "object-relational" model. Finally, Chapter 4 also introduces "semistructured data" as an especially flexible database model, and we see its modern embodiment in the document language XML.

1.3.2 Database Programming

Chapters 5 through 10 cover database programming. We start in Chapter 5 with an abstract treatment of queries in the relational model, introducing the family of operators on relations that form "relational algebra."

Chapters 6 through 8 are devoted to SQL programming. As we mentioned, SQL is the dominant query language of the day. Chapter 6 introduces basic ideas regarding queries in SQL and the expression of database schemas in SQL. Chapter 7 covers aspects of SQL concerning constraints and triggers on the data.

Chapter 8 covers certain advanced aspects of SQL programming. First, while the simplest model of SQL programming is a stand-alone, generic query interface, in practice most SQL programming is embedded in a larger program that is written in a conventional language, such as C. In Chapter 8 we learn how to connect SQL statements with a surrounding program and to pass data from the database to the program's variables and vice versa. This chapter also covers how one uses SQL features that specify transactions, connect clients to servers, and authorize access to databases by nonowners.

In Chapter 9 we turn our attention to standards for object-oriented database programming. Here, we consider two directions. The first, OQL (Object Query Language), can be seen as an attempt to make C++, or other object-oriented programming languages, compatible with the demands of high-level database programming. The second, which is the object-oriented features recently adopted in the SQL standard, can be viewed as an attempt to make relational databases and SQL compatible with object-oriented programming.

Finally, in Chapter 10, we return to the study of abstract query languages that we began in Chapter 5. Here, we study logic-based languages and see how they have been used to extend the capabilities of modern SQL.

1.3.3 Database System Implementation

The third part of the book concerns how one can implement a DBMS. The subject of database system implementation in turn can be divided roughly into three parts:

1. *Storage management*: how secondary storage is used effectively to hold data and allow it to be accessed quickly.

2. *Query processing*: how queries expressed in a very high-level language such as SQL can be executed efficiently.

3. *Transaction management*: how to support transactions with the ACID properties discussed in Section 1.2.4.

Each of these topics is covered by several chapters of the book.

Storage-Management Overview

Chapter 11 introduces the memory hierarchy. However, since secondary storage, especially disk, is so central to the way a DBMS manages data, we examine in the greatest detail the way data is stored and accessed on disk. The "block model" for disk-based data is introduced; it influences the way almost everything is done in a database system.

Chapter 12 relates the storage of data elements — relations, tuples, attribute-values, and their equivalents in other data models — to the requirements of the block model of data. Then we look at the important data structures that are used for the construction of indexes. Recall that an index is a data structure that supports efficient access to data. Chapter 13 covers the important one-dimensional index structures — indexed-sequential files, B-trees, and hash tables. These indexes are commonly used in a DBMS to support queries in which a value for an attribute is given and the tuples with that value are desired. B-trees also are used for access to a relation sorted by a given attribute. Chapter 14 discusses multidimensional indexes, which are data structures for specialized applications such as geographic databases, where queries typically ask for the contents of some region. These index structures can also support complex SQL queries that limit the values of two or more attributes, and some of these structures are beginning to appear in commercial DBMS's.

Query-Processing Overview

Chapter 15 covers the basics of query execution. We learn a number of algorithms for efficient implementation of the operations of relational algebra. These algorithms are designed to be efficient when data is stored on disk and are in some cases rather different from analogous main-memory algorithms.

In Chapter 16 we consider the architecture of the query compiler and optimizer. We begin with the parsing of queries and their semantic checking. Next, we consider the conversion of queries from SQL to relational algebra and the selection of a *logical query plan*, that is, an algebraic expression that represents the particular operations to be performed on data and the necessary constraints regarding order of operations. Finally, we explore the selection of a *physical query plan*, in which the particular order of operations and the algorithm used to implement each operation have been specified.

Transaction-Processing Overview

In Chapter 17 we see how a DBMS supports durability of transactions. The central idea is that a log of all changes to the database is made. Anything that is in main-memory but not on disk can be lost in a crash (say, if the power supply is interrupted). Therefore we have to be careful to move from buffer to disk, in the proper order, both the database changes themselves and the log of what changes were made. There are several log strategies available, but each limits our freedom of action in some ways.

