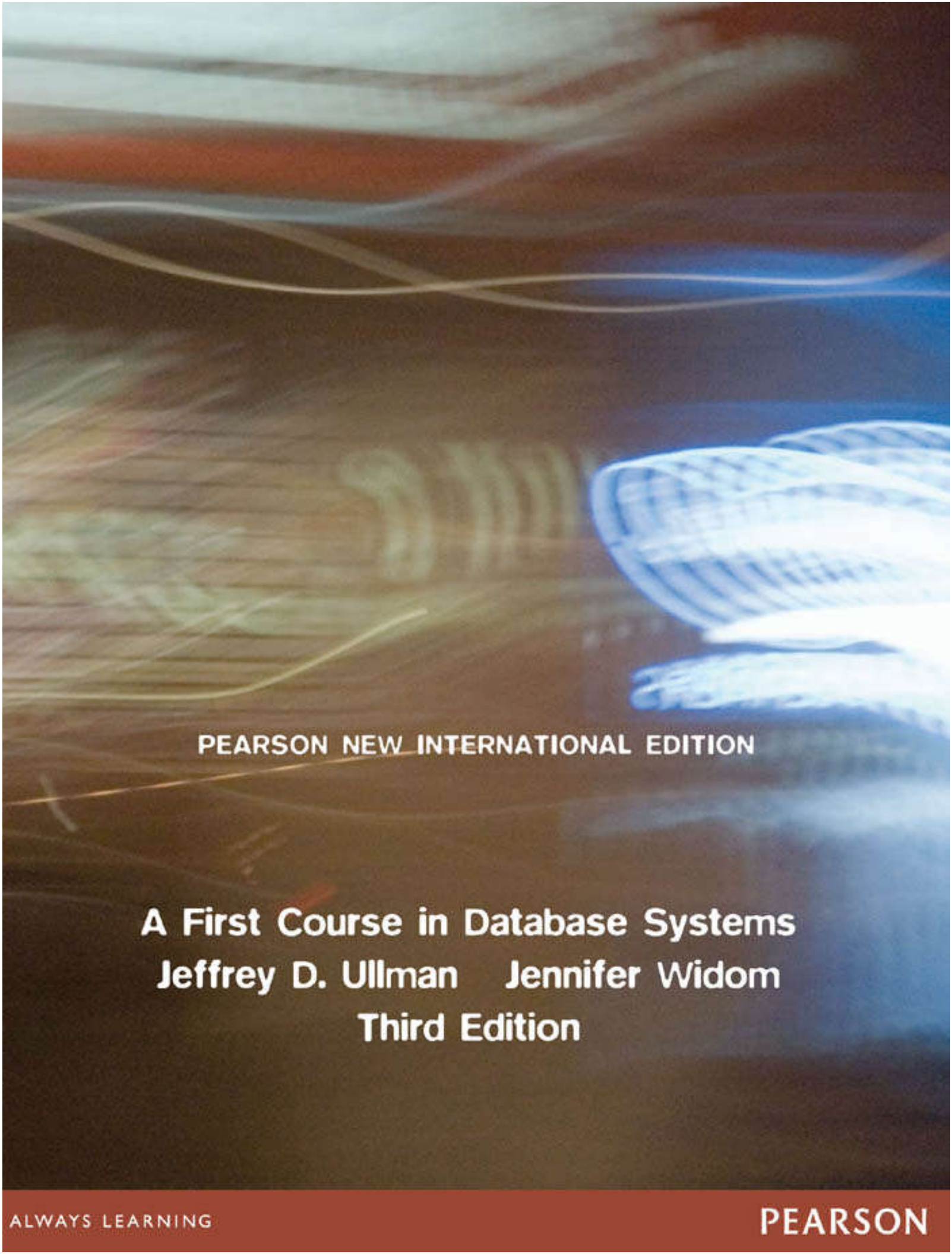
PEARSON NEW INTERNATIONAL EDITION
A First Course in Database Systems
Jeffrey D. Ullman Jennifer Widom
Third Edition
ALWAYS LEARNING
PEARSON

Pearson New International Edition

A First Course in Database Systems
Jeffrey D. Ullman Jennifer Widom
Third Edition

PEARSON

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsoned.co.uk

ISBN 10: 1-292-02582-4
ISBN 13: 978-1-292-02582-7

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Printed in the United States of America

Table of Contents

This page intentionally left blank

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.

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 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.

1.2 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.

1.3 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.

1.4 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.

1.5 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.

2 Overview of a Database Management System

In Fig. 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.

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. 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 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.

2.2 Overview of Query Processing

The great majority of interactions with the DBMS follow the path on the left side of Fig. 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

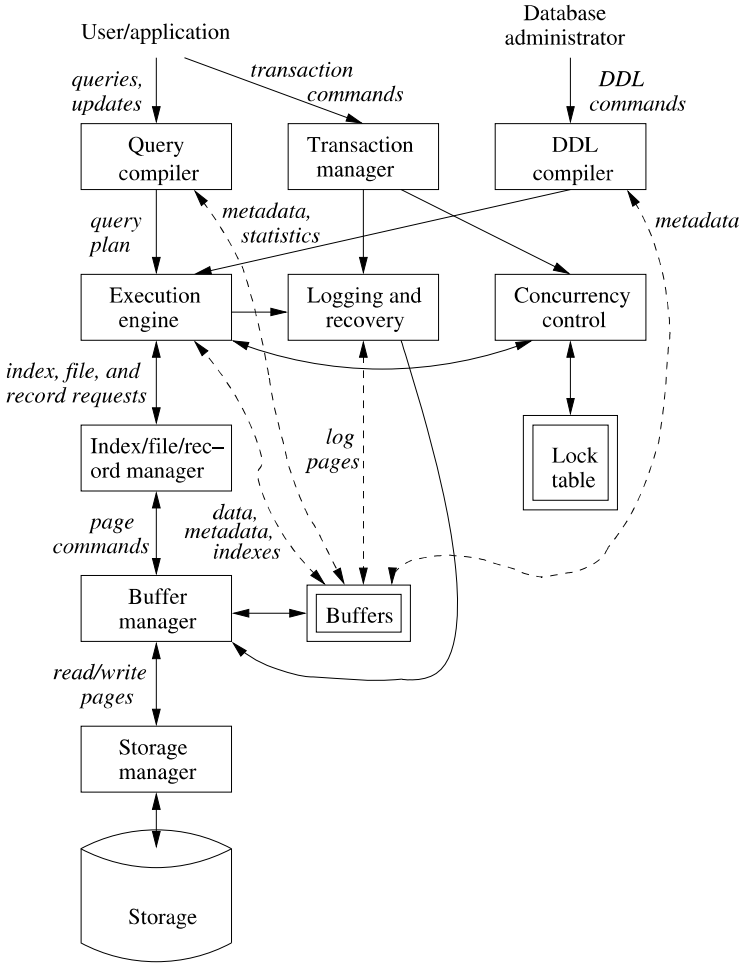

Figure 1: Database management system components

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.

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.

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. 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.

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 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. 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.

3 Outline of Database-System Studies

The study of databases can be divided into five parts.

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.

4 References

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.

The Relational Model of Data

This chapter introduces the most important model of data: the two-dimensional table, or "relation." We begin with an overview of data models in general. We give the basic terminology for relations and show how the model can be used to represent typical forms of data. We then introduce a portion of the language SQL — that part used to declare relations and their structure. The chapter closes with an introduction to relational algebra. We see how this notation serves as both a query language — the aspect of a data model that enables us to ask questions about the data — and as a constraint language — the aspect of a data model that lets us restrict the data in the database in various ways.

1 An Overview of Data Models

The notion of a "data model" is one of the most fundamental in the study of database systems. In this brief summary of the concept, we define some basic terminology and mention the most important data models.

1.1 What is a Data Model?

A *data model* is a notation for describing data or information. The description generally consists of three parts:

1. *Structure of the data.* You may be familiar with tools in programming languages such as C or Java for describing the structure of the data used by a program: arrays and structures ("structs") or objects, for example. The data structures used to implement data in the computer are sometimes referred to, in discussions of database systems, as a *physical data model*, although in fact they are far removed from the gates and electrons that truly serve as the physical implementation of the data. In the database

world, data models are at a somewhat higher level than data structures, and are sometimes referred to as a *conceptual model* to emphasize the difference in level. We shall see examples shortly.

2. *Operations on the data.* In programming languages, operations on the data are generally anything that can be programmed. In database data models, there is usually a limited set of operations that can be performed. We are generally allowed to perform a limited set of *queries* (operations that retrieve information) and *modifications* (operations that change the database). This limitation is not a weakness, but a strength. By limiting operations, it is possible for programmers to describe database operations at a very high level, yet have the database management system implement the operations efficiently. In comparison, it is generally impossible to optimize programs in conventional languages like C, to the extent that an inefficient algorithm (e.g., bubblesort) is replaced by a more efficient one (e.g., quicksort).

3. *Constraints on the data.* Database data models usually have a way to describe limitations on what the data can be. These constraints can range from the simple (e.g., "a day of the week is an integer between 1 and 7" or "a movie has at most one title") to some very complex limitations.

1.2 Important Data Models

Today, the two data models of preeminent importance for database systems are:

1. The relational model, including object-relational extensions.

2. The semistructured-data model, including XML and related standards.

The first, which is present in all commercial database management systems, is the subject of this chapter. The semistructured model, of which XML is the primary manifestation, is an added feature of most relational DBMS's, and appears in a number of other contexts as well.

1.3 The Relational Model in Brief

The relational model is based on tables, of which Fig. 1 is an example. We shall discuss this model beginning in Section 2. This relation, or table, describes movies: their title, the year in which they were made, their length in minutes, and the genre of the movie. We show three particular movies, but you should imagine that there are many more rows to this table — one row for each movie ever made, perhaps.

The structure portion of the relational model might appear to resemble an array of structs in C, where the column headers are the field names, and each

title	*year*	*length*	*genre*
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 1: An example relation

of the rows represent the values of one struct in the array. However, it must be emphasized that this physical implementation is only one possible way the table could be implemented in physical data structures. In fact, it is not the normal way to represent relations, and a large portion of the study of database systems addresses the right ways to implement such tables. Much of the distinction comes from the scale of relations — they are not normally implemented as main-memory structures, and their proper physical implementation must take into account the need to access relations of very large size that are resident on disk.

The operations normally associated with the relational model form the "relational algebra," which we discuss beginning in Section 4. These operations are table-oriented. As an example, we can ask for all those rows of a relation that have a certain value in a certain column. For example, we can ask of the table in Fig. 1 for all the rows where the genre is "comedy."

The constraint portion of the relational data model will be touched upon briefly in Section 5. However, as a brief sample of what kinds of constraints are generally used, we could decide that there is a fixed list of genres for movies, and that the last column of every row must have a value that is on this list. Or we might decide (incorrectly, it turns out) that there could never be two movies with the same title, and constrain the table so that no two rows could have the same string in the first component.

1.4 The Semistructured Model in Brief

Semistructured data resembles trees or graphs, rather than tables or arrays. The principal manifestation of this viewpoint today is XML, a way to represent data by hierarchically nested tagged elements. The tags, similar to those used in HTML, define the role played by different pieces of data, much as the column headers do in the relational model. For example, the same data as in Fig. 1 might appear in an XML "document" as in Fig. 2.

The operations on semistructured data usually involve following paths in the implied tree from an element to one or more of its nested subelements, then to subelements nested within those, and so on. For example, starting at the outer `<Movies>` element (the entire document in Fig. 2), we might move to each of its nested `<Movie>` elements, each delimited by the tag `<Movie>` and matching `</Movie>` tag, and from each `<Movie>` element to its nested `<Genre>`

```
<Movies>
    <Movie title="Gone With the Wind">
        <Year>1939</Year>
        <Length>231</Length>
        <Genre>drama</Genre>
    </Movie>
    <Movie title="Star Wars">
        <Year>1977</Year>
        <Length>124</Length>
        <Genre>sciFi</Genre>
    </Movie>
    <Movie title="Wayne's World">
        <Year>1992</Year>
        <Length>95</Length>
        <Genre>comedy</Genre>
    </Movie>
</Movies>
```

Figure 2: Movie data as XML

element, to see which movies belong to the "comedy" genre.

Constraints on the structure of data in this model often involve the data type of values associated with a tag. For instance, are the values associated with the `<Length>` tag integers or can they be arbitrary character strings? Other constraints determine which tags can appear nested within which other tags. For example, must each `<Movie>` element have a `<Length>` element nested within it? What other tags, besides those shown in Fig. 2 might be used within a `<Movie>` element? Can there be more than one genre for a movie?

1.5 Other Data Models

There are many other models that are, or have been, associated with DBMS's. A modern trend is to add object-oriented features to the relational model. There are two effects of object-orientation on relations:

1. Values can have structure, rather than being elementary types such as integer or strings, as they were in Fig. 1.

2. Relations can have associated methods.

In a sense, these extensions, called the *object-relational* model, are analogous to the way structs in C were extended to objects in C++.

There are even database models of the purely object-oriented kind. In these, the relation is no longer the principal data-structuring concept, but becomes only one option among many structures.

There are several other models that were used in some of the earlier DBMS's, but that have now fallen out of use. The *hierarchical model* was, like semistructured data, a tree-oriented model. Its drawback was that unlike more modern models, it really operated at the physical level, which made it impossible for programmers to write code at a conveniently high level. Another such model was the *network model*, which was a graph-oriented, physical-level model. In truth, both the hierarchical model and today's semistructured models, allow full graph structures, and do not limit us strictly to trees. However, the generality of graphs was built directly into the network model, rather than favoring trees as these other models do.

1.6 Comparison of Modeling Approaches

Even from our brief example, it appears that semistructured models have more flexibility than relations. This difference becomes even more apparent when we discuss, as we shall, how full graph structures are embedded into tree-like, semistructured models. Nevertheless, the relational model is still preferred in DBMS's, and we should understand why. A brief argument follows.

Because databases are large, efficiency of access to data and efficiency of modifications to that data are of great importance. Also very important is ease of use — the productivity of programmers who use the data. Surprisingly, both goals can be achieved with a model, particularly the relational model, that:

1. Provides a simple, limited approach to structuring data, yet is reasonably versatile, so anything can be modeled.

2. Provides a limited, yet useful, collection of operations on data.

Together, these limitations turn into features. They allow us to implement languages, such as SQL, that enable the programmer to express their wishes at a very high level. A few lines of SQL can do the work of thousands of lines of C, or hundreds of lines of the code that had to be written to access data under earlier models such as network or hierarchical. Yet the short SQL programs, because they use a strongly limited sets of operations, can be optimized to run as fast, or faster than the code written in alternative languages.

2 Basics of the Relational Model

The relational model gives us a single way to represent data: as a two-dimensional table called a *relation*. Figure 1, which we copy here as Fig. 3, is an example of a relation, which we shall call `Movies`. The rows each represent a

movie, and the columns each represent a property of movies. In this section, we shall introduce the most important terminology regarding relations, and illustrate them with the `Movies` relation.

title	*year*	*length*	*genre*
`Gone With the Wind`	`1939`	`231`	`drama`
`Star Wars`	`1977`	`124`	`sciFi`
`Wayne's World`	`1992`	`95`	`comedy`

Figure 3: The relation `Movies`

2.1 Attributes

The columns of a relation are named by *attributes*; in Fig. 3 the attributes are `title`, `year`, `length`, and `genre`. Attributes appear at the tops of the columns. Usually, an attribute describes the meaning of entries in the column below. For instance, the column with attribute `length` holds the length, in minutes, of each movie.

2.2 Schemas

The name of a relation and the set of attributes for a relation is called the *schema* for that relation. We show the schema for the relation with the relation name followed by a parenthesized list of its attributes. Thus, the schema for relation `Movies` of Fig. 3 is

```
Movies(title, year, length, genre)
```

The attributes in a relation schema are a set, not a list. However, in order to talk about relations we often must specify a "standard" order for the attributes. Thus, whenever we introduce a relation schema with a list of attributes, as above, we shall take this ordering to be the standard order whenever we display the relation or any of its rows.

In the relational model, a database consists of one or more relations. The set of schemas for the relations of a database is called a *relational database schema*, or just a *database schema*.

2.3 Tuples

The rows of a relation, other than the header row containing the attribute names, are called *tuples*. A tuple has one *component* for each attribute of the relation. For instance, the first of the three tuples in Fig. 3 has the four components `Gone With the Wind`, `1939`, `231`, and `drama` for attributes `title`, `year`, `length`, and `genre`, respectively. When we wish to write a tuple

Conventions for Relations and Attributes

We shall generally follow the convention that relation names begin with a capital letter, and attribute names begin with a lower-case letter. However, later in this book we shall talk of relations in the abstract, where the names of attributes do not matter. In that case, we shall use single capital letters for both relations and attributes, e.g., $R(A, B, C)$ for a generic relation with three attributes.

in isolation, not as part of a relation, we normally use commas to separate components, and we use parentheses to surround the tuple. For example,

(`Gone With the Wind`, `1939`, `231`, `drama`)

is the first tuple of Fig. 3. Notice that when a tuple appears in isolation, the attributes do not appear, so some indication of the relation to which the tuple belongs must be given. We shall always use the order in which the attributes were listed in the relation schema.

2.4 Domains

The relational model requires that each component of each tuple be atomic; that is, it must be of some elementary type such as integer or string. It is not permitted for a value to be a record structure, set, list, array, or any other type that reasonably can have its values broken into smaller components.

It is further assumed that associated with each attribute of a relation is a *domain*, that is, a particular elementary type. The components of any tuple of the relation must have, in each component, a value that belongs to the domain of the corresponding column. For example, tuples of the `Movies` relation of Fig. 3 must have a first component that is a string, second and third components that are integers, and a fourth component whose value is a string.

It is possible to include the domain, or data type, for each attribute in a relation schema. We shall do so by appending a colon and a type after attributes. For example, we could represent the schema for the `Movies` relation as:

```
Movies(title:string, year:integer, length:integer, genre:string)
```

2.5 Equivalent Representations of a Relation

Relations are sets of tuples, not lists of tuples. Thus the order in which the tuples of a relation are presented is immaterial. For example, we can list the three tuples of Fig. 3 in any of their six possible orders, and the relation is "the same" as Fig. 3.

Moreover, we can reorder the attributes of the relation as we choose, without changing the relation. However, when we reorder the relation schema, we must be careful to remember that the attributes are column headers. Thus, when we change the order of the attributes, we also change the order of their columns. When the columns move, the components of tuples change their order as well. The result is that each tuple has its components permuted in the same way as the attributes are permuted.

For example, Fig. 4 shows one of the many relations that could be obtained from Fig. 3 by permuting rows and columns. These two relations are considered "the same." More precisely, these two tables are different presentations of the same relation.

year	*genre*	*title*	*length*
1977	sciFi	Star Wars	124
1992	comedy	Wayne's World	95
1939	drama	Gone With the Wind	231

Figure 4: Another presentation of the relation `Movies`

2.6 Relation Instances

A relation about movies is not static; rather, relations change over time. We expect to insert tuples for new movies, as these appear. We also expect changes to existing tuples if we get revised or corrected information about a movie, and perhaps deletion of tuples for movies that are expelled from the database for some reason.

It is less common for the schema of a relation to change. However, there are situations where we might want to add or delete attributes. Schema changes, while possible in commercial database systems, can be very expensive, because each of perhaps millions of tuples needs to be rewritten to add or delete components. Also, if we add an attribute, it may be difficult or even impossible to generate appropriate values for the new component in the existing tuples.

We shall call a set of tuples for a given relation an *instance* of that relation. For example, the three tuples shown in Fig. 3 form an instance of relation `Movies`. Presumably, the relation `Movies` has changed over time and will continue to change over time. For instance, in 1990, `Movies` did not contain the tuple for `Wayne's World`. However, a conventional database system maintains only one version of any relation: the set of tuples that are in the relation "now." This instance of the relation is called the *current instance*.[1]

[1]Databases that maintain historical versions of data as it existed in past times are called *temporal databases*.

2.7 Keys of Relations

There are many constraints on relations that the relational model allows us to place on database schemas. We will not discuss most constraints here. However, one kind of constraint is so fundamental that we shall introduce it here: *key* constraints. A set of attributes forms a *key* for a relation if we do not allow two tuples in a relation instance to have the same values in all the attributes of the key.

Example 1: We can declare that the relation `Movies` has a key consisting of the two attributes `title` and `year`. That is, we don't believe there could ever be two movies that had both the same title and the same year. Notice that `title` by itself does not form a key, since sometimes "remakes" of a movie appear. For example, there are three movies named *King Kong*, each made in a different year. It should also be obvious that `year` by itself is not a key, since there are usually many movies made in the same year. □

We indicate the attribute or attributes that form a key for a relation by underlining the key attribute(s). For instance, the `Movies` relation could have its schema written as:

`Movies(`<u>`title`</u>`, `<u>`year`</u>`, length, genre)`

Remember that the statement that a set of attributes forms a key for a relation is a statement about all possible instances of the relation, not a statement about a single instance. For example, looking only at the tiny relation of Fig. 3, we might imagine that `genre` by itself forms a key, since we do not see two tuples that agree on the value of their `genre` components. However, we can easily imagine that if the relation instance contained more movies, there would be many dramas, many comedies, and so on. Thus, there would be distinct tuples that agreed on the `genre` component. As a consequence, it would be incorrect to assert that `genre` is a key for the relation `Movies`.

While we might be sure that `title` and `year` can serve as a key for `Movies`, many real-world databases use artificial keys, doubting that it is safe to make any assumption about the values of attributes outside their control. For example, companies generally assign employee ID's to all employees, and these ID's are carefully chosen to be unique numbers. One purpose of these ID's is to make sure that in the company database each employee can be distinguished from all others, even if there are several employees with the same name. Thus, the employee-ID attribute can serve as a key for a relation about employees.

In US corporations, it is normal for every employee to have a Social-Security number. If the database has an attribute that is the Social-Security number, then this attribute can also serve as a key for employees. Note that there is nothing wrong with there being several choices of key, as there would be for employees having both employee ID's and Social-Security numbers.

The idea of creating an attribute whose purpose is to serve as a key is quite widespread. In addition to employee ID's, we find student ID's to distinguish

students in a university. We find drivers' license numbers and automobile registration numbers to distinguish drivers and automobiles, respectively. You undoubtedly can find more examples of attributes created for the primary purpose of serving as keys.

```
Movies(
    title:string,
    year:integer,
    length:integer,
    genre:string,
    studioName:string,
    producerC#:integer
)
MovieStar(
    name:string,
    address:string,
    gender:char,
    birthdate:date
)
StarsIn(
    movieTitle:string,
    movieYear:integer,
    starName:string
)
MovieExec(
    name:string,
    address:string,
    cert#:integer,
    netWorth:integer
)
Studio(
    name:string,
    address:string,
    presC#:integer
)
```

Figure 5: Example database schema about movies

2.8 An Example Database Schema

We shall close this section with an example of a complete database schema. The topic is movies, and it builds on the relation `Movies` that has appeared so far in examples. The database schema is shown in Fig. 5. Here are the things we need to know to understand the intention of this schema.

`Movies`

This relation is an extension of the example relation we have been discussing so far. Remember that its key is `title` and `year` together. We have added two new attributes; `studioName` tells us the studio that owns the movie, and `producerC#` is an integer that represents the producer of the movie in a way that we shall discuss when we talk about the relation `MovieExec` below.

