Chapter 1

Introduction to DBMS Implementation

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 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 "transactions") at once. To avoid some of the undersirable consequences of si-

Core Terminology Review

This book is designed for someone who has studied database systems from the point of view of the user (e.g., SQL programming) at the level of Ullman and Widom's *A First Course in Database Systems,* Prentice-Hall, 1997. The following terms should thus be familiar:

- *Data:* any information worth preserving, most likely in electronic form.
- *Database:* a collection of data, organized for access and modification, preserved over a long period.
- *Query:* an operation that extracts specified data from the database.
- *Relation:* an organization of data into a two-dimensional table, where rows (tuples) represent basic entities or facts of some sort, and columns (attributes) represent properties of those entities.
- *Schema:* a description of the structure of the data in a database, often called "metadata."

multaneous 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 *resiliency,* the ability to recover from failures or errors of many types.

1.1 Introducing: The Megatron 2000 Database System

If you have used a DBMS, perhaps one supporting the common SQL query language, you might imagine that implementing such a system is not hard. You might have in mind an implementation such as the recent (fictitious) offering from Megatron Systems Inc.: the Megatron 2000 Database Management System. This system, which is available under UNIX and other operating systems, and which uses the relational approach, supports the SQL query language.

1.1.1 Megatron 2000 Implementation Details

To begin, Megatron 2000 uses the file system to store its relations. For example, the relation Students(name, id, dept) would be stored in the file

/usr/db/Students. The file Students has one line for each tuple. Values of components of a tuple are stored as character strings, separated by the special marker character #. For instance, the file /usr/db/Students might look like:

```
Smith#123#CS
Johnson#522#EE
```

The database schema is stored in a special file named /usr/db/schema. For each relation, the file schema has a line beginning with that relation name, in which attribute names alternate with types. The character # separates elements of these lines. For example, the schema file might contain lines such as

```
Students#name#STR#id#INT#dept#STR
Depts#name#STR#office#STR
```

Here the relation Students(name, id, dept) is described; the type of attributes name and dept are strings while id is an integer. Another relation with schema Depts(name, office) is shown as well.

Example 1.1: Here is an example of a session using the Megatron 2000 DBMS. We are running on a machine called dbhost, and we invoke the DBMS by the UNIX-level command megatron2000.

```
dbhost> megatron2000
```

produces the response

```
WELCOME TO MEGATRON 2000!
```

We are now talking to the Megatron 2000 user interface, to which we can type SQL queries[1] in response to the Megatron prompt (&). A # ends a query. For instance,

```
& SELECT *
  FROM Students #
```

produces as an answer the table

name	*id*	*dept*
Smith	123	CS
Johnson	522	EE

Megatron 2000 also allows us to execute a query and store the result in a new file, if we end the query with a vertical bar and the name of the file. For instance,

[1] There is a brief review of SQL in Section 1.4.2.

```
& SELECT *
  FROM Students
  WHERE id >= 500 I HighId #
```

creates a new file /usr/db/HighId in which only the line

```
Johnson#522#EE
```

appears. □

1.1.2 How Megatron 2000 Executes Queries

Let us consider a common form of SQL query:

```
SELECT * FROM R WHERE <Condition>
```

Megatron 2000 will do the following:

1. Read the file schema to determine the attributes of relation *R* and their types.
2. Check that the <Condition> is semantically valid for *R*.
3. Display each of the attribute names as the header of a column, and draw a line.
4. Read the file named *R*, and for each line:
   (a) Check the condition, and
   (b) Display the line as a tuple, if the condition is true.

To execute

```
SELECT * FROM R WHERE <condition> I T
```

Megatron 2000 does the following:

1. Process query as before, but omit step (2), which generates column headers and a line separating the headers from the tuples.
2. Write the result to a new file /usr/db/T.
3. Add to the file /usr/db/schema an entry for *T* that looks just like the entry for *R*, except that relation name *T* replaces *R*. That is, the schema for *T* is the same as the schema for *R*.

Example 1.2: Now, let us consider a more complicated query, one involving a join of our two example relations Students and Depts:

```
SELECT office
FROM Students, Depts
WHERE Students.name = 'Smith' AND
      Students.dept = Depts.name #
```

This query requires that Megatron 2000 "join" relations Students and Depts. That is, the system must consider in turn each pair of tuples, one from each relation, and determine whether:

a) The tuples represent the same department, and

b) The name of the student is Smith.