Then, we take up the matter of concurrency control — assuring atomicity and isolation — in Chapter 18. We view transactions as sequences of operations that read or write database elements. The major topic of the chapter is how to manage locks on database elements: the different types of locks that may be used, and the ways that transactions may be allowed to acquire locks and release their locks on elements. Also studied are a number of ways to assure atomicity and isolation without using locks.

Chapter 19 concludes our study of transaction processing. We consider the interaction between the requirements of logging, as discussed in Chapter 17, and the requirements of concurrency that were discussed in Chapter 18. Handling of deadlocks, another important function of the transaction manager, is covered here as well. The extension of concurrency control to a distributed environment is also considered in Chapter 19. Finally, we introduce the possibility that transactions are "long," taking hours or days rather than milliseconds. A long transaction cannot lock data without causing chaos among other potential users of that data, which forces us to rethink concurrency control for applications that involve long transactions.

1.3.4 Information Integration Overview

Much of the recent evolution of database systems has been toward capabilities that allow different *data sources*, which may be databases and/or information resources that are not managed by a DBMS, to work together in a larger whole. We introduced you to these issues briefly, in Section 1.1.7. Thus, in the final Chapter 20, we study important aspects of information integration. We discuss the principal modes of integration, including translated and integrated copies of sources called a "data warehouse," and virtual "views" of a collection of sources, through what is called a "mediator."

1.4 Summary of Chapter 1

✦ *Database Management Systems*: A DBMS is characterized by the ability to support efficient access to large amounts of data, which persists over time. It is also characterized by support for powerful query languages and for durable transactions that can execute concurrently in a manner that appears atomic and independent of other transactions.

✦ *Comparison With File Systems*: Conventional file systems are inadequate as database systems, because they fail to support efficient search, efficient modifications to small pieces of data, complex queries, controlled buffering of useful data in main memory, or atomic and independent execution of transactions.

✦ *Relational Database Systems*: Today, most database systems are based on the relational model of data, which organizes information into tables. SQL is the language most often used in these systems.

✦ *Secondary and Tertiary Storage*: Large databases are stored on secondary storage devices, usually disks. The largest databases require tertiary storage devices, which are several orders of magnitude more capacious than disks, but also several orders of magnitude slower.

✦ *Client-Server Systems*: Database management systems usually support a client-server architecture, with major database components at the server and the client used to interface with the user.

✦ *Future Systems*: Major trends in database systems include support for very large "multimedia" objects such as videos or images and the integration of information from many separate information sources into a single database.

✦ *Database Languages*: There are languages or language components for defining the structure of data (data-definition languages) and for querying and modification of the data (data-manipulation languages).

✦ *Components of a DBMS*: The major components of a database management system are the storage manager, the query processor, and the transaction manager.

✦ *The Storage Manager*: This component is responsible for storing data, metadata (information about the schema or structure of the data), indexes (data structures to speed the access to data), and logs (records of changes to the database). This material is kept on disk. An important storage-management component is the buffer manager, which keeps portions of the disk contents in main memory.

✦ *The Query Processor*: This component parses queries, optimizes them by selecting a query plan, and executes the plan on the stored data.

✦ *The Transaction Manager*: This component is responsible for logging database changes to support recovery after a system crashes. It also supports concurrent execution of transactions in a way that assures atomicity (a transaction is performed either completely or not at all), and isolation (transactions are executed as if there were no other concurrently executing transactions).

1.5 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. One searchable index

of database research papers has been constructed by Michael Ley [5]. Alf-Christian Achilles maintains a searchable directory of many indexes relevant to the database field [1].

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 [3] 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 1998 "Asilomar report" [4] 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], [8], and [9].

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

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

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

4. P. A. Bernstein et al., "The Asilomar report on database research," `http://www.acm.org/sigmod/record/issues/9812/asilomar.html` .

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. Stonebraker, M. 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. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

9. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, New York, 1989.