`MovieStar`

This relation tells us something about stars. The key is `name`, the name of the movie star. It is not usual to assume names of persons are unique and therefore suitable as a key. However, movie stars are different; one would never take a name that some other movie star had used. Thus, we shall use the convenient fiction that movie-star names are unique. A more conventional approach would be to invent a serial number of some sort, like social-security numbers, so that we could assign each individual a unique number and use that attribute as the key. We take that approach for movie executives, as we shall see. Another interesting point about the `MovieStar` relation is that we see two new data types. The gender can be a single character, M or F. Also, birthdate is of type "date," which might be a character string of a special form.

`StarsIn`

This relation connects movies to the stars of that movie, and likewise connects a star to the movies in which they appeared. Notice that movies are represented by the key for `Movies` — the title and year — although we have chosen different attribute names to emphasize that attributes `movieTitle` and `movieYear` represent the movie. Likewise, stars are represented by the key for `MovieStar`, with the attribute called `starName`. Finally, notice that all three attributes are necessary to form a key. It is perfectly reasonable to suppose that relation `StarsIn` could have two distinct tuples that agree in any two of the three attributes. For instance, a star might appear in two movies in one year, giving rise to two tuples that agreed in `movieYear` and `starName`, but disagreed in `movieTitle`.

`MovieExec`

This relation tells us about movie executives. It contains their name, address, and networth as data about the executive. However, for a key we have invented "certificate numbers" for all movie executives, including producers (as appear in the relation `Movies`) and studio presidents (as appear in the relation `Studio`, below). These are integers; a different one is assigned to each executive.

acctNo	*type*	*balance*
12345	savings	12000
23456	checking	1000
34567	savings	25

The relation `Accounts`

firstName	*lastName*	*idNo*	*account*
Robbie	Banks	901-222	12345
Lena	Hand	805-333	12345
Lena	Hand	805-333	23456

The relation `Customers`

Figure 6: Two relations of a banking database

`Studio`

This relation tells about movie studios. We rely on no two studios having the same name, and therefore use `name` as the key. The other attributes are the address of the studio and the certificate number for the president of the studio. We assume that the studio president is surely a movie executive and therefore appears in `MovieExec`.

2.9 Exercises for Section 2

Exercise 2.1: In Fig. 6 are instances of two relations that might constitute part of a banking database. Indicate the following:

a) The attributes of each relation.

b) The tuples of each relation.

c) The components of one tuple from each relation.

d) The relation schema for each relation.

e) The database schema.

f) A suitable domain for each attribute.

g) Another equivalent way to present each relation.

Exercise 2.2: In Section 2.7 we suggested that there are many examples of attributes that are created for the purpose of serving as keys of relations. Give some additional examples.

!! Exercise 2.3: How many different ways (considering orders of tuples and attributes) are there to represent a relation instance if that instance has:

a) Three attributes and three tuples, like the relation `Accounts` of Fig. 6?

b) Four attributes and five tuples?

c) n attributes and m tuples?

3 Defining a Relation Schema in SQL

SQL (pronounced "sequel") is the principal language used to describe and manipulate relational databases. There is a current standard for SQL, called SQL-99. Most commercial database management systems implement something similar, but not identical to, the standard. There are two aspects to SQL:

1. The *Data-Definition* sublanguage for declaring database schemas and
2. The *Data-Manipulation* sublanguage for *querying* (asking questions about) databases and for modifying the database.

The distinction between these two sublanguages is found in most languages; e.g., C or Java have portions that declare data and other portions that are executable code. These correspond to data-definition and data-manipulation, respectively.

In this section we shall begin a discussion of the data-definition portion of SQL.

3.1 Relations in SQL

SQL makes a distinction between three kinds of relations:

1. Stored relations, which are called *tables*. These are the kind of relation we deal with ordinarily — a relation that exists in the database and that can be modified by changing its tuples, as well as queried.
2. *Views*, which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed.

3. Temporary tables, which are constructed by the SQL language processor when it performs its job of executing queries and data modifications. These relations are then thrown away and not stored.

In this section, we shall learn how to declare tables. We do not treat the declaration and definition of views here, and temporary tables are never declared. The SQL `CREATE TABLE` statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation. There are many other features to the `CREATE TABLE` statement, including many forms of constraints that can be declared, and the declaration of *indexes* (data structures that speed up many operations on the table) but we shall leave those for the appropriate time.

3.2 Data Types

To begin, let us introduce the primitive data types that are supported by SQL systems. All attributes must have a data type.

1. Character strings of fixed or varying length. The type `CHAR(n)` denotes a fixed-length string of up to n characters. `VARCHAR(n)` also denotes a string of up to n characters. The difference is implementation-dependent; typically `CHAR` implies that short strings are padded to make n characters, while `VARCHAR` implies that an endmarker or string-length is used. SQL permits reasonable coercions between values of character-string types. Normally, a string is padded by trailing blanks if it becomes the value of a component that is a fixed-length string of greater length. For example, the string `'foo'`,[2] if it became the value of a component for an attribute of type `CHAR(5)`, would assume the value `'foo  '` (with two blanks following the second `o`).

2. Bit strings of fixed or varying length. These strings are analogous to fixed and varying-length character strings, but their values are strings of bits rather than characters. The type `BIT(n)` denotes bit strings of length n, while `BIT VARYING(n)` denotes bit strings of length up to n.

3. The type `BOOLEAN` denotes an attribute whose value is logical. The possible values of such an attribute are `TRUE`, `FALSE`, and — although it would surprise George Boole — `UNKNOWN`.

4. The type `INT` or `INTEGER` (these names are synonyms) denotes typical integer values. The type `SHORTINT` also denotes integers, but the number of bits permitted may be less, depending on the implementation (as with the types `int` and `short int` in C).

[2]Notice that in SQL, strings are surrounded by single-quotes, not double-quotes as in many other programming languages.

Dates and Times in SQL

Different SQL implementations may provide many different representations for dates and times, but the following is the SQL standard representation. A date value is the keyword `DATE` followed by a quoted string of a special form. For example, `DATE '1948-05-14'` follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Finally there is another hyphen and two digits representing the day. Note that single-digit months and days are padded with a leading 0.

A time value is the keyword `TIME` and a quoted string. This string has two digits for the hour, on the military (24-hour) clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, `TIME '15:00:02.5'` represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

5. Floating-point numbers can be represented in a variety of ways. We may use the type `FLOAT` or `REAL` (these are synonyms) for typical floating-point numbers. A higher precision can be obtained with the type `DOUBLE PRECISION`; again the distinction between these types is as in C. SQL also has types that are real numbers with a fixed decimal point. For example, `DECIMAL(n,d)` allows values that consist of n decimal digits, with the decimal point assumed to be d positions from the right. Thus, 0123.45 is a possible value of type `DECIMAL(6,2)`. `NUMERIC` is almost a synonym for `DECIMAL`, although there are possible implementation-dependent differences.

6. Dates and times can be represented by the data types `DATE` and `TIME`, respectively (see the box on "Dates and Times in SQL"). These values are essentially character strings of a special form. We may, in fact, coerce dates and times to string types, and we may do the reverse if the string "makes sense" as a date or time.

3.3 Simple Table Declarations

The simplest form of declaration of a relation schema consists of the keywords `CREATE TABLE` followed by the name of the relation and a parenthesized, comma-separated list of the attribute names and their types.

Example 2: The relation `Movies` with the schema given in Fig. 5 can be declared as in Fig. 7. The title is declared as a string of (up to) 100 characters.

```
CREATE TABLE Movies (
    title       CHAR(100),
    year        INT,
    length      INT,
    genre       CHAR(10),
    studioName  CHAR(30),
    producerC#  INT
);
```

Figure 7: SQL declaration of the table `Movies`

The year and length attributes are each integers, and the genre is a string of (up to) 10 characters. The decision to allow up to 100 characters for a title is arbitrary, but we don't want to limit the lengths of titles too strongly, or long titles would be truncated to fit. We have assumed that 10 characters are enough to represent a genre of movie; again, that is an arbitrary choice, one we could regret if we had a genre with a long name. Likewise, we have chosen 30 characters as sufficient for the studio name. The certificate number for the producer of the movie is another integer. □

Example 3: Figure 8 is a SQL declaration of the relation `MovieStar` from Fig. 5. It illustrates some new options for data types. The name of this table is `MovieStar`, and it has four attributes. The first two attributes, `name` and `address`, have each been declared to be character strings. However, with the name, we have made the decision to use a fixed-length string of 30 characters, padding a name out with blanks at the end if necessary and truncating a name to 30 characters if it is longer. In contrast, we have declared addresses to be variable-length character strings of up to 255 characters.[3] It is not clear that these two choices are the best possible, but we use them to illustrate the two major kinds of string data types.

```
CREATE TABLE MovieStar (
    name      CHAR(30),
    address   VARCHAR(255),
    gender    CHAR(1),
    birthdate DATE
);
```

Figure 8: Declaring the relation schema for the `MovieStar` relation

[3]The number 255 is not the result of some weird notion of what typical addresses look like. A single byte can store integers between 0 and 255, so it is possible to represent a varying-length character string of up to 255 bytes by a single byte for the count of characters plus the bytes to store the string itself. Commercial systems generally support longer varying-length strings, however.

The `gender` attribute has values that are a single letter, `M` or `F`. Thus, we can safely use a single character as the type of this attribute. Finally, the `birthdate` attribute naturally deserves the data type `DATE`. □

3.4 Modifying Relation Schemas

We now know how to declare a table. But what if we need to change the schema of the table after it has been in use for a long time and has many tuples in its current instance? We can remove the entire table, including all of its current tuples, or we could change the schema by adding or deleting attributes.

We can delete a relation R by the SQL statement:

```
DROP TABLE R;
```

Relation R is no longer part of the database schema, and we can no longer access any of its tuples.

More frequently than we would drop a relation that is part of a long-lived database, we may need to modify the schema of an existing relation. These modifications are done by a statement that begins with the keywords `ALTER TABLE` and the name of the relation. We then have several options, the most important of which are

1. `ADD` followed by an attribute name and its data type.

2. `DROP` followed by an attribute name.

Example 4: Thus, for instance, we could modify the `MovieStar` relation by adding an attribute `phone` with:

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

As a result, the `MovieStar` schema now has five attributes: the four mentioned in Fig. 8 and the attribute `phone`, which is a fixed-length string of 16 bytes. In the actual relation, tuples would all have components for `phone`, but we know of no phone numbers to put there. Thus, the value of each of these components is set to the special *null value*, `NULL`. In Section 3.5, we shall see how it is possible to choose another "default" value to be used instead of `NULL` for unknown values.

As another example, the `ALTER TABLE` statement:

```
ALTER TABLE MovieStar DROP birthdate;
```

deletes the `birthdate` attribute. As a result, the schema for `MovieStar` no longer has that attribute, and all tuples of the current `MovieStar` instance have the component for `birthdate` deleted. □

3.5 Default Values

When we create or modify tuples, we sometimes do not have values for all components. For instance, we mentioned in Example 4 that when we add a column to a relation schema, the existing tuples do not have a known value, and it was suggested that `NULL` could be used in place of a "real" value. However, there are times when we would prefer to use another choice of *default* value, the value that appears in a column if no other value is known.

In general, any place we declare an attribute and its data type, we may add the keyword `DEFAULT` and an appropriate value. That value is either `NULL` or a constant. Certain other values that are provided by the system, such as the current time, may also be options.

Example 5: Let us consider Example 3. We might wish to use the character `?` as the default for an unknown `gender`, and we might also wish to use the earliest possible date, `DATE '0000-00-00'` for an unknown `birthdate`. We could replace the declarations of `gender` and `birthdate` in Fig. 8 by:

```
gender CHAR(1) DEFAULT '?',
birthdate DATE DEFAULT DATE '0000-00-00'
```

As another example, we could have declared the default value for new attribute `phone` to be `'unlisted'` when we added this attribute in Example 4. In that case,

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

would be the appropriate `ALTER TABLE` statement. □

3.6 Declaring Keys

There are two ways to declare an attribute or set of attributes to be a key in the `CREATE TABLE` statement that defines a stored relation.

1. We may declare one attribute to be a key when that attribute is listed in the relation schema.
2. We may add to the list of items declared in the schema (which so far have only been attributes) an additional declaration that says a particular attribute or set of attributes forms the key.

If the key consists of more than one attribute, we have to use method (2). If the key is a single attribute, either method may be used.

There are two declarations that may be used to indicate keyness:

a) `PRIMARY KEY`, or

b) `UNIQUE`.

The effect of declaring a set of attributes S to be a key for relation R either using `PRIMARY KEY` or `UNIQUE` is the following:

- Two tuples in R cannot agree on all of the attributes in set S, unless one of them is `NULL`. Any attempt to insert or update a tuple that violates this rule causes the DBMS to reject the action that caused the violation.

In addition, if `PRIMARY KEY` is used, then attributes in S are not allowed to have `NULL` as a value for their components. Again, any attempt to violate this rule is rejected by the system. `NULL` is permitted if the set S is declared `UNIQUE`, however. A DBMS make make other distinctions between the two terms, if it wishes.

Example 6: Let us reconsider the schema for relation `MovieStar`. Since no star would use the name of another star, we shall assume that `name` by itself forms a key for this relation. Thus, we can add this fact to the line declaring `name`. Figure 9 is a revision of Fig. 8 that reflects this change. We could also substitute `UNIQUE` for `PRIMARY KEY` in this declaration. If we did so, then two or more tuples could have `NULL` as the value of `name`, but there could be no other duplicate values for this attribute.

```
CREATE TABLE MovieStar (
   name CHAR(30) PRIMARY KEY,
   address VARCHAR(255),
   gender CHAR(1),
   birthdate DATE
);
```

Figure 9: Making `name` the key

Alternatively, we can use a separate definition of the key. The resulting schema declaration would look like Fig. 10. Again, `UNIQUE` could replace `PRIMARY KEY`. □

```
CREATE TABLE MovieStar (
   name CHAR(30),
   address VARCHAR(255),
   gender CHAR(1),
   birthdate DATE,
   PRIMARY KEY (name)
);
```

Figure 10: A separate declaration of the key

Example 7: In Example 6, the form of either Fig. 9 or Fig. 10 is acceptable, because the key is a single attribute. However, in a situation where the key has more than one attribute, we must use the style of Fig. 10. For instance, the relation `Movie`, whose key is the pair of attributes `title` and `year`, must be declared as in Fig. 11. However, as usual, `UNIQUE` is an option to replace `PRIMARY KEY`. □

```
CREATE TABLE Movies (
    title       CHAR(100),
    year        INT,
    length      INT,
    genre       CHAR(10),
    studioName  CHAR(30),
    producerC#  INT,
    PRIMARY KEY (title, year)
);
```

Figure 11: Making `title` and `year` be the key of `Movies`

3.7 Exercises for Section 3

Exercise 3.1: In this exercise we introduce one of our running examples of a relational database schema. The database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

The `Product` relation gives the manufacturer, model number and type (PC, laptop, or printer) of various products. We assume for convenience that model numbers are unique over all manufacturers and product types; that assumption is not realistic, and a real database would include a code for the manufacturer as part of the model number. The `PC` relation gives for each model number that is a PC the speed (of the processor, in gigahertz), the amount of RAM (in megabytes), the size of the hard disk (in gigabytes), and the price. The `Laptop` relation is similar, except that the screen size (in inches) is also included. The `Printer` relation records for each printer model whether the printer produces color output (true, if so), the process type (laser or ink-jet, typically), and the price.

Write the following declarations:

a) A suitable schema for relation `Product`.

b) A suitable schema for relation `PC`.

c) A suitable schema for relation `Laptop`.

d) A suitable schema for relation `Printer`.

e) An alteration to your `Printer` schema from (d) to delete the attribute `color`.

f) An alteration to your `Laptop` schema from (c) to add the attribute `od` (optical-disk type, e.g., cd or dvd). Let the default value for this attribute be `'none'` if the laptop does not have an optical disk.

Exercise 3.2: This exercise introduces another running example, concerning World War II capital ships. It involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Ships are built in "classes" from the same design, and the class is usually named for the first ship of that class. The relation `Classes` records the name of the class, the type (`'bb'` for battleship or `'bc'` for battlecruiser), the country that built the ship, the number of main guns, the bore (diameter of the gun barrel, in inches) of the main guns, and the displacement (weight, in tons). Relation `Ships` records the name of the ship, the name of its class, and the year in which the ship was launched. Relation `Battles` gives the name and date of battles involving these ships, and relation `Outcomes` gives the result (sunk, damaged, or ok) for each ship in each battle.

Write the following declarations:

a) A suitable schema for relation `Classes`.

b) A suitable schema for relation `Ships`.

c) A suitable schema for relation `Battles`.

d) A suitable schema for relation `Outcomes`.

e) An alteration to your `Classes` relation from (a) to delete the attribute `bore`.

f) An alteration to your `Ships` relation from (b) to include the attribute `yard` giving the shipyard where the ship was built.

4 An Algebraic Query Language

In this section, we introduce the data-manipulation aspect of the relational model. Recall that a data model is not just structure; it needs a way to query the data and to modify the data. To begin our study of operations on relations, we shall learn about a special algebra, called *relational algebra*, that consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

Relational algebra is not used today as a query language in commercial DBMS's, although some of the early prototypes did use this algebra directly. Rather, the "real" query language, SQL, incorporates relational algebra at its center, and many SQL programs are really "syntactically sugared" expressions of relational algebra. Further, when a DBMS processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation. Thus, there are several good reasons to start out learning this algebra.

4.1 Why Do We Need a Special Query Language?

Before introducing the operations of relational algebra, one should ask why, or whether, we need a new kind of programming languages for databases. Won't conventional languages like C or Java suffice to ask and answer any computable question about relations? After all, we can represent a tuple of a relation by a struct (in C) or an object (in Java), and we can represent relations by arrays of these elements.

The surprising answer is that relational algebra is useful because it is *less* powerful than C or Java. That is, there are computations one can perform in any conventional language that one cannot perform in relational algebra. An example is: determine whether the number of tuples in a relation is even or odd. By limiting what we can say or do in our query language, we get two huge rewards — ease of programming and the ability of the compiler to produce highly optimized code — that we discussed in Section 1.6.

4.2 What is an Algebra?

An algebra, in general, consists of operators and atomic operands. For instance, in the algebra of arithmetic, the atomic operands are variables like x and constants like 15. The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division. Any algebra allows us to build *expressions* by applying operators to atomic operands and/or other expressions of the algebra. Usually, parentheses are needed to group operators and their operands. For instance, in arithmetic we have expressions such as $(x + y) * z$ or $\big((x + 7)/(y - 3)\big) + x$.

Relational algebra is another example of an algebra. Its atomic operands are:

1. Variables that stand for relations.
2. Constants, which are finite relations.

We shall next see the operators of relational algebra.

4.3 Overview of Relational Algebra

The operations of the traditional relational algebra fall into four broad classes:

a) The usual set operations — union, intersection, and difference — applied to relations.

b) Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.

c) Operations that combine the tuples of two relations, including "Cartesian product," which pairs the tuples of two relations in all possible ways, and various kinds of "join" operations, which selectively pair tuples from two relations.

d) An operation called "renaming" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

We generally shall refer to expressions of relational algebra as *queries*.

4.4 Set Operations on Relations

The three most common operations on sets are union, intersection, and difference. We assume the reader is familiar with these operations, which are defined as follows on arbitrary sets R and S:

- $R \cup S$, the *union* of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.
- $R \cap S$, the *intersection* of R and S, is the set of elements that are in both R and S.
- $R - S$, the *difference* of R and S, is the set of elements that are in R but not in S. Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R.

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.

2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Sometimes we would like to take the union, intersection, or difference of relations that have the same number of attributes, with corresponding domains, but that use different names for their attributes. If so, we may use the renaming operator to be discussed in Section 4.11 to change the schema of one or both relations and give them the same set of attributes.

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

Figure 12: Two relations

Example 8: Suppose we have the two relations R and S, whose schemas are both that of relation `MovieStar` Section 2.8. Current instances of R and S are shown in Fig. 12. Then the union $R \cup S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

The intersection $R \cap S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. □

4.5 Projection

The *projection* operator is used to produce from a relation R a new relation that has only some of R's columns. The value of expression $\pi_{A_1,A_2,\ldots,A_n}(R)$ is a relation that has only the columns for attributes $A_1, A_2, \ldots, A_n$ of R. The schema for the resulting value is the set of attributes $\{A_1, A_2, \ldots, A_n\}$, which we conventionally show in the order listed.

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890
Wayne's World	1992	95	comedy	Paramount	99999

Figure 13: The relation Movies

Example 9: Consider the relation Movies with the relation schema described in Section 2.8. An instance of this relation is shown in Fig. 13. We can project this relation onto the first three attributes with the expression:

$$\pi_{title,year,length}(\texttt{Movies})$$

The resulting relation is

title	*year*	*length*
Star Wars	1977	124
Galaxy Quest	1999	104
Wayne's World	1992	95

As another example, we can project onto the attribute genre with the expression $\pi_{genre}(\texttt{Movies})$. The result is the single-column relation

genre
sciFi
comedy

Notice that there are only two tuples in the resulting relation, since the last two tuples of Fig. 13 have the same value in their component for attribute genre, and in the relational algebra of sets, duplicate tuples are always eliminated. □

A Note About Data Quality :-)

While we have endeavored to make example data as accurate as possible, we have used bogus values for addresses and other personal information about movie stars, in order to protect the privacy of members of the acting profession, many of whom are shy individuals who shun publicity.

4.6 Selection

The *selection* operator, applied to a relation R, produces a new relation with a subset of R's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R. We denote this operation $\sigma_C(R)$. The schema for the resulting relation is the same as R's schema, and we conventionally show the attributes in the same order as we use for R.

C is a conditional expression of the type with which we are familiar from conventional programming languages; for example, conditional expressions follow the keyword `if` in programming languages such as C or Java. The only difference is that the operands in condition C are either constants or attributes of R. We apply C to each tuple t of R by substituting, for each attribute A appearing in condition C, the component of t for attribute A. If after substituting for each attribute of C the condition C is true, then t is one of the tuples that appear in the result of $\sigma_C(R)$; otherwise t is not in the result.

Example 10: Let the relation `Movies` be as in Fig. 13. Then the value of expression $\sigma_{length \geq 100}(\texttt{Movies})$ is

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890

The first tuple satisfies the condition $length \geq 100$ because when we substitute for *length* the value 124 found in the component of the first tuple for attribute `length`, the condition becomes $124 \geq 100$. The latter condition is true, so we accept the first tuple. The same argument explains why the second tuple of Fig. 13 is in the result.

The third tuple has a `length` component 95. Thus, when we substitute for *length* we get the condition $95 \geq 100$, which is false. Hence the last tuple of Fig. 13 is not in the result. □

Example 11: Suppose we want the set of tuples in the relation `Movies` that represent Fox movies at least 100 minutes long. We can get these tuples with a more complicated condition, involving the `AND` of two subconditions. The expression is

$$\sigma_{length \geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})$$

The tuple

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345

is the only one in the resulting relation. □

4.7 Cartesian Product

The *Cartesian product* (or *cross-product*, or just *product*) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S. This product is denoted $R \times S$. When R and S are relations, the product is essentially the same. However, since the members of R and S are tuples, usually consisting of more than one component, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples. By convention, the components from R (the left operand) precede the components from S in the attribute order for the result.

The relation schema for the resulting relation is the union of the schemas for R and S. However, if R and S should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes. To disambiguate an attribute A that is in the schemas of both R and S, we use $R.A$ for the attribute from R and $S.A$ for the attribute from S.