The algorithm can be described informally as:

```
for(each tuple s in Students)
    for(each tuple d in Depts)
        if(s and d satisfy the WHERE-condition)
            display the office value from Depts;
```

□

1.1.3 What's Wrong With Megatron 2000?

It may come as no surprise that a DBMS is not implemented like our imaginary Megatron 2000. There are a number of ways that the implementation described here is inadequate for applications involving significant amounts of data or multiple users of data. A partial list of problems follows:

- The tuple layout on disk is inadequate, with no flexibility when the database is modified. For instance, if we change EE to ECON in one Students tuple, the entire file has to be rewritten, as every subsequent character is moved two positions down the file.
- Search is very expensive. We always have to read an entire relation, even if the query gives us a value or values that enable us to focus on one tuple, as in the query of Example 1.2. There, we had to look at the entire Student relation, even though the only one we wanted was that for student Smith.
- Query-processing is by "brute force," and much cleverer ways of performing operations like joins are available. For instance, we shall see that in a query like that of Example 1.2, it is not necessary to look at all pairs of tuples, one from each relation, even if the name of one student (Smith) were not specified in the query.
- There is no way for useful data to be buffered in main memory; all data comes off the disk, all the time.

- There is no concurrency control. Several users can modify a file at the same time with unpredictable results.
- There is no reliability; we can lose data in a crash or leave operations half done.
- There is little security. Presumably the underlying operating system controls access in some coarse manner, e.g., different users are either permitted or forbidden to access the file holding a given relation, but one cannot be given access, say. to certain attributes of a relation and not others.

It is the purpose of this book to introduce the reader to better ways of building a database management system. We hope you will enjoy the study.

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.