Example 12: For conciseness, let us use an abstract example that illustrates the product operation. Let relations R and S have the schemas and tuples shown in Fig. 14(a) and (b). Then the product $R \times S$ consists of the six tuples shown in Fig. 14(c). Note how we have paired each of the two tuples of R with each of the three tuples of S. Since B is an attribute of both schemas, we have used $R.B$ and $S.B$ in the schema for $R \times S$. The other attributes are unambiguous, and their names appear in the resulting schema unchanged. □

4.8 Natural Joins

More often than we want to take the product of two relations, we find a need to *join* them by pairing only those tuples that match in some way. The simplest sort of match is the *natural join* of two relations R and S, denoted $R \bowtie S$, in which we pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S. More precisely, let $A_1, A_2, \ldots, A_n$ be all the attributes that are in both the schema of R and the schema of S. Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, \ldots, A_n$.

If the tuples r and s are successfully paired in the join $R \bowtie S$, then the result of the pairing is a tuple, called the *joined tuple*, with one component for each of the attributes in the union of the schemas of R and S. The joined tuple

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

(c) Result $R \times S$

Figure 14: Two relations and their Cartesian product

agrees with tuple r in each attribute in the schema of R, and it agrees with s in each attribute in the schema of S. Since r and s are successfully paired, the joined tuple is able to agree with both these tuples on the attributes they have in common. The construction of the joined tuple is suggested by Fig. 15. However, the order of the attributes need not be that convenient; the attributes of R and S can appear in any order.

Example 13: The natural join of the relations R and S from Fig. 14(a) and (b) is

A	B	C	D
1	2	5	6
3	4	7	8

The only attribute common to R and S is B. Thus, to pair successfully, tuples need only to agree in their B components. If so, the resulting tuple has components for attributes A (from R), B (from either R or S), C (from S), and D (from S).

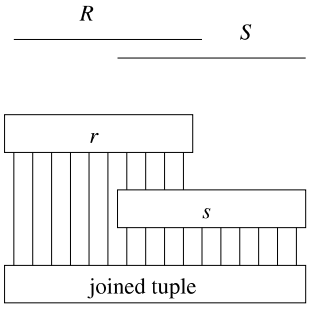

Figure 15: Joining tuples

In this example, the first tuple of R successfully pairs with only the first tuple of S; they share the value 2 on their common attribute B. This pairing yields the first tuple of the result: $(1, 2, 5, 6)$. The second tuple of R pairs successfully only with the second tuple of S, and the pairing yields $(3, 4, 7, 8)$. Note that the third tuple of S does not pair with any tuple of R and thus has no effect on the result of $R \bowtie S$. A tuple that fails to pair with any tuple of the other relation in a join is said to be a *dangling tuple*. □

Example 14: The previous example does not illustrate all the possibilities inherent in the natural join operator. For example, no tuple paired successfully with more than one tuple, and there was only one attribute in common to the two relation schemas. In Fig. 16 we see two other relations, U and V, that share two attributes between their schemas: B and C. We also show an instance in which one tuple joins with several tuples.

For tuples to pair successfully, they must agree in both the B and C components. Thus, the first tuple of U joins with the first two tuples of V, while the second and third tuples of U join with the third tuple of V. The result of these four pairings is shown in Fig. 16(c). □

4.9 Theta-Joins

The natural join forces us to pair tuples using one specific condition. While this way, equating shared attributes, is the most common basis on which relations are joined, it is sometimes desirable to pair tuples from two relations on some other basis. For that purpose, we have a related notation called the *theta-join*. Historically, the "theta" refers to an arbitrary condition, which we shall represent by C rather than θ.

The notation for a theta-join of relations R and S based on condition C is $R \bowtie_C S$. The result of this operation is constructed as follows:

1. Take the product of R and S.
2. Select from the product only those tuples that satisfy the condition C.

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

Figure 16: Natural join of relations

As with the product operation, the schema for the result is the union of the schemas of R and S, with "R." or "S." prefixed to attributes if necessary to indicate from which schema the attribute came.

Example 15: Consider the operation $U \bowtie_{A<D} V$, where U and V are the relations from Fig. 16(a) and (b). We must consider all nine pairs of tuples, one from each relation, and see whether the A component from the U-tuple is less than the D component of the V-tuple. The first tuple of U, with an A component of 1, successfully pairs with each of the tuples from V. However, the second and third tuples from U, with A components of 6 and 9, respectively, pair successfully with only the last tuple of V. Thus, the result has only five tuples, constructed from the five successful pairings. This relation is shown in Fig. 17. □

Notice that the schema for the result in Fig. 17 consists of all six attributes, with U and V prefixed to their respective occurrences of attributes B and C to distinguish them. Thus, the theta-join contrasts with natural join, since in the latter common attributes are merged into one copy. Of course it makes sense to

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Figure 17: Result of $U \bowtie_{A<D} V$

do so in the case of the natural join, since tuples don't pair unless they agree in their common attributes. In the case of a theta-join, there is no guarantee that compared attributes will agree in the result, since they may not be compared with $=$.

Example 16: Here is a theta-join on the same relations U and V that has a more complex condition:

$$U \bowtie_{A<D \text{ AND } U.B \neq V.B} V$$

That is, we require for successful pairing not only that the A component of the U-tuple be less than the D component of the V-tuple, but that the two tuples disagree on their respective B components. The tuple

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

is the only one to satisfy both conditions, so this relation is the result of the theta-join above. □

4.10 Combining Operations to Form Queries

If all we could do was to write single operations on one or two relations as queries, then relational algebra would not be nearly as useful as it is. However, relational algebra, like all algebras, allows us to form expressions of arbitrary complexity by applying operations to the result of other operations.

One can construct expressions of relational algebra by applying operators to subexpressions, using parentheses when necessary to indicate grouping of operands. It is also possible to represent expressions as expression trees; the latter often are easier for us to read, although they are less convenient as a machine-readable notation.

Example 17: Suppose we want to know, from our running `Movies` relation, "What are the titles and years of movies made by Fox that are at least 100 minutes long?" One way to compute the answer to this query is:

1. Select those `Movies` tuples that have $length \geq 100$.

2. Select those `Movies` tuples that have $studioName =$ `'Fox'`.

3. Compute the intersection of (1) and (2).

4. Project the relation from (3) onto attributes `title` and `year`.

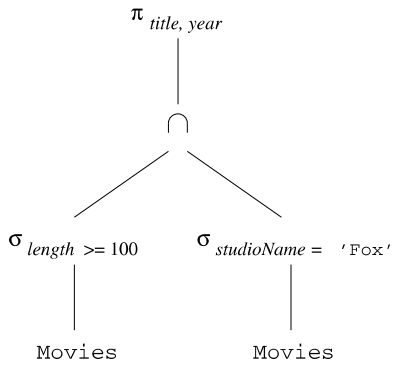

Figure 18: Expression tree for a relational algebra expression

In Fig. 18 we see the above steps represented as an expression tree. Expression trees are evaluated bottom-up by applying the operator at an interior node to the arguments, which are the results of its children. By proceeding bottom-up, we know that the arguments will be available when we need them. The two selection nodes correspond to steps (1) and (2). The intersection node corresponds to step (3), and the projection node is step (4).

Alternatively, we could represent the same expression in a conventional, linear notation, with parentheses. The formula

$$\pi_{title,year}\Big(\sigma_{length\geq 100}(\texttt{Movies}) \cap \sigma_{studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

represents the same expression.

Incidentally, there is often more than one relational algebra expression that represents the same computation. For instance, the above query could also be written by replacing the intersection by logical `AND` within a single selection operation. That is,

$$\pi_{title,year}\Big(\sigma_{length\geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

is an equivalent form of the query. □

Equivalent Expressions and Query Optimization

All database systems have a query-answering system, and many of them are based on a language that is similar in expressive power to relational algebra. Thus, the query asked by a user may have many *equivalent expressions* (expressions that produce the same answer whenever they are given the same relations as operands), and some of these may be much more quickly evaluated. An important job of the query "optimizer" is to replace one expression of relational algebra by an equivalent expression that is more efficiently evaluated.

4.11 Naming and Renaming

In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations. We shall use the operator $\rho_{S(A_1,A_2,\ldots,A_n)}(R)$ to rename a relation R. The resulting relation has exactly the same tuples as R, but the name of the relation is S. Moreover, the attributes of the result relation S are named $A_1, A_2, \ldots, A_n$, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say $\rho_S(R)$.

Example 18: In Example 12 we took the product of two relations R and S from Fig. 14(a) and (b) and used the convention that when an attribute appears in both operands, it is renamed by prefixing the relation name to it. Suppose, however, that we do not wish to call the two versions of B by names $R.B$ and $S.B$; rather we want to continue to use the name B for the attribute that comes from R, and we want to use X as the name of the attribute B coming from S. We can rename the attributes of S so the first is called X. The result of the expression $\rho_{S(X,C,D)}(S)$ is a relation named S that looks just like the relation S from Fig. 14, but its first column has attribute X instead of B.

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Figure 19: $R \times \rho_{S(X,C,D)}(S)$

When we take the product of R with this new relation, there is no conflict of names among the attributes, so no further renaming is done. That is, the result of the expression $R \times \rho_{S(X,C,D)}(S)$ is the relation $R \times S$ from Fig. 14(c), except that the five columns are labeled A, B, X, C, and D, from the left. This relation is shown in Fig. 19.

As an alternative, we could take the product without renaming, as we did in Example 12, and then rename the result. The expression

$$\rho_{RS(A,B,X,C,D)}(R \times S)$$

yields the same relation as in Fig. 19, with the same set of attributes. But this relation has a name, RS, while the result relation in Fig. 19 has no name. □

4.12 Relationships Among Operations

Some of the operations that we have described in Section 4 can be expressed in terms of other relational-algebra operations. For example, intersection can be expressed in terms of set difference:

$$R \cap S = R - (R - S)$$

That is, if R and S are any two relations with the same schema, the intersection of R and S can be computed by first subtracting S from R to form a relation T consisting of all those tuples in R but not S. We then subtract T from R, leaving only those tuples of R that are also in S.

The two forms of join are also expressible in terms of other operations. Theta-join can be expressed by product and selection:

$$R \bowtie_C S = \sigma_C(R \times S)$$

The natural join of R and S can be expressed by starting with the product $R \times S$. We then apply the selection operator with a condition C of the form

$$R.A_1 = S.A_1 \texttt{ AND } R.A_2 = S.A_2 \texttt{ AND} \cdots \texttt{AND } R.A_n = S.A_n$$

where $A_1, A_2, \ldots, A_n$ are all the attributes appearing in the schemas of both R and S. Finally, we must project out one copy of each of the equated attributes. Let L be the list of attributes in the schema of R followed by those attributes in the schema of S that are not also in the schema of R. Then

$$R \bowtie S = \pi_L\Big(\sigma_C(R \times S)\Big)$$

Example 19: The natural join of the relations U and V from Fig. 16 can be written in terms of product, selection, and projection as:

$$\pi_{A,U.B,U.C,D}\Big(\sigma_{U.B=V.B \texttt{ AND } U.C=V.C}(U \times V)\Big)$$

That is, we take the product $U \times V$. Then we select for equality between each pair of attributes with the same name — B and C in this example. Finally, we project onto all the attributes except one of the B's and one of the C's; we have chosen to eliminate the attributes of V whose names also appear in the schema of U.

For another example, the theta-join of Example 16 can be written

$$\sigma_{A<D \text{ AND } U.B \neq V.B}(U \times V)$$

That is, we take the product of the relations U and V and then apply the condition that appeared in the theta-join. □

The rewriting rules mentioned in this section are the only "redundancies" among the operations that we have introduced. The six remaining operations — union, difference, selection, projection, product, and renaming — form an independent set, none of which can be written in terms of the other five.

4.13 A Linear Notation for Algebraic Expressions

In Section 4.10 we used an expression tree to represent a complex expression of relational algebra. An alternative is to invent names for the temporary relations that correspond to the interior nodes of the tree and write a sequence of assignments that create a value for each. The order of the assignments is flexible, as long as the children of a node N have had their values created before we attempt to create the value for N itself.

The notation we shall use for assignment statements is:

1. A relation name and parenthesized list of attributes for that relation. The name `Answer` will be used conventionally for the result of the final step; i.e., the name of the relation at the root of the expression tree.

2. The assignment symbol `:=`.

3. Any algebraic expression on the right. We can choose to use only one operator per assignment, in which case each interior node of the tree gets its own assignment statement. However, it is also permissible to combine several algebraic operations in one right side, if it is convenient to do so.

Example 20: Consider the tree of Fig. 18. One possible sequence of assignments to evaluate this expression is:

R(t,y,l,i,s,p) := $\sigma_{length \geq 100}$(Movies)
S(t,y,l,i,s,p) := $\sigma_{studioName='\texttt{Fox}'}$(Movies)
T(t,y,l,i,s,p) := R ∩ S
Answer(title, year) := $\pi_{t,y}$(T)

The first step computes the relation of the interior node labeled $\sigma_{length \geq 100}$ in Fig. 18, and the second step computes the node labeled $\sigma_{studioName='\texttt{Fox}'}$. Notice that we get renaming "for free," since we can use any attributes and relation name we wish for the left side of an assignment. The last two steps compute the intersection and the projection in the obvious way.

It is also permissible to combine some of the steps. For instance, we could combine the last two steps and write:

$$\texttt{R(t,y,l,i,s,p)} := \sigma_{length \geq 100}(\texttt{Movies})$$
$$\texttt{S(t,y,l,i,s,p)} := \sigma_{studioName='\texttt{Fox}'}(\texttt{Movies})$$
$$\texttt{Answer(title, year)} := \pi_{t,y}(\texttt{R} \cap \texttt{S})$$

We could even substitute for R and S in the last line and write the entire expression in one line. □

4.14 Exercises for Section 4

Exercise 4.1: This exercise builds upon the products schema of Exercise 3.1. Recall that the database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

Some sample data for the relation `Product` is shown in Fig. 20. Sample data for the other three relations is shown in Fig. 21. Manufacturers and model numbers have been "sanitized," but the data is typical of products on sale at the beginning of 2007.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 4.13 if you wish. For the data of Figs. 20 and 21, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

a) What PC models have a speed of at least 3.00?

b) Which manufacturers make laptops with a hard disk of at least 100GB?

c) Find the model number and price of all products (of any type) made by manufacturer B.

d) Find the model numbers of all color laser printers.

e) Find those manufacturers that sell Laptops, but not PC's.

! f) Find those hard-disk sizes that occur in two or more PC's.

maker	*model*	*type*
A	1001	pc
A	1002	pc
A	1003	pc
A	2004	laptop
A	2005	laptop
A	2006	laptop
B	1004	pc
B	1005	pc
B	1006	pc
B	2007	laptop
C	1007	pc
D	1008	pc
D	1009	pc
D	1010	pc
D	3004	printer
D	3005	printer
E	1011	pc
E	1012	pc
E	1013	pc
E	2001	laptop
E	2002	laptop
E	2003	laptop
E	3001	printer
E	3002	printer
E	3003	printer
F	2008	laptop
F	2009	laptop
G	2010	laptop
H	3006	printer
H	3007	printer

Figure 20: Sample data for `Product`

model	*speed*	*ram*	*hd*	*price*
1001	2.66	1024	250	2114
1002	2.10	512	250	995
1003	1.42	512	80	478
1004	2.80	1024	250	649
1005	3.20	512	250	630
1006	3.20	1024	320	1049
1007	2.20	1024	200	510
1008	2.20	2048	250	770
1009	2.00	1024	250	650
1010	2.80	2048	300	770
1011	1.86	2048	160	959
1012	2.80	1024	160	649
1013	3.06	512	80	529

(a) Sample data for relation `PC`

model	*speed*	*ram*	*hd*	*screen*	*price*
2001	2.00	2048	240	20.1	3673
2002	1.73	1024	80	17.0	949
2003	1.80	512	60	15.4	549
2004	2.00	512	60	13.3	1150
2005	2.16	1024	120	17.0	2500
2006	2.00	2048	80	15.4	1700
2007	1.83	1024	120	13.3	1429
2008	1.60	1024	100	15.4	900
2009	1.60	512	80	14.1	680
2010	2.00	2048	160	15.4	2300

(b) Sample data for relation `Laptop`

model	*color*	*type*	*price*
3001	true	ink-jet	99
3002	false	laser	239
3003	true	laser	899
3004	true	ink-jet	120
3005	false	laser	120
3006	true	ink-jet	100
3007	true	laser	200

(c) Sample data for relation `Printer`

Figure 21: Sample data for relations of Exercise 4.1

! g) Find those pairs of PC models that have both the same speed and RAM. A pair should be listed only once; e.g., list (i, j) but not (j, i).

!! h) Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 2.80.

!! i) Find the manufacturer(s) of the computer (PC or laptop) with the highest available speed.

!! j) Find the manufacturers of PC's with at least three different speeds.

!! k) Find the manufacturers who sell exactly three different models of PC.

Exercise 4.2: Draw expression trees for each of your expressions of Exercise 4.1.

Exercise 4.3: This exercise builds upon Exercise 3.2 concerning World War II capital ships. Recall it involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Figures 22 and 23 give some sample data for these four relations.[4] Note that, unlike the data for Exercise 4.1, there are some "dangling tuples" in this data, e.g., ships mentioned in `Outcomes` that are not mentioned in `Ships`.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 4.13 if you wish. For the data of Figs. 22 and 23, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

a) Give the class names and countries of the classes that carried guns of at least 16-inch bore.

b) Find the ships launched prior to 1921.

c) Find the ships sunk in the battle of the Denmark Strait.

d) The treaty of Washington in 1921 prohibited capital ships heavier than 35,000 tons. List the ships that violated the treaty of Washington.

e) List the name, displacement, and number of guns of the ships engaged in the battle of Guadalcanal.

f) List all the capital ships mentioned in the database. (Remember that all these ships may not appear in the `Ships` relation.)

[4]Source: J. N. Westwood, *Fighting Ships of World War II*, Follett Publishing, Chicago, 1975 and R. C. Stern, *US Battleships in Action*, Squadron/Signal Publications, Carrollton, TX, 1980.

class	*type*	*country*	*numGuns*	*bore*	*displacement*
Bismarck	bb	Germany	8	15	42000
Iowa	bb	USA	9	16	46000
Kongo	bc	Japan	8	14	32000
North Carolina	bb	USA	9	16	37000
Renown	bc	Gt. Britain	6	15	32000
Revenge	bb	Gt. Britain	8	15	29000
Tennessee	bb	USA	12	14	32000
Yamato	bb	Japan	9	18	65000

(a) Sample data for relation `Classes`

name	*date*
Denmark Strait	5/24-27/41
Guadalcanal	11/15/42
North Cape	12/26/43
Surigao Strait	10/25/44

(b) Sample data for relation `Battles`

ship	*battle*	*result*
Arizona	Pearl Harbor	sunk
Bismarck	Denmark Strait	sunk
California	Surigao Strait	ok
Duke of York	North Cape	ok
Fuso	Surigao Strait	sunk
Hood	Denmark Strait	sunk
King George V	Denmark Strait	ok
Kirishima	Guadalcanal	sunk
Prince of Wales	Denmark Strait	damaged
Rodney	Denmark Strait	ok
Scharnhorst	North Cape	sunk
South Dakota	Guadalcanal	damaged
Tennessee	Surigao Strait	ok
Washington	Guadalcanal	ok
West Virginia	Surigao Strait	ok
Yamashiro	Surigao Strait	sunk

(c) Sample data for relation `Outcomes`

Figure 22: Data for Exercise 4.3

name	*class*	*launched*
California	Tennessee	1921
Haruna	Kongo	1915
Hiei	Kongo	1914
Iowa	Iowa	1943
Kirishima	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
North Carolina	North Carolina	1941
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
Tennessee	Tennessee	1920
Washington	North Carolina	1941
Wisconsin	Iowa	1944
Yamato	Yamato	1941

Figure 23: Sample data for relation Ships

! g) Find the classes that had only one ship as a member of that class.

! h) Find those countries that had both battleships and battlecruisers.

! i) Find those ships that "lived to fight another day"; they were damaged in one battle, but later fought in another.

Exercise 4.4: Draw expression trees for each of your expressions of Exercise 4.3.

Exercise 4.5: What is the difference between the natural join $R \bowtie S$ and the theta-join $R \bowtie_C S$ where the condition C is that $R.A = S.A$ for each attribute A appearing in the schemas of both R and S?

! **Exercise 4.6:** An operator on relations is said to be *monotone* if whenever we add a tuple to one of its arguments, the result contains all the tuples that it contained before adding the tuple, plus perhaps more tuples. Which of the operators described in this section are monotone? For each, either explain why it is monotone or give an example showing it is not.

! Exercise 4.7: Suppose relations R and S have n tuples and m tuples, respectively. Give the minimum and maximum numbers of tuples that the results of the following expressions can have.

a) $R \cup S$.

b) $R \bowtie S$.

c) $\sigma_C(R) \times S$, for some condition C.

d) $\pi_L(R) - S$, for some list of attributes L.

! Exercise 4.8: The *semijoin* of relations R and S, written $R \ltimes S$, is the set of tuples t in R such that there is at least one tuple in S that agrees with t in all attributes that R and S have in common. Give three different expressions of relational algebra that are equivalent to $R \ltimes S$.

! Exercise 4.9: The *antisemijoin* $R \overline{\ltimes} S$ is the set of tuples t in R that do *not* agree with any tuple of S in the attributes common to R and S. Give an expression of relational algebra equivalent to $R \overline{\ltimes} S$.

!! Exercise 4.10: Let R be a relation with schema

$$(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$$

and let S be a relation with schema $(B_1, B_2, \ldots, B_m)$; that is, the attributes of S are a subset of the attributes of R. The *quotient* of R and S, denoted $R \div S$, is the set of tuples t over attributes $A_1, A_2, \ldots, A_n$ (i.e., the attributes of R that are not attributes of S) such that for every tuple s in S, the tuple ts, consisting of the components of t for $A_1, A_2, \ldots, A_n$ and the components of s for $B_1, B_2, \ldots, B_m$, is a member of R. Give an expression of relational algebra, using the operators we have defined previously in this section, that is equivalent to $R \div S$.

5 Constraints on Relations

We now take up the third important aspect of a data model: the ability to restrict the data that may be stored in a database. So far, we have seen only one kind of constraint, the requirement that an attribute or attributes form a key (Section 3.6). These and many other kinds of constraints can be expressed in relational algebra. In this section, we show how to express both key constraints and "referential-integrity" constraints; the latter require that a value appearing in one column of one relation also appear in some other column of the same or a different relation. Know also that SQL database systems can enforce the same sorts of constraints as we can express in relational algebra.

5.1 Relational Algebra as a Constraint Language

There are two ways in which we can use expressions of relational algebra to express constraints.

1. If R is an expression of relational algebra, then $R = \emptyset$ is a constraint that says "The value of R must be empty," or equivalently "There are no tuples in the result of R."

2. If R and S are expressions of relational algebra, then $R \subseteq S$ is a constraint that says "Every tuple in the result of R must also be in the result of S." Of course the result of S may contain additional tuples not produced by R.

These ways of expressing constraints are actually equivalent in what they can express, but sometimes one or the other is clearer or more succinct. That is, the constraint $R \subseteq S$ could just as well have been written $R - S = \emptyset$. To see why, notice that if every tuple in R is also in S, then surely $R - S$ is empty. Conversely, if $R - S$ contains no tuples, then every tuple in R must be in S (or else it would be in $R - S$).

On the other hand, a constraint of the first form, $R = \emptyset$, could just as well have been written $R \subseteq \emptyset$. Technically, $\emptyset$ is not an expression of relational algebra, but since there are expressions that evaluate to $\emptyset$, such as $R - R$, there is no harm in using $\emptyset$ as a relational-algebra expression.

In the following sections, we shall see how to express significant constraints in one of these two styles. It is the first style — equal-to-the-emptyset — that is most commonly used in SQL programming. However, as shown above, we are free to think in terms of set-containment if we wish and later convert our constraint to the equal-to-the-emptyset style.

5.2 Referential Integrity Constraints

A common kind of constraint, called a *referential integrity constraint*, asserts that a value appearing in one context also appears in another, related context. For example, in our movies database, should we see a `StarsIn` tuple that has person p in the `starName` component, we would expect that p appears as the name of some star in the `MovieStar` relation. If not, then we would question whether the listed "star" really was a star.

In general, if we have any value v as the component in attribute A of some tuple in one relation R, then because of our design intentions we may expect that v will appear in a particular component (say for attribute B) of some tuple of another relation S. We can express this integrity constraint in relational algebra as $\pi_A(R) \subseteq \pi_B(S)$, or equivalently, $\pi_A(R) - \pi_B(S) = \emptyset$.

Example 21 : Consider the two relations from our running movie database:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

We might reasonably assume that the producer of every movie would have to appear in the `MovieExec` relation. If not, there is something wrong, and we would at least want a system implementing a relational database to inform us that we had a movie with a producer of which the database had no knowledge.

To be more precise, the `producerC#` component of each `Movies` tuple must also appear in the `cert#` component of some `MovieExec` tuple. Since executives are uniquely identified by their certificate numbers, we would thus be assured that the movie's producer is found among the movie executives. We can express this constraint by the set-containment

$$\pi_{producerC\#}(\texttt{Movies}) \subseteq \pi_{cert\#}(\texttt{MovieExec})$$

The value of the expression on the left is the set of all certificate numbers appearing in `producerC#` components of `Movies` tuples. Likewise, the expression on the right's value is the set of all certificates in the `cert#` component of `MovieExec` tuples. Our constraint says that every certificate in the former set must also be in the latter set. □

Example 22: We can similarly express a referential integrity constraint where the "value" involved is represented by more than one attribute. For instance, we may want to assert that any movie mentioned in the relation

```
StarsIn(movieTitle, movieYear, starName)
```

also appears in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

Movies are represented in both relations by title-year pairs, because we agreed that one of these attributes alone was not sufficient to identify a movie. The constraint

$$\pi_{movieTitle,\ movieYear}(\texttt{StarsIn}) \subseteq \pi_{title,\ year}(\texttt{Movies})$$

expresses this referential integrity constraint by comparing the title-year pairs produced by projecting both relations onto the appropriate lists of components. □

5.3 Key Constraints

The same constraint notation allows us to express far more than referential integrity. Here, we shall see how we can express algebraically the constraint that a certain attribute or set of attributes is a key for a relation.

Example 23: Recall that `name` is the key for relation

```
MovieStar(name, address, gender, birthdate)
```

That is, no two tuples agree on the `name` component. We shall express algebraically one of several implications of this constraint: that if two tuples agree on `name`, then they must also agree on `address`. Note that in fact these "two" tuples, which agree on the key `name`, must be the same tuple and therefore certainly agree in all attributes.

The idea is that if we construct all pairs of `MovieStar` tuples (t_1, t_2), we must not find a pair that agree in the `name` component and disagree in the `address` component. To construct the pairs we use a Cartesian product, and to search for pairs that violate the condition we use a selection. We then assert the constraint by equating the result to $\emptyset$.

To begin, since we are taking the product of a relation with itself, we need to rename at least one copy, in order to have names for the attributes of the product. For succinctness, let us use two new names, `MS1` and `MS2`, to refer to the `MovieStar` relation. Then the requirement can be expressed by the algebraic constraint:

$$\sigma_{MS1.name=MS2.name \text{ AND } MS1.address \neq MS2.address}(\texttt{MS1} \times \texttt{MS2}) = \emptyset$$

In the above, `MS1` in the product `MS1` $\times$ `MS2` is shorthand for the renaming:

$$\rho_{MS1(name,address,gender,birthdate)}(\texttt{MovieStar})$$

and `MS2` is a similar renaming of `MovieStar`. □

5.4 Additional Constraint Examples

There are many other kinds of constraints that we can express in relational algebra and that are useful for restricting database contents. A large family of constraints involve the permitted values in a context. For example, the fact that each attribute has a type constrains the values of that attribute. Often the constraint is quite straightforward, such as "integers only" or "character strings of length up to 30." Other times we want the values that may appear in an attribute to be restricted to a small enumerated set of values. Other times, there are complex limitations on the values that may appear. We shall give two examples, one of a simple *domain constraint* for an attribute, and the second a more complicated restriction.

Example 24: Suppose we wish to specify that the only legal values for the `gender` attribute of `MovieStar` are `'F'` and `'M'`. We can express this constraint algebraically by:

$$\sigma_{gender \neq \text{'F' AND } gender \neq \text{'M'}}(\texttt{MovieStar}) = \emptyset$$

That is, the set of tuples in `MovieStar` whose `gender` component is equal to neither `'F'` nor `'M'` is empty. □

Example 25: Suppose we wish to require that one must have a net worth of at least $10,000,000 to be the president of a movie studio. We can express this constraint algebraically as follows. First, we need to theta-join the two relations

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

using the condition that `presC#` from `Studio` and `cert#` from `MovieExec` are equal. That join combines pairs of tuples consisting of a studio and an executive, such that the executive is the president of the studio. If we select from this relation those tuples where the net worth is less than ten million, we have a set that, according to our constraint, must be empty. Thus, we may express the constraint as:

$$\sigma_{netWorth<10000000}(\texttt{Studio} \bowtie_{presC\#=cert\#} \texttt{MovieExec}) = \emptyset$$

An alternative way to express the same constraint is to compare the set of certificates that represent studio presidents with the set of certificates that represent executives with a net worth of at least $10,000,000; the former must be a subset of the latter. The containment

$$\pi_{presC\#}(\texttt{Studio}) \subseteq \pi_{cert\#}\Big(\sigma_{netWorth\geq 10000000}(\texttt{MovieExec})\Big)$$

expresses the above idea. □

5.5 Exercises for Section 5

Exercise 5.1: Express the following constraints about the relations of Exercise 3.1, reproduced here:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 4.1, indicate any violations to your constraints.

a) A PC with a processor speed less than 2.00 must not sell for more than $500.

b) A laptop with a screen size less than 15.4 inches must have at least a 100 gigabyte hard disk or sell for less than $1000.

! c) No manufacturer of PC's may also make laptops.

!! d) A manufacturer of a PC must also make a laptop with at least as great a processor speed.

! e) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.

Exercise 5.2: Express the following constraints in relational algebra. The constraints are based on the relations of Exercise 3.2:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 4.3, indicate any violations to your constraints.

a) No class of ships may have guns with larger than 16-inch bore.

b) If a class of ships has more than 9 guns, then their bore must be no larger than 14 inches.

! c) No class may have more than 2 ships.

! d) No country may have both battleships and battlecruisers.

!! e) No ship with more than 9 guns may be in a battle with a ship having fewer than 9 guns that was sunk.

! **Exercise 5.3:** Suppose R and S are two relations. Let C be the referential integrity constraint that says: whenever R has a tuple with some values $v_1, v_2, \ldots, v_n$ in particular attributes $A_1, A_2, \ldots, A_n$, there must be a tuple of S that has the same values $v_1, v_2, \ldots, v_n$ in particular attributes $B_1, B_2, \ldots, B_n$. Show how to express constraint C in relational algebra.

! **Exercise 5.4:** Another algebraic way to express a constraint is $E_1 = E_2$, where both E_1 and E_2 are relational-algebra expressions. Can this form of constraint express more than the two forms we discussed in this section?

6 Summary

✦ *Data Models*: A data model is a notation for describing the structure of the data in a database, along with the constraints on that data. The data model also normally provides a notation for describing operations on that data: queries and data modifications.

- *Relational Model*: Relations are tables representing information. Columns are headed by attributes; each attribute has an associated domain, or data type. Rows are called tuples, and a tuple has one component for each attribute of the relation.
- *Schemas*: A relation name, together with the attributes of that relation and their types, form the relation schema. A collection of relation schemas forms a database schema. Particular data for a relation or collection of relations is called an instance of that relation schema or database schema.
- *Keys*: An important type of constraint on relations is the assertion that an attribute or set of attributes forms a key for the relation. No two tuples of a relation can agree on all attributes of the key, although they can agree on some of the key attributes.
- *Semistructured Data Model*: In this model, data is organized in a tree or graph structure. XML is an important example of a semistructured data model.
- *SQL*: The language SQL is the principal query language for relational database systems. The current standard is called SQL-99. Commercial systems generally vary from this standard but adhere to much of it.
- *Data Definition*: SQL has statements to declare elements of a database schema. The `CREATE TABLE` statement allows us to declare the schema for stored relations (called tables), specifying the attributes, their types, default values, and keys.
- *Altering Schemas*: We can change parts of the database schema with an `ALTER` statement. These changes include adding and removing attributes from relation schemas and changing the default value associated with an attribute. We may also use a `DROP` statement to completely eliminate relations or other schema elements.
- *Relational Algebra*: This algebra underlies most query languages for the relational model. Its principal operators are union, intersection, difference, selection, projection, Cartesian product, natural join, theta-join, and renaming.
- *Selection and Projection*: The selection operator produces a result consisting of all tuples of the argument relation that satisfy the selection condition. Projection removes undesired columns from the argument relation to produce the result.
- *Joins*: We join two relations by comparing tuples, one from each relation. In a natural join, we splice together those pairs of tuples that agree on all attributes common to the two relations. In a theta-join, pairs of tuples are concatenated if they meet a selection condition associated with the theta-join.

- ✦ *Constraints in Relational Algebra*: Many common kinds of constraints can be expressed as the containment of one relational algebra expression in another, or as the equality of a relational algebra expression to the empty set.

7 References

The classic paper by Codd on the relational model is [1]. This paper introduces relational algebra, as well. The use of relational algebra to describe constraints is from [2].

The semistructured data model is from [3]. XML is a standard developed by the World-Wide-Web Consortium. The home page for information about XML is [4].

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

2. J.-M. Nicolas, "Logic for improving integrity checking in relational databases," *Acta Informatica* **18**:3, pp. 227–253, 1982.

3. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *IEEE Intl. Conf. on Data Engineering*, pp. 251–260, March 1995.

4. World-Wide-Web Consortium, `http://www.w3.org/XML/`

Design Theory for Relational Databases

There are many ways we could go about designing a relational database schema for an application. There are numerous high-level notations for describing the structure of data and ways in which these high-level designs can be converted into relations. We can also examine the requirements for a database and define relations directly, without going through a high-level intermediate stage. Whatever approach we use, it is common for an initial relational schema to have room for improvement, especially by eliminating redundancy. Often, the problems with a schema involve trying to combine too much into one relation.

Fortunately, there is a well developed theory for relational databases: "dependencies," their implications for what makes a good relational database schema, and what we can do about a schema if it has flaws. In this chapter, we first identify the problems that are caused in some relation schemas by the presence of certain dependencies; these problems are referred to as "anomalies."

Our discussion starts with "functional dependencies," a generalization of the idea of a key for a relation. We then use the notion of functional dependencies to define normal forms for relation schemas. The impact of this theory, called "normalization," is that we decompose relations into two or more relations when that will remove anomalies. Next, we introduce "multivalued dependencies," which intuitively represent a condition where one or more attributes of a relation are independent from one or more other attributes. These dependencies also lead to normal forms and decomposition of relations to eliminate redundancy.

1 Functional Dependencies

There is a design theory for relations that lets us examine a design carefully and make improvements based on a few simple principles. The theory begins by

having us state the constraints that apply to the relation. The most common constraint is the "functional dependency," a statement of a type that generalizes the idea of a key for a relation. Later in this chapter, we shall see how this theory gives us simple tools to improve our designs by the process of "decomposition" of relations: the replacement of one relation by several, whose sets of attributes together include all the attributes of the original.

1.1 Definition of Functional Dependency

A *functional dependency* (FD) on a relation R is a statement of the form "If two tuples of R agree on all of the attributes $A_1, A_2, \ldots, A_n$ (i.e., the tuples have the same values in their respective components for each of these attributes), then they must also agree on all of another list of attributes $B_1, B_2, \ldots, B_m$. We write this FD formally as $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$ and say that

$$\text{"}A_1, A_2, \ldots, A_n \text{ functionally determine } B_1, B_2, \ldots, B_m\text{"}$$

Figure 1 suggests what this FD tells us about any two tuples t and u in the relation R. However, the A's and B's can be anywhere; it is not necessary for the A's and B's to appear consecutively or for the A's to precede the B's.

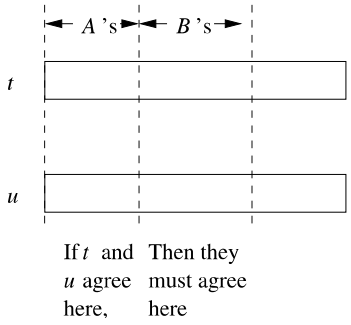

Figure 1: The effect of a functional dependency on two tuples

If we can be sure every instance of a relation R will be one in which a given FD is true, then we say that R *satisfies* the FD. It is important to remember that when we say that R satisfies an FD f, we are asserting a constraint on R, not just saying something about one particular instance of R.

It is common for the right side of an FD to be a single attribute. In fact, we shall see that the one functional dependency $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$ is equivalent to the set of FD's:

$$\begin{aligned} &A_1A_2 \cdots A_n \to B_1 \\ &A_1A_2 \cdots A_n \to B_2 \\ &\cdots \\ &A_1A_2 \cdots A_n \to B_m \end{aligned}$$

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

Example 1: Let us consider the relation

`Movies1(title, year, length, genre, studioName, starName)`

an instance of which is shown in Fig. 2. While related to our running `Movies` relation, it has additional attributes, which is why we call it "`Movies1`" instead of "`Movies`." Notice that this relation tries to "do too much." It holds information that in our running database schema was attributed to three different relations: `Movies`, `Studio`, and `StarsIn`. As we shall see, the schema for `Movies1` is not a good design. But to see what is wrong with the design, we must first determine the functional dependencies that hold for the relation. We claim that the following FD holds:

`title year` $\rightarrow$ `length genre studioName`

Informally, this FD says that if two tuples have the same value in their `title` components, and they also have the same value in their `year` components, then these two tuples must also have the same values in their `length` components, the same values in their `genre` components, and the same values in their `studioName` components. This assertion makes sense, since we believe that it is not possible for there to be two movies released in the same year with the same title (although there could be movies of the same title released in different years). Thus, we expect that given a title and year, there is a unique movie. Therefore, there is a unique length for the movie, a unique genre, and a unique studio.

On the other hand, we observe that the statement

`title year` $\rightarrow$ `starName`

is false; it is not a functional dependency. Given a movie, it is entirely possible that there is more than one star for the movie listed in our database. Notice that even had we been lazy and only listed one star for *Star Wars* and one star for *Wayne's World* (just as we only listed one of the many stars for *Gone With the Wind*), this FD would not suddenly become true for the relation `Movies1`.

The reason is that the FD says something about all possible instances of the relation, not about one of its instances. The fact that we *could* have an instance with multiple stars for a movie rules out the possibility that title and year functionally determine starName. □

1.2 Keys of Relations

We say a set of one or more attributes $\{A_1, A_2, \ldots, A_n\}$ is a *key* for a relation R if:

1. Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of R to agree on all of $A_1, A_2, \ldots, A_n$.
2. No proper subset of $\{A_1, A_2, \ldots, A_n\}$ functionally determines all other attributes of R; i.e., a key must be *minimal*.

When a key consists of a single attribute A, we often say that A (rather than $\{A\}$) is a key.

Example 2: Attributes {`title`, `year`, `starName`} form a key for the relation `Movies1` of Fig. 2. First, we must show that they functionally determine all the other attributes. That is, suppose two tuples agree on these three attributes: `title`, `year`, and `starName`. Because they agree on `title` and `year`, they must agree on the other attributes — `length`, `genre`, and `studioName` — as we discussed in Example 1. Thus, two different tuples cannot agree on all of `title`, `year`, and `starName`; they would in fact be the same tuple.

Now, we must argue that no proper subset of {`title`, `year`, `starName`} functionally determines all other attributes. To see why, begin by observing that `title` and `year` do not determine `starName`, because many movies have more than one star. Thus, {`title`, `year`} is not a key.

{`year`, `starName`} is not a key because we could have a star in two movies in the same year; therefore

$$\texttt{year starName} \rightarrow \texttt{title}$$

is not an FD. Also, we claim that {`title`, `starName`} is not a key, because two movies with the same title, made in different years, occasionally have a star in common.[1] □

Sometimes a relation has more than one key. If so, it is common to designate one of the keys as the *primary key*. In commercial database systems, the choice of primary key can influence some implementation issues such as how the relation is stored on disk. However, the theory of FD's gives no special role to "primary keys."

[1] It's an interesting challenge to discover stars that appeared in two versions of the same movie.

What Is "Functional" About Functional Dependencies?

$A_1A_2\cdots A_n \to B$ is called a "functional" dependency because in principle there is a function that takes a list of values, one for each of attributes $A_1, A_2, \ldots, A_n$ and produces a unique value (or no value at all) for B. For instance, in the `Movies1` relation, we can imagine a function that takes a string like `"Star Wars"` and an integer like 1977 and produces the unique value of `length`, namely 124, that appears in the relation `Movies1`. However, this function is not the usual sort of function that we meet in mathematics, because there is no way to compute it from first principles. That is, we cannot perform some operations on strings like `"Star Wars"` and integers like 1977 and come up with the correct length. Rather, the function is only computed by lookup in the relation. We look for a tuple with the given `title` and `year` values and see what value that tuple has for `length`.

1.3 Superkeys

A set of attributes that contains a key is called a *superkey*, short for "superset of a key." Thus, every key is a superkey. However, some superkeys are not (minimal) keys. Note that every superkey satisfies the first condition of a key: it functionally determines all other attributes of the relation. However, a superkey need not satisfy the second condition: minimality.

Example 3: In the relation of Example 2, there are many superkeys. Not only is the key

{`title`, `year`, `starName`}

a superkey, but any superset of this set of attributes, such as

{`title`, `year`, `starName`, `length`, `studioName`}

is a superkey. □

1.4 Exercises for Section 1

Exercise 1.1: Consider a relation about people in the United States, including their name, Social Security number, street address, city, state, ZIP code, area code, and phone number (7 digits). What FD's would you expect to hold? What are the keys for the relation? To answer this question, you need to know something about the way these numbers are assigned. For instance, can an area

Other Key Terminology

In some books and articles one finds different terminology regarding keys. One can find the term "key" used the way we have used the term "superkey," that is, a set of attributes that functionally determine all the attributes, with no requirement of minimality. These sources typically use the term "candidate key" for a key that is minimal — that is, a "key" in the sense we use the term.

code straddle two states? Can a ZIP code straddle two area codes? Can two people have the same Social Security number? Can they have the same address or phone number?

Exercise 1.2: Consider a relation representing the present position of molecules in a closed container. The attributes are an ID for the molecule, the x, y, and z coordinates of the molecule, and its velocity in the x, y, and z dimensions. What FD's would you expect to hold? What are the keys?

!! Exercise 1.3: Suppose R is a relation with attributes $A_1, A_2, \ldots, A_n$. As a function of n, tell how many superkeys R has, if:

a) The only key is A_1.

b) The only keys are A_1 and A_2.

c) The only keys are $\{A_1, A_2\}$ and $\{A_3, A_4\}$.

d) The only keys are $\{A_1, A_2\}$ and $\{A_1, A_3\}$.

2 Rules About Functional Dependencies

In this section, we shall learn how to *reason* about FD's. That is, suppose we are told of a set of FD's that a relation satisfies. Often, we can deduce that the relation must satisfy certain other FD's. This ability to discover additional FD's is essential when we discuss the design of good relation schemas in Section 3.

2.1 Reasoning About Functional Dependencies

Let us begin with a motivating example that will show us how we can infer a functional dependency from other given FD's.

Example 4: If we are told that a relation $R(A, B, C)$ satisfies the FD's $A \to B$ and $B \to C$, then we can deduce that R also satisfies the FD $A \to C$. How does that reasoning go? To prove that $A \to C$, we must consider two tuples of R that agree on A and prove they also agree on C.

Let the tuples agreeing on attribute A be (a, b_1, c_1) and (a, b_2, c_2). Since R satisfies $A \to B$, and these tuples agree on A, they must also agree on B. That is, $b_1 = b_2$, and the tuples are really (a, b, c_1) and (a, b, c_2), where b is both b_1 and b_2. Similarly, since R satisfies $B \to C$, and the tuples agree on B, they agree on C. Thus, $c_1 = c_2$; i.e., the tuples *do* agree on C. We have proved that any two tuples of R that agree on A also agree on C, and that is the FD $A \to C$. □

FD's often can be presented in several different ways, without changing the set of legal instances of the relation. We say:

- Two sets of FD's S and T are *equivalent* if the set of relation instances satisfying S is exactly the same as the set of relation instances satisfying T.
- More generally, a set of FD's S *follows* from a set of FD's T if every relation instance that satisfies all the FD's in T also satisfies all the FD's in S.

Note then that two sets of FD's S and T are equivalent if and only if S follows from T, and T follows from S.

In this section we shall see several useful rules about FD's. In general, these rules let us replace one set of FD's by an equivalent set, or to add to a set of FD's others that follow from the original set. An example is the *transitive rule* that lets us follow chains of FD's, as in Example 4. We shall also give an algorithm for answering the general question of whether one FD follows from one or more other FD's.

2.2 The Splitting/Combining Rule

Recall that in Section 1.1 we commented that the FD:

$$A_1 A_2 \cdots A_n \to B_1 B_2 \cdots B_m$$

was equivalent to the set of FD's:

$$A_1 A_2 \cdots A_n \to B_1, \quad A_1 A_2 \cdots A_n \to B_2, \ldots, A_1 A_2 \cdots A_n \to B_m$$

That is, we may split attributes on the right side so that only one attribute appears on the right of each FD. Likewise, we can replace a collection of FD's having a common left side by a single FD with the same left side and all the right sides combined into one set of attributes. In either event, the new set of FD's is equivalent to the old. The equivalence noted above can be used in two ways.

- We can replace an FD $A_1 A_2 \cdots A_n \to B_1 B_2 \cdots B_m$ by a set of FD's $A_1 A_2 \cdots A_n \to B_i$ for $i = 1, 2, \ldots, m$. This transformation we call the *splitting rule*.

- We can replace a set of FD's $A_1A_2\cdots A_n \to B_i$ for $i = 1, 2, \ldots, m$ by the single FD $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$. We call this transformation the *combining rule*.

Example 5: In Example 1 the set of FD's:

```
title year → length
title year → genre
title year → studioName
```

is equivalent to the single FD:

```
title year → length genre studioName
```

that we asserted there. □

The reason the splitting and combining rules are true should be obvious. Suppose we have two tuples that agree in $A_1, A_2, \ldots, A_n$. As a single FD, we would assert "then the tuples must agree in all of $B_1, B_2, \ldots, B_m$." As individual FD's, we assert "then the tuples agree in B_1, and they agree in B_2, and,..., and they agree in B_m." These two conclusions say exactly the same thing.

One might imagine that splitting could be applied to the left sides of FD's as well as to right sides. However, there is no splitting rule for left sides, as the following example shows.