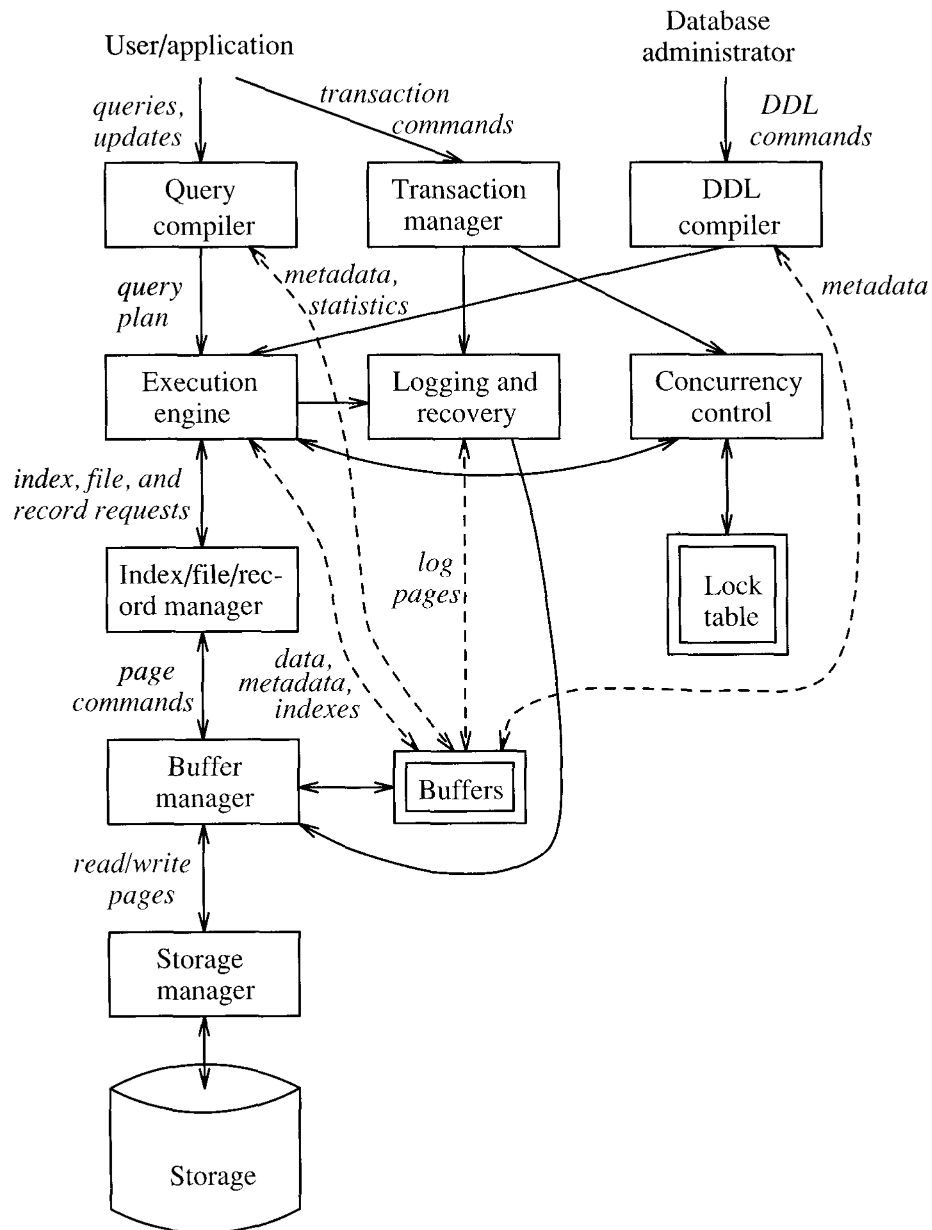

Figure 1.1: Database management system components

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). There are two paths along which user actions affect the database:

1. *Answering the query.* The query is parsed and optimized by a *query compiler.* The resulting *query plan,* or sequence of actions to be taken 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 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.

2. *Transaction processing.* Queries and other actions are grouped into *transactions,* which are units that must be executed atomically and in isolation, as discussed in the introduction to this chapter; 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:

   (a) A *concurrency-control manager,* (or *scheduler),* responsible for assuring atomicity and isolation of transactions, and

   (b) 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 Main-Memory Buffers and the Buffer Manager

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.

Thus, a DBMS component railed 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 6.8.

1.2.4 Transaction Processing

As we mentioned, 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

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., a certain attribute is a key, students may not take more than 8 courses at a time, and so on). Transactions are expected to preserve the consistency of the database. We discuss this matter in more detail in Section 9.1.

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 ("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 6.1 and with which you may be familiar already. 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 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 This Book

The subject of database system implementation 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.

1.3.1 Prerequisites

Although this book assumes you have no prior knowledge of DBMS implementation, it is intended as the text for a "second course" in a sequence of courses

covering databases, or as part of a comprehensive, one-semester course. In particular, it is a follow-on to the text *A First Course in Database Systems* by Jeff Ullman and Jennifer Widom. The latter book covers:

1. *Database design*: the informal, high-level, specification of the schema of a database, using notations such as the entity/relationship model or ODL (Object Description Language), and the implementation of designs in the data-definition portion of SQL.

2. *Database programming:* writing queries and database modification commands using appropriate languages, especially SQL.

The impact of database-design technology on DBMS implementation is small, but you should have familiarity with the relational model and how data is represented by relations, since much of what we say in this book addresses how one stores relations, optimizes queries about relations, and how one controls access to relations by methods such as locking. Further, in order to appreciate the technology behind query processing, you should be familiar with SQL programming. A brief review of these topics is in Section 1.4.

Additionally, we assume you are familiar with *files* (named storage areas in which data can be kept). We expect that you are familiar with the architecture of a conventional file system, i.e., the part of an operating system that manages its files. The way a DBMS manages files is rather different, and we cover the basics of this important topic.

1.3.2 Storage-Management Overview

This book begins with chapters on storage management. Chapter 2 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 3 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 4 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. Chapter 5 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.

1.3.3 Query-Processing Overview

Chapter 6 introduces the relational algebra as a way to describe the execution of queries. This chapter then covers the basics of query execution, including a number of algorithms for efficient implementation of key operations such as joins of relations.

In Chapter 7 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.

1.3.4 Transaction-Processing Overview

In Chapter 8 we see how a DBMS supports durability of transactions. The central idea is that a log of all changes to the database is made. Since anything that is in main-memory but not on disk can be lost in a crash (say, if the power supply is interrupted), 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 9. 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 10 concludes our study of transaction processing. We consider the interaction between the requirements of logging, as discussed in Chapter 8, and the requirements of concurrency that were discussed in Chapter 9. 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 10. 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.