Example 6: Consider one of the FD's such as:

```
title year → length
```

for the relation `Movies1` in Example 1. If we try to split the left side into

```
title → length
year → length
```

then we get two false FD's. That is, `title` does not functionally determine `length`, since there can be several movies with the same title (e.g., *King Kong*) but of different lengths. Similarly, `year` does not functionally determine `length`, because there are certainly movies of different lengths made in any one year. □

2.3 Trivial Functional Dependencies

A constraint of any kind on a relation is said to be *trivial* if it holds for every instance of the relation, regardless of what other constraints are assumed. When the constraints are FD's, it is easy to tell whether an FD is trivial. They are the FD's $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ such that

$$\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$$

That is, a trivial FD has a right side that is a subset of its left side. For example,

$$\texttt{title year} \rightarrow \texttt{title}$$

is a trivial FD, as is

$$\texttt{title} \rightarrow \texttt{title}$$

Every trivial FD holds in every relation, since it says that "two tuples that agree in all of $A_1, A_2, \ldots, A_n$ agree in a subset of them." Thus, we may assume any trivial FD, without having to justify it on the basis of what FD's are asserted for the relation.

There is an intermediate situation in which some, but not all, of the attributes on the right side of an FD are also on the left. This FD is not trivial, but it can be simplifed by removing from the right side of an FD those attributes that appear on the left. That is:

- The FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ is equivalent to

  $$A_1A_2\cdots A_n \rightarrow C_1C_2\cdots C_k$$

  where the C's are all those B's that are not also A's.

We call this rule, illustrated in Fig. 3, the *trivial-dependency rule*.

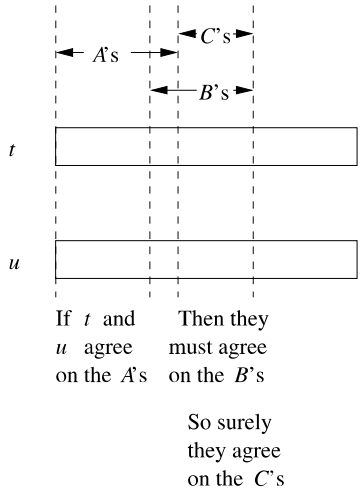

Figure 3: The trivial-dependency rule

2.4 Computing the Closure of Attributes

Before proceeding to other rules, we shall give a general principle from which all true rules follow. Suppose $\{A_1, A_2, \ldots, A_n\}$ is a set of attributes and S

is a set of FD's. The *closure* of $\{A_1, A_2, \ldots, A_n\}$ under the FD's in S is the set of attributes B such that every relation that satisfies all the FD's in set S also satisfies $A_1A_2\cdots A_n \rightarrow B$. That is, $A_1A_2\cdots A_n \rightarrow B$ follows from the FD's of S. We denote the closure of a set of attributes $A_1A_2\cdots A_n$ by $\{A_1, A_2, \ldots, A_n\}^+$. Note that $A_1, A_2, \ldots, A_n$ are always in $\{A_1, A_2, \ldots, A_n\}^+$ because the FD $A_1A_2\cdots A_n \rightarrow A_i$ is trivial when i is one of $1, 2, \ldots, n$.

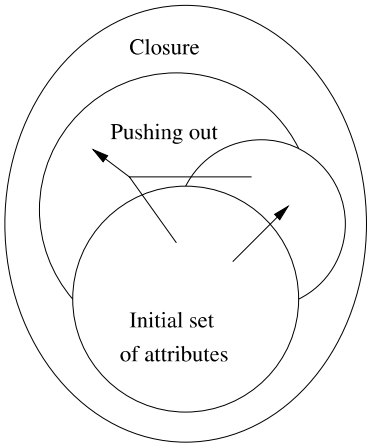

Figure 4: Computing the closure of a set of attributes

Figure 4 illustrates the closure process. Starting with the given set of attributes, we repeatedly expand the set by adding the right sides of FD's as soon as we have included their left sides. Eventually, we cannot expand the set any further, and the resulting set is the closure. More precisely:

Algorithm 7: Closure of a Set of Attributes.

INPUT: A set of attributes $\{A_1, A_2, \ldots, A_n\}$ and a set of FD's S.

OUTPUT: The closure $\{A_1, A_2, \ldots, A_n\}^+$.

1. If necessary, split the FD's of S, so each FD in S has a single attribute on the right.

2. Let X be a set of attributes that eventually will become the closure. Initialize X to be $\{A_1, A_2, \ldots, A_n\}$.

3. Repeatedly search for some FD

$$B_1B_2\cdots B_m \rightarrow C$$

such that all of $B_1, B_2, \ldots, B_m$ are in the set of attributes X, but C is not. Add C to the set X and repeat the search. Since X can only grow, and the number of attributes of any relation schema must be finite, eventually nothing more can be added to X, and this step ends.

4. The set X, after no more attributes can be added to it, is the correct value of $\{A_1, A_2, \ldots, A_n\}^+$.

□

Example 8: Let us consider a relation with attributes A, B, C, D, E, and F. Suppose that this relation has the FD's $AB \to C$, $BC \to AD$, $D \to E$, and $CF \to B$. What is the closure of $\{A, B\}$, that is, $\{A, B\}^+$?

First, split $BC \to AD$ into $BC \to A$ and $BC \to D$. Then, start with $X = \{A, B\}$. First, notice that both attributes on the left side of FD $AB \to C$ are in X, so we may add the attribute C, which is on the right side of that FD. Thus, after one iteration of Step 3, X becomes $\{A, B, C\}$.

Next, we see that the left sides of $BC \to A$ and $BC \to D$ are now contained in X, so we may add to X the attributes A and D. A is already there, but D is not, so X next becomes $\{A, B, C, D\}$. At this point, we may use the FD $D \to E$ to add E to X, which is now $\{A, B, C, D, E\}$. No more changes to X are possible. In particular, the FD $CF \to B$ can not be used, because its left side never becomes contained in X. Thus, $\{A, B\}^+ = \{A, B, C, D, E\}$. □

By computing the closure of any set of attributes, we can test whether any given FD $A_1A_2 \cdots A_n \to B$ follows from a set of FD's S. First compute $\{A_1, A_2, \ldots, A_n\}^+$ using the set of FD's S. If B is in $\{A_1, A_2, \ldots, A_n\}^+$, then $A_1A_2 \cdots A_n \to B$ does follow from S, and if B is not in $\{A_1, A_2, \ldots, A_n\}^+$, then this FD does not follow from S. More generally, $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$ follows from set of FD's S if and only if all of $B_1, B_2, \ldots, B_m$ are in

$$\{A_1, A_2, \ldots, A_n\}^+$$

Example 9: Consider the relation and FD's of Example 8. Suppose we wish to test whether $AB \to D$ follows from these FD's. We compute $\{A, B\}^+$, which is $\{A, B, C, D, E\}$, as we saw in that example. Since D is a member of the closure, we conclude that $AB \to D$ does follow.

On the other hand, consider the FD $D \to A$. To test whether this FD follows from the given FD's, first compute $\{D\}^+$. To do so, we start with $X = \{D\}$. We can use the FD $D \to E$ to add E to the set X. However, then we are stuck. We cannot find any other FD whose left side is contained in $X = \{D, E\}$, so $\{D\}^+ = \{D, E\}$. Since A is not a member of $\{D, E\}$, we conclude that $D \to A$ does not follow. □

2.5 Why the Closure Algorithm Works

In this section, we shall show why Algorithm 7 correctly decides whether or not an FD $A_1A_2 \cdots A_n \to B$ follows from a given set of FD's S. There are two parts to the proof:

1. We must prove that Algorithm 7 does not claim too much. That is, we must show that if $A_1A_2 \cdots A_n \to B$ is asserted by the closure test (i.e.,

B is in $\{A_1, A_2, \ldots, A_n\}^+$), then $A_1A_2 \cdots A_n \to B$ holds in any relation that satisfies all the FD's in S.

2. We must prove that Algorithm 7 does not fail to discover a FD that truly follows from the set of FD's S.

Why the Closure Algorithm Claims only True FD's

We can prove by induction on the number of times that we apply the growing operation of Step 3 that for every attribute D in X, the FD $A_1A_2 \cdots A_n \to D$ holds. That is, every relation R satisfying all of the FD's in S also satisfies $A_1A_2 \cdots A_n \to D$.

BASIS: The basis case is when there are zero steps. Then D must be one of $A_1, A_2, \ldots, A_n$, and surely $A_1A_2 \cdots A_n \to D$ holds in any relation, because it is a trivial FD.

INDUCTION: For the induction, suppose D was added when we used the FD $B_1B_2 \cdots B_m \to D$ of S. We know by the inductive hypothesis that R satisfies $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$. Now, suppose two tuples of R agree on all of $A_1, A_2, \ldots, A_n$. Then since R satisfies $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$, the two tuples must agree on all of $B_1, B_2, \ldots, B_m$. Since R satisfies $B_1B_2 \cdots B_m \to D$, we also know these two tuples agree on D. Thus, R satisfies $A_1A_2 \cdots A_n \to D$.

Why the Closure Algorithm Discovers All True FD's

Suppose $A_1A_2 \cdots A_n \to B$ were an FD that Algorithm 7 says does not follow from set S. That is, the closure of $\{A_1, A_2, \ldots, A_n\}$ using set of FD's S does not include B. We must show that FD $A_1A_2 \cdots A_n \to B$ really doesn't follow from S. That is, we must show that there is at least one relation instance that satisfies all the FD's in S, and yet does not satisfy $A_1A_2 \cdots A_n \to B$.

This instance I is actually quite simple to construct; it is shown in Fig. 5. I has only two tuples: t and s. The two tuples agree in all the attributes of $\{A_1, A_2, \ldots, A_n\}^+$, and they disagree in all the other attributes. We must show first that I satisfies all the FD's of S, and then that it does not satisfy $A_1A_2 \cdots A_n \to B$.

	$\{A_1, A_2, \ldots, A_n\}^+$	Other Attributes
t:	1 1 1 $\cdots$ 1 1	0 0 0 $\cdots$ 0 0
s:	1 1 1 $\cdots$ 1 1	1 1 1 $\cdots$ 1 1

Figure 5: An instance I satisfying S but not $A_1A_2 \cdots A_n \to B$

Suppose there were some FD $C_1C_2 \cdots C_k \to D$ in set S (after splitting right sides) that instance I does not satisfy. Since I has only two tuples, t and s, those must be the two tuples that violate $C_1C_2 \cdots C_k \to D$. That is, t and s agree in all the attributes of $\{C_1, C_2, \ldots, C_k\}$, yet disagree on D. If we

examine Fig. 5 we see that all of $C_1, C_2, \ldots, C_k$ must be among the attributes of $\{A_1, A_2, \ldots, A_n\}^+$, because those are the only attributes on which t and s agree. Likewise, D must be among the other attributes, because only on those attributes do t and s disagree.

But then we did not compute the closure correctly. $C_1C_2\cdots C_k \to D$ should have been applied when X was $\{A_1, A_2, \ldots, A_n\}$ to add D to X. We conclude that $C_1C_2\cdots C_k \to D$ cannot exist; i.e., instance I satisfies S.

Second, we must show that I does not satisfy $A_1A_2\cdots A_n \to B$. However, this part is easy. Surely, $A_1, A_2, \ldots, A_n$ are among the attributes on which t and s agree. Also, we know that B is not in $\{A_1, A_2, \ldots, A_n\}^+$, so B is one of the attributes on which t and s disagree. Thus, I does not satisfy $A_1A_2\cdots A_n \to B$. We conclude that Algorithm 7 asserts neither too few nor too many FD's; it asserts exactly those FD's that do follow from S.

2.6 The Transitive Rule

The transitive rule lets us cascade two FD's, and generalizes the observation of Example 4.

- If $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ and $B_1B_2\cdots B_m \to C_1C_2\cdots C_k$ hold in relation R, then $A_1A_2\cdots A_n \to C_1C_2\cdots C_k$ also holds in R.

If some of the C's are among the A's, we may eliminate them from the right side by the trivial-dependencies rule.

To see why the transitive rule holds, apply the test of Section 2.4. To test whether $A_1A_2\cdots A_n \to C_1C_2\cdots C_k$ holds, we need to compute the closure $\{A_1, A_2, \ldots, A_n\}^+$ with respect to the two given FD's.

The FD $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ tells us that all of $B_1, B_2, \ldots, B_m$ are in $\{A_1, A_2, \ldots, A_n\}^+$. Then, we can use the FD $B_1B_2\cdots B_m \to C_1C_2\cdots C_k$ to add $C_1, C_2, \ldots, C_k$ to $\{A_1, A_2, \ldots, A_n\}^+$. Since all the C's are in

$$\{A_1, A_2, \ldots, A_n\}^+$$

we conclude that $A_1A_2\cdots A_n \to C_1C_2\cdots C_k$ holds for any relation that satisfies both $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ and $B_1B_2\cdots B_m \to C_1C_2\cdots C_k$.

Example 10: Here is another version of the `Movies` relation that includes both the studio of the movie and some information about that studio.

title	*year*	*length*	*genre*	*studioName*	*studioAddr*
`Star Wars`	1977	124	`sciFi`	`Fox`	`Hollywood`
`Eight Below`	2005	120	`drama`	`Disney`	`Buena Vista`
`Wayne's World`	1992	95	`comedy`	`Paramount`	`Hollywood`

Two of the FD's that we might reasonably claim to hold are:

`title year` → `studioName`
`studioName` → `studioAddr`

Closures and Keys

Notice that $\{A_1, A_2, \ldots, A_n\}^+$ is the set of all attributes of a relation if and only if $A_1, A_2, \ldots, A_n$ is a superkey for the relation. For only then does $A_1, A_2, \ldots, A_n$ functionally determine all the other attributes. We can test if $A_1, A_2, \ldots, A_n$ is a key for a relation by checking first that $\{A_1, A_2, \ldots, A_n\}^+$ is all attributes, and then checking that, for no set X formed by removing one attribute from $\{A_1, A_2, \ldots, A_n\}$, is X^+ the set of all attributes.

The first is justified because there can be only one movie with a given title and year, and there is only one studio that owns a given movie. The second is justified because studios have unique addresses.

The transitive rule allows us to combine the two FD's above to get a new FD:

$$\texttt{title year} \rightarrow \texttt{studioAddr}$$

This FD says that a title and year (i.e., a movie) determines an address — the address of the studio owning the movie. □

2.7 Closing Sets of Functional Dependencies

Sometimes we have a choice of which FD's we use to represent the full set of FD's for a relation. If we are given a set of FD's S (such as the FD's that hold in a given relation), then any set of FD's equivalent to S is said to be a *basis* for S. To avoid some of the explosion of possible bases, we shall limit ourselves to considering only bases whose FD's have singleton right sides. If we have any basis, we can apply the splitting rule to make the right sides be singletons. A *minimal basis* for a relation is a basis B that satisfies three conditions:

1. All the FD's in B have singleton right sides.
2. If any FD is removed from B, the result is no longer a basis.
3. If for any FD in B we remove one or more attributes from the left side of F, the result is no longer a basis.

Notice that no trivial FD can be in a minimal basis, because it could be removed by rule (2).

Example 11: Consider a relation $R(A, B, C)$ such that each attribute functionally determines the other two attributes. The full set of derived FD's thus includes six FD's with one attribute on the left and one on the right; $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$, $C \rightarrow A$, and $C \rightarrow B$. It also includes the three

A Complete Set of Inference Rules

If we want to know whether one FD follows from some given FD's, the closure computation of Section 2.4 will always serve. However, it is interesting to know that there is a set of rules, called *Armstrong's axioms*, from which it is possible to derive any FD that follows from a given set. These axioms are:

1. *Reflexivity.* If $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$, then $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$. These are what we have called trivial FD's.

2. *Augmentation.* If $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$, then

   $$A_1A_2\cdots A_nC_1C_2\cdots C_k \to B_1B_2\cdots B_mC_1C_2\cdots C_k$$

   for any set of attributes $C_1, C_2, \ldots, C_k$. Since some of the C's may also be A's or B's or both, we should eliminate from the left side duplicate attributes and do the same for the right side.

3. *Transitivity.* If

   $$A_1A_2\cdots A_n \to B_1B_2\cdots B_m \text{ and } B_1B_2\cdots B_m \to C_1C_2\cdots C_k$$

   then $A_1A_2\cdots A_n \to C_1C_2\cdots C_k$.

nontrivial FD's with two attributes on the left: $AB \to C$, $AC \to B$, and $BC \to A$. There are also FD's with more than one attribute on the right, such as $A \to BC$, and trivial FD's such as $A \to A$.

Relation R and its FD's have several minimal bases. One is

$$\{A \to B,\ B \to A,\ B \to C,\ C \to B\}$$

Another is $\{A \to B,\ B \to C,\ C \to A\}$. There are several other minimal bases for R, and we leave their discovery as an exercise. □

2.8 Projecting Functional Dependencies

When we study design of relation schemas, we shall also have need to answer the following question about FD's. Suppose we have a relation R with set of FD's S, and we project R by computing $R_1 = \pi_L(R)$, for some list of attributes R. What FD's hold in R_1?

The answer is obtained in principle by computing the *projection of functional dependencies* S, which is all FD's that:

a) Follow from S, and

b) Involve only attributes of R_1.

Since there may be a large number of such FD's, and many of them may be redundant (i.e., they follow from other such FD's), we are free to simplify that set of FD's if we wish. However, in general, the calculation of the FD's for R_1 is exponential in the number of attributes of R_1. The simple algorithm is summarized below.

Algorithm 12: Projecting a Set of Functional Dependencies.

INPUT: A relation R and a second relation R_1 computed by the projection $R_1 = \pi_L(R)$. Also, a set of FD's S that hold in R.

OUTPUT: The set of FD's that hold in R_1.

METHOD:

1. Let T be the eventual output set of FD's. Initially, T is empty.
2. For each set of attributes X that is a subset of the attributes of R_1, compute X^+. This computation is performed with respect to the set of FD's S, and may involve attributes that are in the schema of R but not R_1. Add to T all nontrivial FD's $X \to A$ such that A is both in X^+ and an attribute of R_1.
3. Now, T is a basis for the FD's that hold in R_1, but may not be a minimal basis. We may construct a minimal basis by modifying T as follows:
   (a) If there is an FD F in T that follows from the other FD's in T, remove F from T.
   (b) Let $Y \to B$ be an FD in T, with at least two attributes in Y, and let Z be Y with one of its attributes removed. If $Z \to B$ follows from the FD's in T (including $Y \to B$), then replace $Y \to B$ by $Z \to B$.
   (c) Repeat the above steps in all possible ways until no more changes to T can be made.

□

Example 13: Suppose $R(A, B, C, D)$ has FD's $A \to B$, $B \to C$, and $C \to D$. Suppose also that we wish to project out the attribute B, leaving a relation $R_1(A, C, D)$. In principle, to find the FD's for R_1, we need to take the closure of all eight subsets of $\{A, C, D\}$, using the full set of FD's, including those involving B. However, there are some obvious simplifications we can make.

- Closing the empty set and the set of all attributes cannot yield a nontrivial FD.

- If we already know that the closure of some set X is all attributes, then we cannot discover any new FD's by closing supersets of X.

Thus, we may start with the closures of the singleton sets, and then move on to the doubleton sets if necessary. For each closure of a set X, we add the FD $X \to E$ for each attribute E that is in X^+ and in the schema of R_1, but not in X.

First, $\{A\}^+ = \{A, B, C, D\}$. Thus, $A \to C$ and $A \to D$ hold in R_1. Note that $A \to B$ is true in R, but makes no sense in R_1 because B is not an attribute of R_1.

Next, we consider $\{C\}^+ = \{C, D\}$, from which we get the additional FD $C \to D$ for R_1. Since $\{D\}^+ = \{D\}$, we can add no more FD's, and are done with the singletons.

Since $\{A\}^+$ includes all attributes of R_1, there is no point in considering any superset of $\{A\}$. The reason is that whatever FD we could discover, for instance $AC \to D$, follows from an FD with only A on the left side: $A \to D$ in this case. Thus, the only doubleton whose closure we need to take is $\{C, D\}^+ = \{C, D\}$. This observation allows us to add nothing. We are done with the closures, and the FD's we have discovered are $A \to C$, $A \to D$, and $C \to D$.

If we wish, we can observe that $A \to D$ follows from the other two by transitivity. Therefore a simpler, equivalent set of FD's for R_1 is $A \to C$ and $C \to D$. This set is, in fact, a minimal basis for the FD's of R_1. □

2.9 Exercises for Section 2

Exercise 2.1: Consider a relation with schema $R(A, B, C, D)$ and FD's $AB \to C$, $C \to D$, and $D \to A$.

a) What are all the nontrivial FD's that follow from the given FD's? You should restrict yourself to FD's with single attributes on the right side.

b) What are all the keys of R?

c) What are all the superkeys for R that are not keys?

Exercise 2.2: Repeat Exercise 2.1 for the following schemas and sets of FD's:

i) $S(A, B, C, D)$ with FD's $A \to B$, $B \to C$, and $B \to D$.

ii) $T(A, B, C, D)$ with FD's $AB \to C$, $BC \to D$, $CD \to A$, and $AD \to B$.

iii) $U(A, B, C, D)$ with FD's $A \to B$, $B \to C$, $C \to D$, and $D \to A$.

Exercise 2.3: Show that the following rules hold, by using the closure test of Section 2.4.

a) *Augmenting left sides.* If $A_1 A_2 \cdots A_n \to B$ is an FD, and C is another attribute, then $A_1 A_2 \cdots A_n C \to B$ follows.

b) *Full augmentation.* If $A_1A_2\cdots A_n \rightarrow B$ is an FD, and C is another attribute, then $A_1A_2\cdots A_nC \rightarrow BC$ follows. Note: from this rule, the "augmentation" rule mentioned in the box of Section 2.7 on "A Complete Set of Inference Rules" can easily be proved.

c) *Pseudotransitivity.* Suppose FD's $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ and

$$C_1C_2\cdots C_k \rightarrow D$$

hold, and the B's are each among the C's. Then

$$A_1A_2\cdots A_nE_1E_2\cdots E_j \rightarrow D$$

holds, where the E's are all those of the C's that are not found among the B's.

d) *Addition.* If FD's $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ and

$$C_1C_2\cdots C_k \rightarrow D_1D_2\cdots D_j$$

hold, then FD $A_1A_2\cdots A_nC_1C_2\cdots C_k \rightarrow B_1B_2\cdots B_mD_1D_2\cdots D_j$ also holds. In the above, we should remove one copy of any attribute that appears among both the A's and C's or among both the B's and D's.

! Exercise 2.4: Show that each of the following are *not* valid rules about FD's by giving example relations that satisfy the given FD's (following the "if") but not the FD that allegedly follows (after the "then").

a) If $A \rightarrow B$ then $B \rightarrow A$.

b) If $AB \rightarrow C$ and $A \rightarrow C$, then $B \rightarrow C$.

c) If $AB \rightarrow C$, then $A \rightarrow C$ or $B \rightarrow C$.

! Exercise 2.5: Show that if a relation has no attribute that is functionally determined by all the other attributes, then the relation has no nontrivial FD's at all.

! Exercise 2.6: Let X and Y be sets of attributes. Show that if $X \subseteq Y$, then $X^+ \subseteq Y^+$, where the closures are taken with respect to the same set of FD's.

! Exercise 2.7: Prove that $(X^+)^+ = X^+$.

!! Exercise 2.8: We say a set of attributes X is *closed* (with respect to a given set of FD's) if $X^+ = X$. Consider a relation with schema $R(A, B, C, D)$ and an unknown set of FD's. If we are told which sets of attributes are closed, we can discover the FD's. What are the FD's if:

a) All sets of the four attributes are closed.

b) The only closed sets are $\emptyset$ and $\{A, B, C, D\}$.

c) The closed sets are $\emptyset$, {A,B}, and $\{A, B, C, D\}$.

! Exercise 2.9 : Find all the minimal bases for the FD's and relation of Example 11.

! Exercise 2.10 : Suppose we have relation $R(A, B, C, D, E)$, with some set of FD's, and we wish to project those FD's onto relation $S(A, B, C)$. Give the FD's that hold in S if the FD's for R are:

a) $AB \to DE$, $C \to E$, $D \to C$, and $E \to A$.

b) $A \to D$, $BD \to E$, $AC \to E$, and $DE \to B$.

c) $AB \to D$, $AC \to E$, $BC \to D$, $D \to A$, and $E \to B$.

d) $A \to B$, $B \to C$, $C \to D$, $D \to E$, and $E \to A$.

In each case, it is sufficient to give a minimal basis for the full set of FD's of S.

!! Exercise 2.11 : Show that if an FD F follows from some given FD's, then we can prove F from the given FD's using Armstrong's axioms (defined in the box "A Complete Set of Inference Rules" in Section 2.7). *Hint*: Examine Algorithm 7 and show how each step of that algorithm can be mimicked by inferring some FD's by Armstrong's axioms.

3 Design of Relational Database Schemas

Careless selection of a relational database schema can lead to redundancy and related anomalies. For instance, consider the relation in Fig. 2, which we reproduce here as Fig. 6. Notice that the length and genre for *Star Wars* and *Wayne's World* are each repeated, once for each star of the movie. The repetition of this information is redundant. It also introduces the potential for several kinds of errors, as we shall see.

In this section, we shall tackle the problem of design of good relation schemas in the following stages:

1. We first explore in more detail the problems that arise when our schema is poorly designed.
2. Then, we introduce the idea of "decomposition," breaking a relation schema (set of attributes) into two smaller schemas.
3. Next, we introduce "Boyce-Codd normal form," or "BCNF," a condition on a relation schema that eliminates these problems.
4. These points are tied together when we explain how to assure the BCNF condition by decomposing relation schemas.

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 6: The relation `Movies1` exhibiting anomalies

3.1 Anomalies

Problems such as redundancy that occur when we try to cram too much into a single relation are called *anomalies*. The principal kinds of anomalies that we encounter are:

1. *Redundancy.* Information may be repeated unnecessarily in several tuples. Examples are the length and genre for movies in Fig. 6.

2. *Update Anomalies.* We may change information in one tuple but leave the same information unchanged in another. For example, if we found that *Star Wars* is really 125 minutes long, we might carelessly change the length in the first tuple of Fig. 6 but not in the second or third tuples. You might argue that one should never be so careless, but it is possible to redesign relation `Movies1` so that the risk of such mistakes does not exist.

3. *Deletion Anomalies.* If a set of values becomes empty, we may lose other information as a side effect. For example, should we delete Vivien Leigh from the set of stars of *Gone With the Wind*, then we have no more stars for that movie in the database. The last tuple for *Gone With the Wind* in the relation `Movies1` would disappear, and with it information that it is 231 minutes long and a drama.

3.2 Decomposing Relations

The accepted way to eliminate these anomalies is to *decompose* relations. Decomposition of R involves splitting the attributes of R to make the schemas of two new relations. After describing the decomposition process, we shall show how to pick a decomposition that eliminates anomalies.

Given a relation $R(A_1, A_2, \ldots, A_n)$, we may *decompose* R into two relations $S(B_1, B_2, \ldots, B_m)$ and $T(C_1, C_2, \ldots, C_k)$ such that:

1. $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$.

2. $S = \pi_{B_1, B_2, \ldots, B_m}(R)$.

3. $T = \pi_{C_1,C_2,\ldots,C_k}(R)$.

Example 14: Let us decompose the `Movies1` relation of Fig. 6. Our choice, whose merit will be seen in Section 3.3, is to use:

1. A relation called `Movies2`, whose schema is all the attributes except for `starName`.

2. A relation called `Movies3`, whose schema consists of the attributes `title`, `year`, and `starName`.

The projection of `Movies1` onto these two new schemas is shown in Fig. 7. □

title	*year*	*length*	*genre*	*studioName*
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	231	drama	MGM
Wayne's World	1992	95	comedy	Paramount

(b) The relation `Movies2`.

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

(b) The relation `Movies3`.

Figure 7: Projections of relation `Movies1`

Notice how this decomposition eliminates the anomalies we mentioned in Section 3.1. The redundancy has been eliminated; for example, the length of each film appears only once, in relation `Movies2`. The risk of an update anomaly is gone. For instance, since we only have to change the length of *Star Wars* in one tuple of `Movies2`, we cannot wind up with two different lengths for that movie.

Finally, the risk of a deletion anomaly is gone. If we delete all the stars for *Gone With the Wind*, say, that deletion makes the movie disappear from `Movies3`. But all the other information about the movie can still be found in `Movies2`.

It might appear that `Movies3` still has redundancy, since the title and year of a movie can appear several times. However, these two attributes form a key for movies, and there is no more succinct way to represent a movie. Moreover, `Movies3` does not offer an opportunity for an update anomaly. For instance, one might suppose that if we changed to 2008 the year in the Carrie Fisher tuple, but not the other two tuples for *Star Wars*, then there would be an update anomaly. However, there is nothing in our assumed FD's that prevents there being a different movie named *Star Wars* in 2008, and Carrie Fisher may star in that one as well. Thus, we do not want to prevent changing the year in one *Star Wars* tuple, nor is such a change necessarily incorrect.

3.3 Boyce-Codd Normal Form

The goal of decomposition is to replace a relation by several that do not exhibit anomalies. There is, it turns out, a simple condition under which the anomalies discussed above can be guaranteed not to exist. This condition is called *Boyce-Codd normal form*, or *BCNF*.

- A relation R is in BCNF if and only if: whenever there is a nontrivial FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ for R, it is the case that $\{A_1, A_2, \ldots, A_n\}$ is a superkey for R.

That is, the left side of every nontrivial FD must be a superkey. Recall that a superkey need not be minimal. Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.

Example 15: Relation `Movies1`, as in Fig. 6, is not in BCNF. To see why, we first need to determine what sets of attributes are keys. We argued in Example 2 why {`title`, `year`, `starName`} is a key. Thus, any set of attributes containing these three is a superkey. The same arguments we followed in Example 2 can be used to explain why no set of attributes that does not include all three of `title`, `year`, and `starName` could be a superkey. Thus, we assert that {`title`, `year`, `starName`} is the only key for `Movies1`.

However, consider the FD

`title year` $\rightarrow$ `length genre studioName`

which holds in `Movies1` according to our discussion in Example 2.

Unfortunately, the left side of the above FD is not a superkey. In particular, we know that `title` and `year` do not functionally determine the sixth attribute, `starName`. Thus, the existence of this FD violates the BCNF condition and tells us `Movies1` is not in BCNF. □

Example 16: On the other hand, `Movies2` of Fig. 7 is in BCNF. Since

`title year` $\rightarrow$ `length genre studioName`

holds in this relation, and we have argued that neither `title` nor `year` by itself functionally determines any of the other attributes, the only key for `Movies2` is {`title`, `year`}. Moreover, the only nontrivial FD's must have at least `title` and `year` on the left side, and therefore their left sides must be superkeys. Thus, `Movies2` is in BCNF. □

Example 17: We claim that any two-attribute relation is in BCNF. We need to examine the possible nontrivial FD's with a single attribute on the right. There are not too many cases to consider, so let us consider them in turn. In what follows, suppose that the attributes are A and B.

1. There are no nontrivial FD's. Then surely the BCNF condition must hold, because only a nontrivial FD can violate this condition. Incidentally, note that $\{A, B\}$ is the only key in this case.
2. $A \to B$ holds, but $B \to A$ does not hold. In this case, A is the only key, and each nontrivial FD contains A on the left (in fact the left can only be A). Thus there is no violation of the BCNF condition.
3. $B \to A$ holds, but $A \to B$ does not hold. This case is symmetric to case (2).
4. Both $A \to B$ and $B \to A$ hold. Then both A and B are keys. Surely any FD has at least one of these on the left, so there can be no BCNF violation.

It is worth noticing from case (4) above that there may be more than one key for a relation. Further, the BCNF condition only requires that *some* key be contained in the left side of any nontrivial FD, not that all keys are contained in the left side. Also observe that a relation with two attributes, each functionally determining the other, is not completely implausible. For example, a company may assign its employees unique employee ID's and also record their Social Security numbers. A relation with attributes `empID` and `ssNo` would have each attribute functionally determining the other. Put another way, each attribute is a key, since we don't expect to find two tuples that agree on either attribute. □

3.4 Decomposition into BCNF

By repeatedly choosing suitable decompositions, we can break any relation schema into a collection of subsets of its attributes with the following important properties:

1. These subsets are the schemas of relations in BCNF.
2. The data in the original relation is represented faithfully by the data in the relations that are the result of the decomposition, in a sense to be made precise in Section 4.1. Roughly, we need to be able to reconstruct the original relation instance exactly from the decomposed relation instances.

Example 17 suggests that perhaps all we have to do is break a relation schema into two-attribute subsets, and the result is surely in BCNF. However, such an arbitrary decomposition will not satisfy condition (2), as we shall see in Section 4.1. In fact, we must be more careful and use the violating FD's to guide our decomposition.

The decomposition strategy we shall follow is to look for a nontrivial FD $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ that violates BCNF; i.e., $\{A_1, A_2, \ldots, A_n\}$ is not a superkey. We shall add to the right side as many attributes as are functionally determined by $\{A_1, A_2, \ldots, A_n\}$. This step is not mandatory, but it often reduces the total amount of work done, and we shall include it in our algorithm. Figure 8 illustrates how the attributes are broken into two overlapping relation schemas. One is all the attributes involved in the violating FD, and the other is the left side of the FD plus all the attributes *not* involved in the FD, i.e., all the attributes except those B's that are not A's.

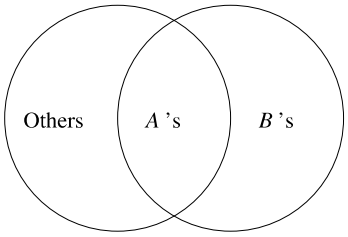

Figure 8: Relation schema decomposition based on a BCNF violation

Example 18: Consider our running example, the `Movies1` relation of Fig. 6. We saw in Example 15 that

`title year` $\to$ `length genre studioName`

is a BCNF violation. In this case, the right side already includes all the attributes functionally determined by `title` and `year`, so we shall use this BCNF violation to decompose `Movies1` into:

1. The schema {`title`, `year`, `length`, `genre`, `studioName`} consisting of all the attributes on either side of the FD.

2. The schema {`title`, `year`, `starName`} consisting of the left side of the FD plus all attributes of `Movies1` that do not appear in either side of the FD (only `starName`, in this case).

Notice that these schemas are the ones selected for relations `Movies2` and `Movies3` in Example 14. We observed in Example 16 that `Movies2` is in BCNF. `Movies3` is also in BCNF; it has no nontrivial FD's. □

In Example 18, one judicious application of the decomposition rule is enough to produce a collection of relations that are in BCNF. In general, that is not the case, as the next example shows.

Example 19: Consider a relation with schema

{title, year, studioName, president, presAddr}

That is, each tuple of this relation tells about a movie, its studio, the president of the studio, and the address of the president of the studio. Three FD's that we would assume in this relation are

title year → studioName
studioName → president
president → presAddr

By closing sets of these five attributes, we discover that {title, year} is the only key for this relation. Thus the last two FD's above violate BCNF. Suppose we choose to decompose starting with

studioName → president

First, we add to the right side of this functional dependency any other attributes in the closure of studioName. That closure includes presAddr, so our final choice of FD for the decomposition is:

studioName → president presAddr

The decomposition based on this FD yields the following two relation schemas.

{title, year, studioName}
{studioName, president, presAddr}

If we use Algorithm 12 to project FD's, we determine that the FD's for the first relation has a basis:

title year → studioName

while the second has:

studioName → president
president → presAddr

The sole key for the first relation is {title, year}, and it is therefore in BCNF. However, the second has {studioName} for its only key but also has the FD:

president → presAddr

which is a BCNF violation. Thus, we must decompose again, this time using the above FD. The resulting three relation schemas, all in BCNF, are:

{title, year, studioName}
{studioName, president}
{president, presAddr}

□

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in BCNF. We can be sure of ultimate success, because every time we apply the decomposition rule to a relation R, the two resulting schemas each have fewer attributes than that of R. As we saw in Example 17, when we get down to two attributes, the relation is sure to be in BCNF; often relations with larger sets of attributes are also in BCNF. The strategy is summarized below.

Algorithm 20: BCNF Decomposition Algorithm.

INPUT: A relation R_0 with a set of functional dependencies S_0.

OUTPUT: A decomposition of R_0 into a collection of relations, all of which are in BCNF.

METHOD: The following steps can be applied recursively to any relation R and set of FD's S. Initially, apply them with $R = R_0$ and $S = S_0$.

1. Check whether R is in BCNF. If so, nothing more needs to be done. Return $\{R\}$ as the answer.
2. If there are BCNF violations, let one be $X \to Y$. Use Algorithm 7 to compute X^+. Choose $R_1 = X^+$ as one relation schema and let R_2 have attributes X and those attributes of R that are not in X^+.
3. Use Algorithm 12 to compute the sets of FD's for R_1 and R_2; let these be S_1 and S_2, respectively.
4. Recursively decompose R_1 and R_2 using this algorithm. Return the union of the results of these decompositions.

□

3.5 Exercises for Section 3

Exercise 3.1: For each of the following relation schemas and sets of FD's:

a) $R(A, B, C, D)$ with FD's $AB \to C$, $C \to D$, and $D \to A$.

b) $R(A, B, C, D)$ with FD's $B \to C$ and $B \to D$.

c) $R(A, B, C, D)$ with FD's $AB \to C$, $BC \to D$, $CD \to A$, and $AD \to B$.

d) $R(A, B, C, D)$ with FD's $A \to B$, $B \to C$, $C \to D$, and $D \to A$.

e) $R(A, B, C, D, E)$ with FD's $AB \to C$, $DE \to C$, and $B \to D$.

f) $R(A, B, C, D, E)$ with FD's $AB \to C$, $C \to D$, $D \to B$, and $D \to E$.

do the following:

i) Indicate all the BCNF violations. Do not forget to consider FD's that are not in the given set, but follow from them. However, it is not necessary to give violations that have more than one attribute on the right side.

ii) Decompose the relations, as necessary, into collections of relations that are in BCNF.

Exercise 3.2: We mentioned in Section 3.4 that we would exercise our option to expand the right side of an FD that is a BCNF violation if possible. Consider a relation R whose schema is the set of attributes $\{A, B, C, D\}$ with FD's $A \to B$ and $A \to C$. Either is a BCNF violation, because the only key for R is $\{A, D\}$. Suppose we begin by decomposing R according to $A \to B$. Do we ultimately get the same result as if we first expand the BCNF violation to $A \to BC$? Why or why not?

! Exercise 3.3: Let R be as in Exercise 3.2, but let the FD's be $A \to B$ and $B \to C$. Again compare decomposing using $A \to B$ first against decomposing by $A \to BC$ first.

! Exercise 3.4: Suppose we have a relation schema $R(A, B, C)$ with FD $A \to B$. Suppose also that we decide to decompose this schema into $S(A, B)$ and $T(B, C)$. Give an example of an instance of relation R whose projection onto S and T and subsequent rejoining as in Section 4.1 does not yield the same relation instance. That is, $\pi_{A,B}(R) \bowtie \pi_{B,C}(R) \neq R$.

4 Decomposition: The Good, Bad, and Ugly

So far, we observed that before we decompose a relation schema into BCNF, it can exhibit anomalies; after we decompose, the resulting relations do not exhibit anomalies. That's the "good." But decomposition can also have some bad, if not downright ugly, consequences. In this section, we shall consider three distinct properties we would like a decomposition to have.

1. *Elimination of Anomalies* by decomposition as in Section 3.
2. *Recoverability of Information.* Can we recover the original relation from the tuples in its decomposition?
3. *Preservation of Dependencies.* If we check the projected FD's in the relations of the decomposition, can we can be sure that when we reconstruct the original relation from the decomposition by joining, the result will satisfy the original FD's?

It turns out that the BCNF decomposition of Algorithm 20 gives us (1) and (2), but does not necessarily give us all three. In Section 5 we shall see another way to pick a decomposition that gives us (2) and (3) but does not necessarily give us (1). In fact, there is no way to get all three at once.

4.1 Recovering Information from a Decomposition

Since we learned that every two-attribute relation is in BCNF, why did we have to go through the trouble of Algorithm 20? Why not just take any relation R and decompose it into relations, each of whose schemas is a pair of R's attributes? The answer is that the data in the decomposed relations, even if their tuples were each the projection of a relation instance of R, might not allow us to join the relations of the decomposition and get the instance of R back. If we do get R back, then we say the decomposition has a *lossless join*.

However, if we decompose using Algorithm 20, where all decompositions are motivated by a BCNF-violating FD, then the projections of the original tuples can be joined again to produce all and only the original tuples. We shall consider why here. Then, in Section 4.2 we shall give an algorithm called the "chase," for testing whether the projection of a relation onto any decomposition allows us to recover the relation by rejoining.

To simplify the situation, consider a relation $R(A, B, C)$ and an FD $B \to C$ that is a BCNF violation. The decomposition based on the FD $B \to C$ separates the attributes into relations $R_1(A, B)$ and $R_2(B, C)$.

Let t be a tuple of R. We may write $t = (a, b, c)$, where a, b, and c are the components of t for attributes A, B, and C, respectively. Tuple t projects as (a, b) in $R_1(A, B) = \pi_{A,B}(R)$ and as (b, c) in $R_2(B, C) = \pi_{B,C}(R)$. When we compute the natural join $R_1 \bowtie R_2$, these two projected tuples join, because they agree on the common B component (they both have b there). They give us $t = (a, b, c)$, the tuple we started with, in the join. That is, regardless of what tuple t we started with, we can always join its projections to get t back.

However, getting back those tuples we started with is not enough to assure that the original relation R is truly represented by the decomposition. Consider what happens if there are two tuples of R, say $t = (a, b, c)$ and $v = (d, b, e)$. When we project t onto $R_1(A, B)$ we get $u = (a, b)$, and when we project v onto $R_2(B, C)$ we get $w = (b, e)$. These tuples also match in the natural join, and the resulting tuple is $x = (a, b, e)$. Is it possible that x is a bogus tuple? That is, could (a, b, e) not be a tuple of R?

Since we assume the FD $B \to C$ for relation R, the answer is "no." Recall that this FD says any two tuples of R that agree in their B components must also agree in their C components. Since t and v agree in their B components, they also agree on their C components. That means $c = e$; i.e., the two values we supposed were different are really the same. Thus, tuple (a, b, e) of R is really (a, b, c); that is, $x = t$.

Since t is in R, it must be that x is in R. Put another way, as long as FD $B \to C$ holds, the joining of two projected tuples cannot produce a bogus tuple.

Rather, every tuple produced by the natural join is guaranteed to be a tuple of R.

This argument works in general. We assumed A, B, and C were each single attributes, but the same argument would apply if they were any sets of attributes X, Y and Z. That is, if $Y \to Z$ holds in R, whose attributes are $X \cup Y \cup Z$, then $R = \pi_{X \cup Y}(R) \bowtie \pi_{Y \cup Z}(R)$.

We may conclude:

- If we decompose a relation according to Algorithm 20, then the original relation can be recovered exactly by the natural join.

To see why, we argued above that at any one step of the recursive decomposition, a relation is equal to the join of its projections onto the two components. If those components are decomposed further, they can also be recovered by the natural join from their decomposed relations. Thus, an easy induction on the number of binary decomposition steps says that the original relation is always the natural join of whatever relations it is decomposed into. We can also prove that the natural join is associative and commutative, so the order in which we perform the natural join of the decomposition components does not matter.

The FD $Y \to Z$, or its symmetric FD $Y \to X$, is essential. Without one of these FD's, we might not be able to recover the original relation. Here is an example.

Example 21: Suppose we have the relation $R(A, B, C)$ as above, but neither of the FD's $B \to A$ nor $B \to C$ holds. Then R might consist of the two tuples

A	B	C
1	2	3
4	2	5

The projections of R onto the relations with schemas $\{A, B\}$ and $\{B, C\}$ are $R_1 = \pi_{AB}(R) =$

A	B
1	2
4	2

and $R_2 = \pi_{BC}(R) =$

B	C
2	3
2	5

respectively. Since all four tuples share the same B-value, 2, each tuple of one relation joins with both tuples of the other relation. When we try to reconstruct R by the natural join of the projected relations, we get $R_3 = R_1 \bowtie R_2 =$

Is Join the Only Way to Recover?

We have assumed that the only possible way we could reconstruct a relation from its projections is to use the natural join. However, might there be some other algorithm to reconstruct the original relation that would work even in cases where the natural join fails? There is in fact no such other way. In Example 21, the relations R and R_3 are different instances, yet have exactly the same projections onto $\{A, B\}$ and $\{B, C\}$, namely the instances we called R_1 and R_2, respectively. Thus, given R_1 and R_2, no algorithm whatsoever can tell whether the original instance was R or R_3.

Moreover, this example is not unusual. Given any decomposition of a relation with attributes $X \cup Y \cup Z$ into relations with schemas $X \cup Y$ and $Y \cup Z$, where neither $Y \to X$ nor $Y \to Z$ holds, we can construct an example similar to Example 21 where the original instance cannot be determined from its projections.

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5

That is, we get "too much"; we get two bogus tuples, $(1, 2, 5)$ and $(4, 2, 3)$, that were not in the original relation R. □

4.2 The Chase Test for Lossless Join

In Section 4.1 we argued why a particular decomposition, that of $R(A, B, C)$ into $\{A, B\}$ and $\{B, C\}$, with a particular FD, $B \to C$, had a lossless join. Now, consider a more general situation. We have decomposed relation R into relations with sets of attributes $S_1, S_2, \ldots, S_k$. We have a given set of FD's F that hold in R. Is it true that if we project R onto the relations of the decomposition, then we can recover R by taking the natural join of all these relations? That is, is it true that $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples t such that for all $i = 1, 2, \ldots, k$, t projected onto the set of attributes S_i is a tuple in $\pi_{S_i}(R)$.

- Any tuple t in R is surely in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$. The reason is that the projection of t onto S_i is surely in $\pi_{S_i}(R)$ for each i, and therefore by our first point above, t is in the result of the join.

- As a consequence, $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$ when the FD's in F hold for R if and only if every tuple in the join is also in R. That is, the membership test is all we need to verify that the decomposition has a lossless join.

The *chase* test for a lossless join is just an organized way to see whether a tuple t in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ can be proved, using the FD's in F, also to be a tuple in R. If t is in the join, then there must be tuples in R, say $t_1, t_2, \ldots, t_k$, such that t is the join of the projections of each t_i onto the set of attributes S_i, for $i = 1, 2, \ldots, k$. We therefore know that t_i agrees with t on the attributes of S_i, but t_i has unknown values in its components not in S_i.

We draw a picture of what we know, called a *tableau*. Assuming R has attributes $A, B, \ldots$ we use $a, b, \ldots$ for the components of t. For t_i, we use the same letter as t in the components that are in S_i, but we subscript the letter with i if the component is not in i. In that way, t_i will agree with t for the attributes of S_i, but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

Example 22: Suppose we have relation $R(A, B, C, D)$, which we have decomposed into relations with sets of attributes $S_1 = \{A, D\}$, $S_2 = \{A, C\}$, and $S_3 = \{B, C, D\}$. Then the tableau for this decomposition is shown in Fig. 9.