1.3.5 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. Thus, Chapter 11 is devoted to a study of important aspects of this new technology, called *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, called a *mediator.*

1.4 Review of Database Models and Languages

In this section, we shall give the reader a brief review of SQL and the relational model. We also review the notion of objects as in an object-oriented database. The examples are taken from Ullman and Widom's, *A First Course in Database Systems.*

1.4.1 Relational Model Review

A *relation* is a set of *tuples*, which in turn are lists of values. All tuples of a relation have the same number of components. and corresponding components from different tuples are of the same type. We display a relation by listing each of its tuples as a row. Column headers called *attributes* represent the meaning of each component of the tuples. The relation name and its attribute names and types are the *schema* for the relation.

Example 1.3 : The relation Movie, which we use frequently in examples, might consist of the following:

title	*year*	*length*
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

The schema for the relation is

```
Movie(title, year, length)
```

The attributes are title, year, and length, which we may suppose are of types string, integer, and integer, respectively. Each of the three rows below the line is a tuple. For instance, the first tuple says that "Star Wars" was made in 1977 and is 124 minutes long. □

A *database schema* is a collection of relation schemas. In our running example of movies, we shall often use relations

```
Movie(title, year, length, studioName)
MovieStar(name, address, gender, birthdate)
StarsIn(title, year, starName)
Studio(name, address)
```

The first is like the Movie relation from Example 1.3, although it adds the name of the producing studio when we need some additional connections in examples. The second gives information about movie stars, and the third connects movies to their stars. The fourth gives some information about movie studios. The intent of the various attributes should be clear from their names.

1.4.2 SQL Review

The database language SQL has a large number of capabilities, including statements that query and modify the database. Database modification is through three commands, called INSERT, DELETE, and UPDATE, whose syntaxes we shall not review here. Queries are generally expressed with a "select-from-where" statement, which actually has the general form shown in Fig. 1.2. Only the first two lines *(clauses),* the ones introduced by the keywords SELECT and FROM, are required.

```
SELECT <attribute list>
FROM <relation list>
WHERE <condition>
GROUP BY <attribute list>
HAVING <condition>
ORDER BY <attribute list>
```

Figure 1.2: General form of an SQL query

Although better ways exist, the result of such a query can be computed by:

1. Taking all possible combinations of tuples from the relations in the FROM clause,
2. Throwing away any that do not meet the condition of the WHERE clause,
3. Grouping the remaining tuples according to their values in the attributes mentioned in the GROUP BY clause (if any),
4. Testing each group according to the condition in the HAVING clause (if any), and rejecting all groups that do not meet this condition,
5. Computing tuples from specified attributes and aggregations of attributes (e.g., sum within a group) as specified by the SELECT clause, and finally
6. Ordering the resulting tuples according to values in the list of attributes in the ORDER BY clause.

Example 1.4: Figure 1.3 is a simple SQL query with only the first three clauses. It asks for the names of stars that starred in movies made by Paramount

Studios, and the titles of the movies that they starred in. Note that title and year together are the key for Movie, since there could be two movies of the same title (but not in the same year, we hope). D

```
SELECT starName, Movie.title
FROM Movie, StarsIn
WHERE Movie.title = StarsIn.title AND
      Movie.year = StarsIn.year AND
      studioName = 'Paramount';
```

Figure 1.3: Finding the Paramount stars

Example 1.5: Figure 1.4 is a more complicated query. It asks us first to find the stars who starred in at least three movies. That part of the query is accomplished by grouping the Starsln tuples by the name of the star (the GROUP BY clause) and then filtering out the groups that have two or fewer tuples (the HAVING clause).

```
SELECT starName, MIN(year) AS minYear
FROM Starsln
GROUP BY starName
HAVING COUNT(*) >= 3
ORDER BY minYear;
```

Figure 1.4: Finding earliest years of stars appearing in at least three movies

Next, from each of the surviving groups, the SELECT clause tells us to produce the name of the star and the earliest year that star appeared in a movie. The second component of the select-list, which is MIN(year), is given the attribute name minYear. Last, the ORDER BY clause says that the output tuples are to be listed in increasing order of the value of minYear; that is, the stars appear in the order of their first movie. □

Subqueries

One of the powerful features that SQL provides is the ability to use *subqueries* within a WHERE, FROM, or HAVING clause. A subquery is a complete select-from-where statement whose value is tested in one of these clauses.