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

Figure 9: Tableau for the decomposition of R into $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$

The first row corresponds to set of attributes A and D. Notice that the components for attributes A and D are the unsubscripted letters a and d. However, for the other attributes, b and c, we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple (a, b_1, c_1, d) represents a tuple of R that contributes to $t = (a, b, c, d)$ by being projected onto $\{A, D\}$ and then joined with other tuples. Since the B- and C-components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.

Similarly, the second row has the unsubscripted letters in attributes A and C, while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for $\{B, C, D\}$ and subscript 3 on a. Since

each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters. □

Remember that our goal is to use the given set of FD's F to prove that t is really in R. In order to do so, we "chase" the tableau by applying the FD's in F to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as t (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple t in the join of the projections was actually a tuple of R.

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurences.

Example 23: Let us continue with the decomposition of Example 22, and suppose the given FD's are $A \to B$, $B \to C$, and $CD \to A$. Start with the tableau of Fig. 9. Since the first two rows agree in their A-components, the FD $A \to B$ tells us they must also agree in their B-components. That is, $b_1 = b_2$. We can replace either one with the other, since they are both subscripted. Let us replace b_2 by b_1. Then the resulting tableau is:

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Now, we see that the first two rows have equal B-values, and so we may use the FD $B \to C$ to deduce that their C-components, c_1 and c, are the same. Since c is unsubscripted, we replace c_1 by c, leaving:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

Next, we observe that the first and third rows agree in both columns C and D. Thus, we may apply the FD $CD \to A$ to deduce that these rows also have the same A-value; that is, $a = a_3$. We replace a_3 by a, giving us:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

At this point, we see that the last row has become equal to t, that is, (a, b, c, d). We have proved that if R satisfies the FD's $A \to B$, $B \to C$, and $CD \to A$, then whenever we project onto $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$ and rejoin, what we get must have been in R. In particular, what we get is the same as the tuple of R that we projected onto $\{B, C, D\}$. □

4.3 Why the Chase Works

There are two issues to address:

1. When the chase results in a row that matches the tuple t (i.e., the tableau is shown to have a row with all unsubscripted variables), why must the join be lossless?

2. When, after applying FD's whenever we can, we still find no row of all unsubscripted variables, why must the join not be lossless?

Question (1) is easy to answer. The chase process itself is a proof that one of the projected tuples from R must in fact be the tuple t that is produced by the join. We also know that every tuple in R is sure to come back if we project and join. Thus, the chase has proved that the result of projection and join is exactly R.

For the second question, suppose that we eventually derive a tableau without an unsubscripted row, and that this tableau does not allow us to apply any of the FD's to equate any symbols. Then think of the tableau as an instance of the relation R. It obviously satisfies the given FD's, because none can be applied to equate symbols. We know that the ith row has unsubscripted symbols in the attributes of S_i, the ith relation of the decomposition. Thus, when we project this relation onto the S_i's and take the natural join, we get the tuple with all unsubscripted variables. This tuple is not in R, so we conclude that the join is not lossless.

Example 24: Consider the relation $R(A, B, C, D)$ with the FD $B \to AD$ and the proposed decomposition $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$. Here is the initial tableau:

A	B	C	D
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

When we apply the lone FD, we deduce that $a = a_2$ and $d_1 = d_2$. Thus, the final tableau is:

A	B	C	D
a	b	c_1	d_1
a	b	c	d_1
a_3	b_3	c	d

No more changes can be made because of the given FD's, and there is no row that is fully unsubscripted. Thus, this decomposition does not have a lossless join. We can verify that fact by treating the above tableau as a relation with three tuples. When we project onto $\{A, B\}$, we get $\{(a, b)\}$, $(a_3, b_3)\}$. The projection onto $\{B, C\}$ is $\{(b, c_1)$, (b, c), $(b_3, c)\}$, and the projection onto $\{C, D\}$ is (c_1, d_1), (c, d_1), $(c, d)\}$. If we join the first two projections, we get $\{(a, b, c_1)$, (a, b, c), $(a_3, b_3, c)\}$. Joining this relation with the third projection gives $\{(a, b, c_1, d_1)$, (a, b, c, d_1), (a, b, c, d), (a_3, b_3, c, d_1), $(a_3, b_3, c, d)\}$. Notice that this join has two more tuples than R, and in particular it has the tuple (a, b, c, d), as it must. □

4.4 Dependency Preservation

We mentioned that it is not possible, in some cases, to decompose a relation into BCNF relations that have both the lossless-join and dependency-preservation properties. Below is an example where we need to make a tradeoff between preserving dependencies and BNCF.

Example 25: Suppose we have a relation `Bookings` with attributes:

1. `title`, the name of a movie.
2. `theater`, the name of a theater where the movie is being shown.
3. `city`, the city where the theater is located.

The intent behind a tuple (m, t, c) is that the movie with title m is currently being shown at theater t in city c.

We might reasonably assert the following FD's:

`theater` → `city`
`title city` → `theater`

The first says that a theater is located in one city. The second is not obvious but is based on the common practice of not booking a movie into two theaters in the same city. We shall assert this FD if only for the sake of the example.

Let us first find the keys. No single attribute is a key. For example, `title` is not a key because a movie can play in several theaters at once and in several cities at once.[2] Also, `theater` is not a key, because although `theater` functionally determines `city`, there are multiscreen theaters that show many movies at once. Thus, `theater` does not determine `title`. Finally, `city` is not a key because cities usually have more than one theater and more than one movie playing.

[2]In this example we assume that there are not two "current" movies with the same title, even though we have previously recognized that there could be two movies with the same title made in different years.

On the other hand, two of the three sets of two attributes are keys. Clearly {`title`, `city`} is a key because of the given FD that says these attributes functionally determine `theater`.

It is also true that {`theater`, `title`} is a key, because its closure includes `city` due to the given FD `theater` → `city`. The remaining pair of attributes, `city` and `theater`, do not functionally determine `title`, because of multiscreen theaters, and are therefore not a key. We conclude that the only two keys are

{`title`, `city`}
{`theater`, `title`}

Now we immediately see a BCNF violation. We were given functional dependency `theater` → `city`, but its left side, `theater`, is not a superkey. We are therefore tempted to decompose, using this BCNF-violating FD, into the two relation schemas:

{`theater`, `city`}
{`theater`, `title`}

There is a problem with this decomposition, concerning the FD

`title city`→`theater`

There could be current relations for the decomposed schemas that satisfy the FD `theater` → `city` (which can be checked in the relation {`theater`, `city`}) but that, when joined, yield a relation not satisfying `title city`→`theater`. For instance, the two relations

theater	*city*
Guild	Menlo Park
Park	Menlo Park

and

theater	*title*
Guild	Antz
Park	Antz

are permissible according to the FD's that apply to each of the above relations, but when we join them we get two tuples

theater	*city*	*title*
Guild	Menlo Park	Antz
Park	Menlo Park	Antz

that violate the FD `title city` → `theater`. □

4.5 Exercises for Section 4

Exercise 4.1: Let $R(A, B, C, D, E)$ be decomposed into relations with the following three sets of attributes: $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, C, E\}$. For each of the following sets of FD's, use the chase test to tell whether the decomposition of R is lossless. For those that are not lossless, give an example of an instance of R that returns more than R when projected onto the decomposed relations and rejoined.

a) $B \to E$ and $CE \to A$.

b) $AC \to E$ and $BC \to D$.

c) $A \to D$, $D \to E$, and $B \to D$.

d) $A \to D$, $CD \to E$, and $E \to D$.

! Exercise 4.2: For each of the sets of FD's in Exercise 4.1, are dependencies preserved by the decomposition?

5 Third Normal Form

The solution to the problem illustrated by Example 25 is to relax our BCNF requirement slightly, in order to allow the occasional relation schema that cannot be decomposed into BCNF relations without our losing the ability to check the FD's. This relaxed condition is called "third normal form." In this section we shall give the requirements for third normal form, and then show how to do a decomposition in a manner quite different from Algorithm 20, in order to obtain relations in third normal form that have both the lossless-join and dependency-preservation properties.

5.1 Definition of Third Normal Form

A relation R is in *third normal form* (3NF) if:

- Whenever $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ is a nontrivial FD, either

  $$\{A_1, A_2, \ldots, A_n\}$$

  is a superkey, or those of $B_1, B_2, \ldots, B_m$ that are not among the A's, are each a member of some key (not necessarily the same key).

An attribute that is a member of some key is often said to be *prime*. Thus, the 3NF condition can be stated as "for each nontrivial FD, either the left side is a superkey, or the right side consists of prime attributes only."

Note that the difference between this 3NF condition and the BCNF condition is the clause "is a member of some key (i.e., prime)." This clause "excuses" an FD like `theater` $\to$ `city` in Example 25, because the right side, `city`, is prime.

Other Normal Forms

If there is a "third normal form," what happened to the first two "normal forms"? They indeed were defined, but today there is little use for them. *First normal form* is simply the condition that every component of every tuple is an atomic value. *Second normal form* is a less restrictive verison of 3NF. There is also a "fourth normal form" that we shall meet in Section 6.

5.2 The Synthesis Algorithm for 3NF Schemas

We can now explain and justify how we decompose a relation R into a set of relations such that:

a) The relations of the decomposition are all in 3NF.

b) The decomposition has a lossless join.

c) The decomposition has the dependency-preservation property.

Algorithm 26: Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.

INPUT: A relation R and a set F of functional dependencies that hold for R.

OUTPUT: A decomposition of R into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.

METHOD: Perform the following steps:

1. Find a minimal basis for F, say G.
2. For each functional dependency $X \to A$ in G, use XA as the schema of one of the relations in the decomposition.
3. If none of the relation schemas from Step 2 is a superkey for R, add another relation whose schema is a key for R.

□

Example 27: Consider the relation $R(A, B, C, D, E)$ with FD's $AB \to C$, $C \to B$, and $A \to D$. To start, notice that the given FD's are their own minimal basis. To check, we need to do a bit of work. First, we need to verify that we cannot eliminate any of the given dependencies. That is, we show, using Algorithm 7, that no two of the FD's imply the third. For example, we must take the closure of $\{A, B\}$, the left side of the first FD, using only the

second and third FD's, $C \to B$ and $A \to D$. This closure includes D but not C, so we conclude that the first FD $AB \to C$ is not implied by the second and third FD's. We get a similar conclusion if we try to drop the second or third FD.

We must also verify that we cannot eliminate any attributes from a left side. In this simple case, the only possibility is that we could eliminate A or B from the first FD. For example, if we eliminate A, we would be left with $B \to C$. We must show that $B \to C$ is not implied by the three original FD's, $AB \to C$, $C \to B$, and $A \to D$. With these FD's, the closure of $\{B\}$ is just B, so $B \to C$ does not follow. A similar conclusion is drawn if we try to drop B from $AB \to C$. Thus, we have our minimal basis.

We start the 3NF synthesis by taking the attributes of each FD as a relation schema. That is, we get relations $S_1(A, B, C)$, $S_2(B, C)$, and $S_3(A, D)$. It is never necessary to use a relation whose schema is a proper subset of another relation's schema, so we can drop S_2.

We must also consider whether we need to add a relation whose schema is a key. In this example, R has two keys: $\{A, B, E\}$ and $\{A, C, E\}$, as you can verify. Neither of these keys is a subset of the schemas chosen so far. Thus, we must add one of them, say $S_4(A, B, E)$. The final decomposition of R is thus $S_1(A, B, C)$, $S_3(A, D)$, and $S_4(A, B, E)$. □

5.3 Why the 3NF Synthesis Algorithm Works

We need to show three things: that the lossless-join and dependency-preservation properties hold, and that all the relations of the decomposition are in 3NF.

1. *Lossless Join.* Start with a relation of the decomposition whose set of attributes K is a superkey. Consider the sequence of FD's that are used in Algorithm 7 to expand K to become K^+. Since K is a superkey, we know K^+ is all the attributes. The same sequence of FD applications on the tableau cause the subscripted symbols in the row corresponding to K to be equated to unsubscripted symbols in the same order as the attributes were added to the closure. Thus, the chase test concludes that the decomposition is lossless.

2. *Dependency Preservation.* Each FD of the minimal basis has all its attributes in some relation of the decomposition. Thus, each dependency can be checked in the decomposed relations.

3. *Third Normal Form.* If we have to add a relation whose schema is a key, then this relation is surely in 3NF. The reason is that all attributes of this relation are prime, and thus no violation of 3NF could be present in this relation. For the relations whose schemas are derived from the FD's of a minimal basis, the proof that they are in 3NF is beyond the scope of this book. The argument involves showing that a 3NF violation implies that the basis is not minimal.

5.4 Exercises for Section 5

Exercise 5.1: For each of the relation schemas and sets of FD's of Exercise 3.1:

i) Indicate all the 3NF violations.

ii) Decompose the relations, as necessary, into collections of relations that are in 3NF.

Exercise 5.2: Consider the relation `Courses`(C, T, H, R, S, G), whose attributes may be thought of informally as course, teacher, hour, room, student, and grade. Let the set of FD's for `Courses` be $C \to T$, $HR \to C$, $HT \to R$, $HS \to R$, and $CS \to G$. Intuitively, the first says that a course has a unique teacher, and the second says that only one course can meet in a given room at a given hour. The third says that a teacher can be in only one room at a given hour, and the fourth says the same about students. The last says that students get only one grade in a course.

a) What are all the keys for `Courses`?

b) Verify that the given FD's are their own minimal basis.

c) Use the 3NF synthesis algorithm to find a lossless-join, dependency-preserving decomposition of R into 3NF relations. Are any of the relations not in BCNF?

Exercise 5.3: Consider a relation `Stocks`(B, O, I, S, Q, D), whose attributes may be thought of informally as broker, office (of the broker), investor, stock, quantity (of the stock owned by the investor), and dividend (of the stock). Let the set of FD's for `Stocks` be $S \to D$, $I \to B$, $IS \to Q$, and $B \to O$. Repeat Exercise 5.2 for the relation `Stocks`.

Exercise 5.4: Verify, using the chase, that the decomposition of Example 27 has a lossless join.

!! Exercise 5.5: Suppose we modified Algorithm 20 (BNCF decomposition) so that instead of decomposing a relation R whenever R was not in BCNF, we only decomposed R if it was not in 3NF. Provide a counterexample to show that this modified algorithm would not necessarily produce a 3NF decomposition with dependency preservation.

6 Multivalued Dependencies

A "multivalued dependency" is an assertion that two attributes or sets of attributes are independent of one another. This condition is, as we shall see, a generalization of the notion of a functional dependency, in the sense that

every FD implies the corresponding multivalued dependency. However, there are some situations involving independence of attribute sets that cannot be explained as FD's. In this section we shall explore the cause of multivalued dependencies and see how they can be used in database schema design.

6.1 Attribute Independence and Its Consequent Redundancy

There are occasional situations where we design a relation schema and find it is in BCNF, yet the relation has a kind of redundancy that is not related to FD's. The most common source of redundancy in BCNF schemas is an attempt to put two or more set-valued properties of the key into a single relation.

Example 28: In this example, we shall suppose that stars may have several addresses, which we break into street and city components. The set of addresses is one of the set-valued properties this relation will store. The second set-valued property of stars that we shall put into this relation is the set of titles and years of movies in which the star appeared. Then Fig. 10 is a typical instance of this relation.

name	*street*	*city*	*title*	*year*
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

Figure 10: Sets of addresses independent from movies

We focus in Fig. 10 on Carrie Fisher's two hypothetical addresses and her three best-known movies. There is no reason to associate an address with one movie and not another. Thus, the only way to express the fact that addresses and movies are independent properties of stars is to have each address appear with each movie. But when we repeat address and movie facts in all combinations, there is obvious redundancy. For instance, Fig. 10 repeats each of Carrie Fisher's addresses three times (once for each of her movies) and each movie twice (once for each address).

Yet there is no BCNF violation in the relation suggested by Fig. 10. There are, in fact, no nontrivial FD's at all. For example, attribute `city` is not functionally determined by the other four attributes. There might be a star with two homes that had the same street address in different cities. Then there would be two tuples that agreed in all attributes but `city` and disagreed in `city`. Thus,

$$\texttt{name street title year} \rightarrow \texttt{city}$$

is not an FD for our relation. We leave it to the reader to check that none of the five attributes is functionally determined by the other four. Since there are no nontrivial FD's, it follows that all five attributes form the only key and that there are no BCNF violations. □

6.2 Definition of Multivalued Dependencies

A *multivalued dependency* (abbreviated MVD) is a statement about some relation R that when you fix the values for one set of attributes, then the values in certain other attributes are independent of the values of all the other attributes in the relation. More precisely, we say the MVD

$$A_1 A_2 \cdots A_n \twoheadrightarrow B_1 B_2 \cdots B_m$$

holds for a relation R if when we restrict ourselves to the tuples of R that have particular values for each of the attributes among the A's, then the set of values we find among the B's is independent of the set of values we find among the attributes of R that are not among the A's or B's. Still more precisely, we say this MVD holds if

> For each pair of tuples t and u of relation R that agree on all the A's, we can find in R some tuple v that agrees:
>
> 1. With both t and u on the A's,
> 2. With t on the B's, and
> 3. With u on all attributes of R that are not among the A's or B's.

Note that we can use this rule with t and u interchanged, to infer the existence of a fourth tuple w that agrees with u on the B's and with t on the other attributes. As a consequence, for any fixed values of the A's, the associated values of the B's and the other attributes appear in all possible combinations in different tuples. Figure 11 suggests how v relates to t and u when an MVD holds. However, the A's and B's to not have to appear consecutively.

In general, we may assume that the A's and B's (left side and right side) of an MVD are disjoint. However, as with FD's, it is permissible to add some of the A's to the right side if we wish.

Example 29: In Example 28 we encountered an MVD that in our notation is expressed:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

That is, for each star's name, the set of addresses appears in conjunction with each of the star's movies. For an example of how the formal definition of this MVD applies, consider the first and fourth tuples from Fig. 10:

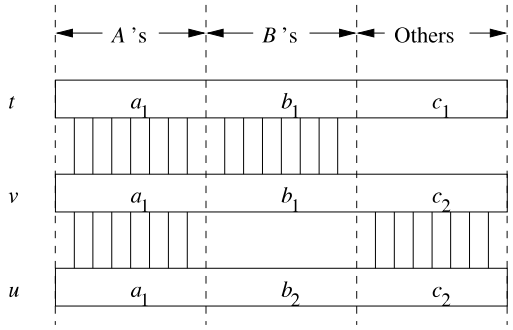

Figure 11: A multivalued dependency guarantees that v exists

name	*street*	*city*	*title*	*year*
`C. Fisher`	`123 Maple St.`	`Hollywood`	`Star Wars`	`1977`
`C. Fisher`	`5 Locust Ln.`	`Malibu`	`Empire Strikes Back`	`1980`

If we let the first tuple be t and the second be u, then the MVD asserts that we must also find in R the tuple that has name `C. Fisher`, a street and city that agree with the first tuple, and other attributes (`title` and `year`) that agree with the second tuple. There is indeed such a tuple; it is the third tuple of Fig. 10.

Similarly, we could let t be the second tuple above and u be the first. Then the MVD tells us that there is a tuple of R that agrees with the second in attributes `name`, `street`, and `city` and with the first in `name`, `title`, and `year`. This tuple also exists; it is the second tuple of Fig. 10. □

6.3 Reasoning About Multivalued Dependencies

There are a number of rules about MVD's that are similar to the rules we learned for FD's in Section 2. For example, MVD's obey

- *Trivial MVD's.* The MVD

  $$A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$$

  holds in any relation if $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$.

- The *transitive rule*, which says that if $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ and $B_1B_2\cdots B_m \twoheadrightarrow C_1C_2\cdots C_k$ hold for some relation, then so does

  $$A_1A_2\cdots A_n \twoheadrightarrow C_1C_2\cdots C_k$$

  Any C's that are also A's must be deleted from the right side.

On the other hand, MVD's do not obey the splitting part of the splitting/combining rule, as the following example shows.

Example 30: Consider again Fig. 10, where we observed the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

If the splitting rule applied to MVD's, we would expect

$$\texttt{name} \twoheadrightarrow \texttt{street}$$

also to be true. This MVD says that each star's street addresses are independent of the other attributes, including `city`. However, that statement is false. Consider, for instance, the first two tuples of Fig. 10. The hypothetical MVD would allow us to infer that the tuples with the streets interchanged:

name	*street*	*city*	*title*	*year*
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple St.	Malibu	Star Wars	1977

were in the relation. But these are not true tuples, because, for instance, the home on 5 Locust Ln. is in Malibu, not Hollywood. □

However, there are several new rules dealing with MVD's that we can learn.

- *FD Promotion.* Every FD is an MVD. That is, if

  $$A_1A_2\cdots A_n \to B_1B_2\cdots B_m$$

  then $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$.

  To see why, suppose R is some relation for which the FD

  $$A_1A_2\cdots A_n \to B_1B_2\cdots B_m$$

  holds, and suppose t and u are tuples of R that agree on the A's. To show that the MVD $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ holds, we have to show that R also contains a tuple v that agrees with t and u on the A's, with t on the B's, and with u on all other attributes. But v can be u. Surely u agrees with t and u on the A's, because we started by assuming that these two tuples agree on the A's. The FD $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ assures us that u agrees with t on the B's. And of course u agrees with itself on the other attributes. Thus, whenever an FD holds, the corresponding MVD holds.

- *Complementation Rule.* If $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ is an MVD for relation R, then R also satisfies $A_1A_2\cdots A_n \twoheadrightarrow C_1C_2\cdots C_k$, where the C's are all attributes of R not among the A's and B's.

That is, swapping the B's between two tuples that agree in the A's has the same effect as swapping the C's.

Example 31 : Again consider the relation of Fig. 10, for which we asserted the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

The complementation rule says that

$$\texttt{name} \twoheadrightarrow \texttt{title year}$$

must also hold in this relation, because `title` and `year` are the attributes not mentioned in the first MVD. The second MVD intuitively means that each star has a set of movies starred in, which are independent of the star's addresses. □

An MVD whose right side is a subset of the left side is trivial — it holds in every relation. However, an interesting consequence of the complementation rule is that there are some other MVD's that are trivial, but that look distinctly nontrivial.

- *More Trivial MVD's.* If all the attributes of relation R are

  $$\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$$

  then $A_1A_2 \cdots A_n \twoheadrightarrow B_1B_2 \cdots B_m$ holds in R.

To see why these additional trivial MVD's hold, notice that if we take two tuples that agree in $A_1, A_2, \ldots, A_n$ and swap their components in attributes $B_1, B_2, \ldots, B_m$, we get the same two tuples back, although in the opposite order.

6.4 Fourth Normal Form

The redundancy that we found in Section 6.1 to be caused by MVD's can be eliminated if we use these dependencies for decomposition. In this section we shall introduce a new normal form, called "fourth normal form." In this normal form, all nontrivial MVD's are eliminated, as are all FD's that violate BCNF. As a result, the decomposed relations have neither the redundancy from FD's that we discussed in Section 3.1 nor the redundancy from MVD's that we discussed in Section 6.1.

The "fourth normal form" condition is essentially the BCNF condition, but applied to MVD's instead of FD's. Formally:

- A relation R is in *fourth normal form* (4NF) if whenever

  $$A_1A_2 \cdots A_n \twoheadrightarrow B_1B_2 \cdots B_m$$

is a nontrivial MVD, $\{A_1, A_2, \ldots, A_n\}$ is a superkey.

That is, if a relation is in 4NF, then every nontrivial MVD is really an FD with a superkey on the left. Note that the notions of keys and superkeys depend on FD's only; adding MVD's does not change the definition of "key."

Example 32: The relation of Fig. 10 violates the 4NF condition. For example,

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is a nontrivial MVD, yet `name` by itself is not a superkey. In fact, the only key for this relation is all the attributes. □

Fourth normal form is truly a generalization of BCNF. Recall from Section 6.3 that every FD is also an MVD. Thus, every BCNF violation is also a 4NF violation. Put another way, every relation that is in 4NF is therefore in BCNF.

However, there are some relations that are in BCNF but not 4NF. Figure 10 is a good example. The only key for this relation is all five attributes, and there are no nontrivial FD's. Thus it is surely in BCNF. However, as we observed in Example 32, it is not in 4NF.

6.5 Decomposition into Fourth Normal Form

The 4NF decomposition algorithm is quite analogous to the BCNF decomposition algorithm.

Algorithm 33: Decomposition into Fourth Normal Form.

INPUT: A relation R_0 with a set of functional and multivalued dependencies S_0.

OUTPUT: A decomposition of R_0 into relations all of which are in 4NF. The decomposition has the lossless-join property.

METHOD: Do the following steps, with $R = R_0$ and $S = S_0$:

1. Find a 4NF violation in R, say $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$, where

   $$\{A_1, A_2, \ldots, A_n\}$$

   is not a superkey. Note this MVD could be a true MVD in S, or it could be derived from the corresponding FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ in S, since every FD is an MVD. If there is none, return; R by itself is a suitable decomposition.

2. If there is such a 4NF violation, break the schema for the relation R that has the 4NF violation into two schemas:

(a) R_1, whose schema is A's and the B's.

(b) R_2, whose schema is the A's and all attributes of R that are not among the A's or B's.

3. Find the FD's and MVD's that hold in R_1 and R_2 (Section 7 explains how to do this task in general, but often this "projection" of dependencies is straightforward). Recursively decompose R_1 and R_2 with respect to their projected dependencies.

□

Example 34: Let us continue Example 32. We observed that

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

was a 4NF violation. The decomposition rule above tells us to replace the five-attribute schema by one schema that has only the three attributes in the above MVD and another schema that consists of the left side, `name`, plus the attributes that do not appear in the MVD. These attributes are `title` and `year`, so the following two schemas

$$\{\texttt{name, street, city}\}$$
$$\{\texttt{name, title, year}\}$$

are the result of the decomposition. In each schema there are no nontrivial multivalued (or functional) dependencies, so they are in 4NF. Note that in the relation with schema {`name, street, city`}, the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is trivial since it involves all attributes. Likewise, in the relation with schema {`name, title, year`}, the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{title year}$$

is trivial. Should one or both schemas of the decomposition not be in 4NF, we would have had to decompose the non-4NF schema(s). □

As for the BCNF decomposition, each decomposition step leaves us with schemas that have strictly fewer attributes than we started with, so eventually we get to schemas that need not be decomposed further; that is, they are in 4NF. Moreover, the argument justifying the decomposition that we gave in Section 4.1 carries over to MVD's as well. When we decompose a relation because of an MVD $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$, this dependency is enough to justify the claim that we can reconstruct the original relation from the relations of the decomposition.

We shall, in Section 7, give an algorithm by which we can verify that the MVD used to justify a 4NF decomposition also proves that the decomposition has a lossless join. Also in that section, we shall show how it is possible, although time-consuming, to perform the projection of MVD's onto the decomposed relations. This projection is required if we are to decide whether or not further decomposition is necessary.

6.6 Relationships Among Normal Forms

As we have mentioned, 4NF implies BCNF, which in turn implies 3NF. Thus, the sets of relation schemas (including dependencies) satisfying the three normal forms are related as in Fig. 12. That is, if a relation with certain dependencies is in 4NF, it is also in BCNF and 3NF. Also, if a relation with certain dependencies is in BCNF, then it is in 3NF.

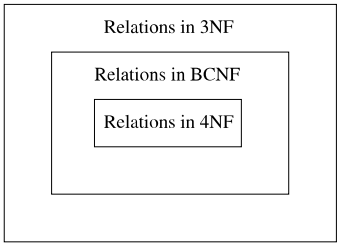

Figure 12: 4NF implies BCNF implies 3NF

Another way to compare the normal forms is by the guarantees they make about the set of relations that result from a decomposition into that normal form. These observations are summarized in the table of Fig. 13. That is, BCNF (and therefore 4NF) eliminates the redundancy and other anomalies that are caused by FD's, while only 4NF eliminates the additional redundancy that is caused by the presence of MVD's that are not FD's. Often, 3NF is enough to eliminate this redundancy, but there are examples where it is not. BCNF does not guarantee preservation of FD's, and none of the normal forms guarantee preservation of MVD's, although in typical cases the dependencies are preserved.

Property	3NF	BCNF	4NF
Eliminates redundancy due to FD's	No	Yes	Yes
Eliminates redundancy due to MVD's	No	No	Yes
Preserves FD's	Yes	No	No
Preserves MVD's	No	No	No

Figure 13: Properties of normal forms and their decompositions

6.7 Exercises for Section 6

Exercise 6.1 : Suppose we have a relation $R(A, B, C)$ with an MVD $A \twoheadrightarrow B$.

If we know that the tuples (a, b_1, c_1), (a, b_2, c_2), and (a, b_3, c_3) are in the current instance of R, what other tuples do we know must also be in R?

Exercise 6.2: Suppose we have a relation in which we want to record for each person their name, Social Security number, and birthdate. Also, for each child of the person, the name, Social Security number, and birthdate of the child, and for each automobile the person owns, its serial number and make. To be more precise, this relation has all tuples

$$(n, s, b, cn, cs, cb, as, am)$$

such that

1. n is the name of the person with Social Security number s.
2. b is n's birthdate.
3. cn is the name of one of n's children.
4. cs is cn's Social Security number.
5. cb is cn's birthdate.
6. as is the serial number of one of n's automobiles.
7. am is the make of the automobile with serial number as.

For this relation:

a) Tell the functional and multivalued dependencies we would expect to hold.

b) Suggest a decomposition of the relation into 4NF.

Exercise 6.3: For each of the following relation schemas and dependencies

a) $R(A, B, C, D)$ with MVD's $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$.

b) $R(A, B, C, D)$ with MVD's $A \twoheadrightarrow B$ and $B \twoheadrightarrow CD$.

c) $R(A, B, C, D)$ with MVD $AB \twoheadrightarrow C$ and FD $B \to D$.

d) $R(A, B, C, D, E)$ with MVD's $A \twoheadrightarrow B$ and $AB \twoheadrightarrow C$ and FD's $A \to D$ and $AB \to E$.

do the following:

i) Find all the 4NF violations.

ii) Decompose the relations into a collection of relation schemas in 4NF.

Exercise 6.4: Give informal arguments why we would not expect any of the five attributes in Example 28 to be functionally determined by the other four.

7 An Algorithm for Discovering MVD's

Reasoning about MVD's, or combinations of MVD's and FD's, is rather more difficult than reasoning about FD's alone. For FD's, we have Algorithm 7 to decide whether or not an FD follows from some given FD's. In this section, we shall first show that the closure algorithm is really the same as the chase algorithm we studied in Section 4.2. The ideas behind the chase can be extended to incorporate MVD's as well as FD's. Once we have that tool in place, we can solve all the problems we need to solve about MVD's and FD's, such as finding whether an MVD follows from given dependencies or projecting MVD's and FD's onto the relations of a decomposition.

7.1 The Closure and the Chase

In Section 2.4 we saw how to take a set of attributes X and compute its closure X^+ of all attributes that functionally depend on X. In that manner, we can test whether an FD $X \to Y$ follows from a given set of FD's F, by closing X with respect to F and seeing whether $Y \subseteq X^+$. We could see the closure as a variant of the chase, in which the starting tableau and the goal condition are different from what we used in Section 4.2.

Suppose we start with a tableau that consists of two rows. These rows agree in the attributes of X and disagree in all other attributes. If we apply the FD's in F to chase this tableau, we shall equate the symbols in exactly those columns that are in $X^+ - X$. Thus, a chase-based test for whether $X \to Y$ follows from F can be summarized as:

1. Start with a tableau having two rows that agree only on X.
2. Chase the tableau using the FD's of F.
3. If the final tableau agrees in all columns of Y, then $X \to Y$ holds; otherwise it does not.

Example 35: Let us repeat Example 8, where we had a relation

$$R(A, B, C, D, E, F)$$

with FD's $AB \to C$, $BC \to AD$, $D \to E$, and $CF \to B$. We want to test whether $AB \to D$ holds. Start with the tableau:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_2	d_2	e_2	f_2

We can apply $AB \to C$ to infer $c_1 = c_2$; say both become c_1. The resulting tableau is:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_1	d_2	e_2	f_2

Next, apply $BC \rightarrow AD$ to infer that $d_1 = d_2$, and apply $D \rightarrow E$ to infer $e_1 = e_2$. At this point, the tableau is:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_1	d_1	e_1	f_2

and we can go no further. Since the two tuples now agree in the D column, we know that $AB \rightarrow D$ does follow from the given FD's. □

7.2 Extending the Chase to MVD's

The method of inferring an FD using the chase can be applied to infer MVD's as well. When we try to infer an FD, we are asking whether we can conclude that two possibly unequal values must indeed be the same. When we apply an FD $X \rightarrow Y$, we search for pairs of rows in the tableau that agree on all the columns of X, and we force the symbols in each column of Y to be equal.

However, MVD's do not tell us to conclude symbols are equal. Rather, $X \twoheadrightarrow Y$ tells us that if we find two rows of the tableau that agree in X, then we can form two new tuples by swapping all their components in the attributes of Y; the resulting two tuples must also be in the relation, and therefore in the tableau. Likewise, if we want to infer some MVD $X \twoheadrightarrow Y$ from given FD's and MVD's, we start with a tableau consisting of two tuples that agree in X and disagree in all attributes not in the set X. We apply the given FD's to equate symbols, and we apply the given MVD's to swap the values in certain attributes between two existing rows of the tableau in order to add new rows to the tableau. If we ever discover that one of the original tuples, with its components for Y replaced by those of the other original tuple, is in the tableau, then we have inferred the MVD.

There is a point of caution to be observed in this more complex chase process. Since symbols may get equated and replaced by other symbols, we may not recognize that we have created one of the desired tuples, because some of the original symbols may be replaced by others. The simplest way to avoid a problem is to define the target tuple initially, and never change its symbols. That is, let the target row be one with an unsubscripted letter in each component. Let the two initial rows of the tableau for the test of $X \twoheadrightarrow Y$ have the unsubscripted letters in X. Let the first row also have unsubscripted letters in Y, and let the second row have the unsubscripted letters in all attributes not in X or Y. Fill in the other positions of the two rows with new symbols that each occur only once. When we equate subscripted and unsubscripted symbols, always replace a subscripted one by the unsubscripted one, as we did in Section 4.2. Then, when applying the chase, we have only to ask whether the all-unsubscripted-letters row ever appears in the tableau.

Example 36: Suppose we have a relation $R(A, B, C, D)$ with given dependencies $A \to B$ and $B \twoheadrightarrow C$. We wish to prove that $A \twoheadrightarrow C$ holds in R. Start with the two-row tableau that represents $A \twoheadrightarrow C$:

A	B	C	D
a	b_1	c	d_1
a	b	c_2	d

Notice that our target row is (a, b, c, d). Both rows of the tableau have the unsubscripted letter in the column for A. The first row has the unsubscripted letter in C, and the second row has unsubscripted letters in the remaining columns.

We first apply the FD $A \to B$ to infer that $b = b_1$. We must therefore replace the subscripted b_1 by the unsubscripted b. The tableau becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d

Next, we apply the MVD $B \twoheadrightarrow C$, since the two rows now agree in the B column. We swap the C columns to get two more rows which we add to the tableau, which becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d
a	b	c_2	d_1
a	b	c	d

We have now a row with all unsubscripted symbols, which proves that $A \twoheadrightarrow C$ holds in relation R. Notice how the tableau manipulations really give a proof that $A \twoheadrightarrow C$ holds. This proof is: "Given two tuples of R that agree in A, they must also agree in B because $A \to B$. Since they agree in B, we can swap their C components by $B \twoheadrightarrow C$, and the resulting tuples will be in R. Thus, if two tuples of R agree in A, the tuples that result when we swap their C's are also in R; i.e., $A \twoheadrightarrow C$." □

Example 37: There is a surprising rule for FD's and MVD's that says whenever there is an MVD $X \twoheadrightarrow Y$, and any FD whose right side is a (not necessarily proper) subset of Y, say Z, then $X \to Z$. We shall use the chase process to prove a simple example of this rule. Let us be given relation $R(A, B, C, D)$ with MVD $A \twoheadrightarrow BC$ and FD $D \to C$. We claim that $A \to C$.

Since we are trying to prove an FD, we don't have to worry about a target tuple of unsubscripted letters. We can start with any two tuples that agree in A and disagree in every other column. such as:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_2	d_2

Our goal is to prove that $c_1 = c_2$.

The only thing we can do to start is to apply the MVD $A \twoheadrightarrow BC$, since the two rows agree on A, but no other columns. When we swap the B and C columns of these two rows, we get two new rows to add:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_2	d_2
a	b_2	c_2	d_1
a	b_1	c_1	d_2

Now, we have pairs of rows that agree in D, so we can apply the FD $D \to C$. For instance, the first and third rows have the same D-value d_1, so we can apply the FD and conclude $c_1 = c_2$. That is our goal, so we have proved $A \to C$. The new tableau is:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_1	d_2
a	b_2	c_1	d_1
a	b_1	c_1	d_2

It happens that no further changes are possible, using the given dependencies. However, that doesn't matter, since we already proved what we need. □

7.3 Why the Chase Works for MVD's

The arguments are essentially the same as we have given before. Each step of the chase, whether it equates symbols or generates new rows, is a true observation about tuples of the given relation R that is justified by the FD or MVD that we apply in that step. Thus, a positive conclusion of the chase is always a proof that the concluded FD or MVD holds in R.

When the chase ends in failure — the goal row (for an MVD) or the desired equality of symbols (for an FD) is not produced — then the final tableau is a counterexample. It satisfies the given dependencies, or else we would not be finished making changes. However, it does not satisfy the dependency we were trying to prove.

There is one other issue that did not come up when we performed the chase using only FD's. Since the chase with MVD's adds rows to the tableau, how do we know we ever terminate the chase? Could we keep adding rows forever, never reaching our goal, but not sure that after a few more steps we would achieve that goal? Fortunately, that cannot happen. The reason is that we

never create any new symbols. We start out with at most two symbols in each of k columns, and all rows we create will have one of these two symbols in its component for that column. Thus, we cannot ever have more than 2^k rows in our tableau, if k is the number of columns. The chase with MVD's can take exponential time, but it cannot run forever.

7.4 Projecting MVD's

Recall that our reason for wanting to infer MVD's was to perform a cascade of decompositions leading to 4NF relations. To do that task, we need to be able to project the given dependencies onto the schemas of the two relations that we get in the first step of the decomposition. Only then can we know whether they are in 4NF or need to be decomposed further.

In the worst case, we have to test every possible FD and MVD for each of the decomposed relations. The chase test is applied on the full set of attributes of the original relation. However, the goal for an MVD is to produce a row of the tableau that has unsubscripted letters in all the attributes of one of the relations of the decomposition; that row may have any letters in the other attributes. The goal for an FD is the same: equality of the symbols in a given column.

Example 38: Suppose we have a relation $R(A, B, C, D, E)$ that we decompose, and let one of the relations of the decomposition be $S(A, B, C)$. Suppose that the MVD $A \twoheadrightarrow CD$ holds in R. Does this MVD imply any dependency in S? We claim that $A \twoheadrightarrow C$ holds in S, as does $A \twoheadrightarrow B$ (by the complementation rule). Let us verify that $A \twoheadrightarrow C$ holds in S. We start with the tableau:

A	B	C	D	E
a	b_1	c	d_1	e_1
a	b	c_2	d	e

Use the MVD of R, $A \twoheadrightarrow CD$ to swap the C and D components of these two rows to get two new rows:

A	B	C	D	E
a	b_1	c	d_1	e_1
a	b	c_2	d	e
a	b_1	c_2	d	e_1
a	b	c	d_1	e

Notice that the last row has unsubscripted symbols in all the attributes of S, that is, A, B, and C. That is enough to conclude that $A \twoheadrightarrow C$ holds in S. □

Often, our search for FD's and MVD's in the projected relations does not have to be completely exhaustive. Here are some simplifications.

1. It is surely not necessary to check the trivial FD's and MVD's.
2. For FD's, we can restrict ourselves to looking for FD's with a singleton right side, because of the combining rule for FD's.
3. An FD or MVD whose left side does not contain the left side of any given dependency surely cannot hold, since there is no way for its chase test to get started. That is, the two rows with which you start the test are unchanged by the given dependencies.

7.5 Exercises for Section 7

Exercise 7.1: Use the chase test to tell whether each of the following dependencies hold in a relation $R(A, B, C, D, E)$ with the dependencies $A \twoheadrightarrow BC$, $B \rightarrow D$, and $C \twoheadrightarrow E$.

a) $A \rightarrow D$.

b) $A \twoheadrightarrow D$.

c) $A \rightarrow E$.

d) $A \twoheadrightarrow E$.

! Exercise 7.2: If we project the relation R of Exercise 7.1 onto $S(A, C, E)$, what nontrivial FD's and MVD's hold in S?

! Exercise 7.3: Show the following rules for MVD's. In each case, you can set up the proof as a chase test, but you must think a little more generally than in the examples, since the set of attributes are arbitrary sets X, Y, Z, and the other unnamed attributes of the relation in which these dependencies hold.

a) The *Union Rule*. If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y \cup Z)$.

b) The *Intersection Rule*. If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y \cap Z)$.

c) The *Difference Rule*. If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y - Z)$.

d) *Removing attributes shared by left and right side*. If $X \twoheadrightarrow Y$ holds, then $X \twoheadrightarrow (Y - X)$ holds.

! Exercise 7.4: Give counterexample relations to show why the following rules for MVD's do *not* hold. *Hint*: apply the chase test and see what happens.

a) If $A \twoheadrightarrow BC$, then $A \twoheadrightarrow B$.

b) If $A \twoheadrightarrow B$, then $A \rightarrow B$.

c) If $AB \twoheadrightarrow C$, then $A \twoheadrightarrow C$.

8 Summary

- *Functional Dependencies*: A functional dependency is a statement that two tuples of a relation that agree on some particular set of attributes must also agree on some other particular set of attributes.
- *Keys of a Relation*: A superkey for a relation is a set of attributes that functionally determines all the attributes of the relation. A key is a superkey, no proper subset of which is also a superkey.
- *Reasoning About Functional Dependencies*: There are many rules that let us infer that one FD $X \rightarrow A$ holds in any relation instance that satisfies some other given set of FD's. To verify that $X \rightarrow A$ holds, compute the closure of X, using the given FD's to expand X until it includes A.
- *Minimal Basis for a set of FD's*: For any set of FD's, there is at least one minimal basis, which is a set of FD's equivalent to the original (each set implies the other set), with singleton right sides, no FD that can be eliminated while preserving equivalence, and no attribute in a left side that can be eliminated while preserving equivalence.
- *Boyce-Codd Normal Form*: A relation is in BCNF if the only nontrivial FD's say that some superkey functionally determines one or more of the other attributes. A major benefit of BCNF is that it eliminates redundancy caused by the existence of FD's.
- *Lossless-Join Decomposition*: A useful property of a decomposition is that the original relation can be recovered exactly by taking the natural join of the relations in the decomposition. Any decomposition gives us back at least the tuples with which we start, but a carelessly chosen decomposition can give tuples in the join that were not in the original relation.
- *Dependency-Preserving Decomposition*: Another desirable property of a decomposition is that we can check all the functional dependencies that hold in the original relation by checking FD's in the decomposed relations.
- *Third Normal Form*: Sometimes decomposition into BCNF can lose the dependency-preservation property. A relaxed form of BCNF, called 3NF, allows an FD $X \rightarrow A$ even if X is not a superkey, provided A is a member of some key. 3NF does not guarantee to eliminate all redundancy due to FD's, but often does so.
- *The Chase*: We can test whether a decomposition has the lossless-join property by setting up a tableau — a set of rows that represent tuples of the original relation. We chase a tableau by applying the given functional dependencies to infer that certain pairs of symbols must be the same. The decomposition is lossless with respect to a given set of FD's if and only if the chase leads to a row identical to the tuple whose membership in the join of the projected relations we assumed.

- *Synthesis Algorithm for 3NF*: If we take a minimal basis for a given set of FD's, turn each of these FD's into a relation, and add a key for the relation, if necessary, the result is a decomposition into 3NF that has the lossless-join and dependency-preservation properties.

- *Multivalued Dependencies*: A multivalued dependency is a statement that two sets of attributes in a relation have sets of values that appear in all possible combinations.

- *Fourth Normal Form*: MVD's can also cause redundancy in a relation. 4NF is like BCNF, but also forbids nontrivial MVD's whose left side is not a superkey. It is possible to decompose a relation into 4NF without losing information.

- *Reasoning About MVD's*: We can infer MVD's and FD's from a given set of MVD's and FD's by a chase process. We start with a two-row tableau that represent the dependency we are trying to prove. FD's are applied by equating symbols, and MVD's are applied by adding rows to the tableau that have the appropriate components interchanged.

9 References

Third normal form was described in [6]. This paper introduces the idea of functional dependencies, as well as the basic relational concept. Boyce-Codd normal form is in a later paper [7].

Multivalued dependencies and fourth normal form were defined by Fagin in [9]. However, the idea of multivalued dependencies also appears independently in [8] and [11].

Armstrong was the first to study rules for inferring FD's [2]. The rules for FD's that we have covered here (including what we call "Armstrong's axioms") and rules for inferring MVD's as well, come from [3].

The technique for testing an FD by computing the closure for a set of attributes is from [4], as is the fact that a minimal basis provides a 3NF decomposition. The fact that this decomposition provides the lossless-join and dependency-preservation propoerties is from [5].

The tableau test for the lossless-join property and the chase are from [1]. More information and the history of the idea is found in [10].

1. A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Transactions on Database Systems* **4**:3, pp. 297-314, 1979.

2. W. W. Armstrong, "Dependency structures of database relationships," *Proceedings of the 1974 IFIP Congress*, pp. 580–583.

3. C. Beeri, R. Fagin, and J. H. Howard, "A complete axiomatization for functional and multivalued dependencies," *ACM SIGMOD International Conference on Management of Data*, pp. 47–61, 1977.

4. P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Transactions on Database Systems* **1**:4, pp. 277–298, 1976.

5. J. Biskup, U. Dayal, and P. A. Bernstein, "Synthesizing independent database schemas," *ACM SIGMOD International Conference on Management of Data*, pp. 143–152, 1979.

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

7. E. F. Codd, "Further normalization of the data base relational model," in *Database Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

8. C. Delobel, "Normalization and hierarchical dependencies in the relational data model," *ACM Transactions on Database Systems* **3**:3, pp. 201–222, 1978.

9. R. Fagin, "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems* **2**:3, pp. 262–278, 1977.

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

11. C. Zaniolo and M. A. Melkanoff, "On the design of relational database schemata," *ACM Transactions on Database Systems* **6**:1, pp. 1–47, 1981.