Example 1.6 : In Fig. 1.5 is an SQL query with a subquery. The overall query finds the title and year of the movies made not in Hollywood. The subquery

```
SELECT title, year
FROM Movie
WHERE studioName IN (
    SELECT name
    FROM Studio
    WHERE address NOT LIKE '%Hollywood%'
);
```

Figure 1.5: Finding the movies not made in Hollywood

```
SELECT name
FROM Studio
WHERE address NOT LIKE '%Hollywood%'
```

produces a one-column relation consisting of the names of all the studios that do not have "Hollywood" somewhere in their address. This subquery is then used in the WHERE clause of the outer query to identify those movies whose studio appears in this set of studio names. □

Views

Another important capability of SQL is the definition of *views,* which are descriptions of relations that are not stored, but constructed as needed from stored relations.

Example 1.7: Figure 1.6 shows the definition of a view; it is the title and year of movies made by Paramount studios. The definition of view ParamountMovie is stored as part of the schema of the database, but its tuples are not computed at this time. If we use ParamountMovie as a relation in a query, then its tuples, or the necessary subset of its tuples if the query does not need the whole relation, will be constructed logically by folding the view definition into the query. Thus, these tuples are never actually stored in the database. □

```
CREATE VIEW ParamountMovie AS
    SELECT title, year
    FROM Movie
    WHERE studioName = 'Paramount';
```

Figure 1.6: View for only the movies made by Paramount

1.4.3 Relational and Object-Oriented Data

Most of what is discussed in this book assumes that the database is relational: data is modeled by tables, data items are tuples or rows of the table, and tuples have a fixed number of components, each of a fixed type determined by the relation's schema. This view of data was suggested in Example 1.3. At a different level, we can think of a tuple as a "struct" (the C term) or record, with one field for the value of each attribute.

There is another model of data that is used in some database systems: data as objects. In this model, the elementary data item is an *object.* Objects are grouped into *classes,* and each class has a schema that is a list of of *properties*:

1. Some of those properties are attributes, which can be represented like attributes of a relational tuple.
2. Other properties are *relationships*, which connect an object to one or more other objects. We can think, at an implementation level, of a relationship as a list of pointers to these objects.
3. Still other properties are *methods,* that is, functions that can be applied to objects of the class.

While the code for methods will typically be stored outside of the objects, the rest of the object-oriented formulation of data fits well into our general framework. That is, we shall generally think of "files" as the largest units of data. Files are simply named collections of data, and files are generally composed of smaller units, for which the following terminology is used:

a) In the earliest databases, files were composed of *records,* which were composed *offields.* A record is analogous to a "struct" in C and its descendant programming languages such as C++ or Java.

b) In relational databases, files are *relations*; they are composed of *tuples,* which are composed of *attributes.*

c) In object-oriented databases, files are the *extents* of *classes,* that is, the set of currently existing objects of one class. Extents are composed of *objects,* and the objects have *fields* or "instance variables," whose values represent either attributes of the object or the set of related objects according to some relationship.

It is useful to draw the following analogies:

1. A file, relation, and extent are similar concepts; they are each values consisting of some smaller elements that have a common schema (records, tuples, or objects).
2. The schema of a file or relation and the definition of a class are similar concepts; each describes the elements of a file, relation, or an extent (of the class).

3. Records, tuples, and objects are similar concepts. Each is often implemented as if they were "structs" in C.

1.5 Summary of Chapter 1

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

4 *Comparison With File Systems:* Converntional 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.

✦ *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.

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

4 *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).

4 *SQL:* This query language, based on the relational model, is an important standard. Both the language and the relational model are central to large portions of this book.

4 *Data Concepts:* File systems, conventional programming languages like C, the relational model, and object-oriented data models share many common notions, often with different terminology. There are analogies among structs, tuples, and objects, and among files, relations, and classes.

1.6 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 [6]. Alf-Christian Achilles maintains a searchable directory of many indexes relevant to the database field [1].

The background assumed for this text is obtainable from [8]. The SQL2 and SQL3 standards are obtainable on-line by anonymous FTP from [5]. We suggest [4] for those wanting an SQL2 manual.

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 [2] 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.

The 1998 "Asilomar report" [3] 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.

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

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

3. P. A. Bernstein et al., "The Asilomar report on database research," `http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/Asilomar_Final.htm`.

4. Date, C. J. and H. Darwen, *A Guide to the SQL Standard,* Fourth Edition, Addison-Wesley, Reading, MA, 1997.

5. `ftp://jerry.ece.umassd.edu/isowg3`.

6. `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`.

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

8. J. D. Ullman and J. Widom, *A First Course in Database Systems,* Prentice-Hall, Englewood Cliffs NJ, 1997.