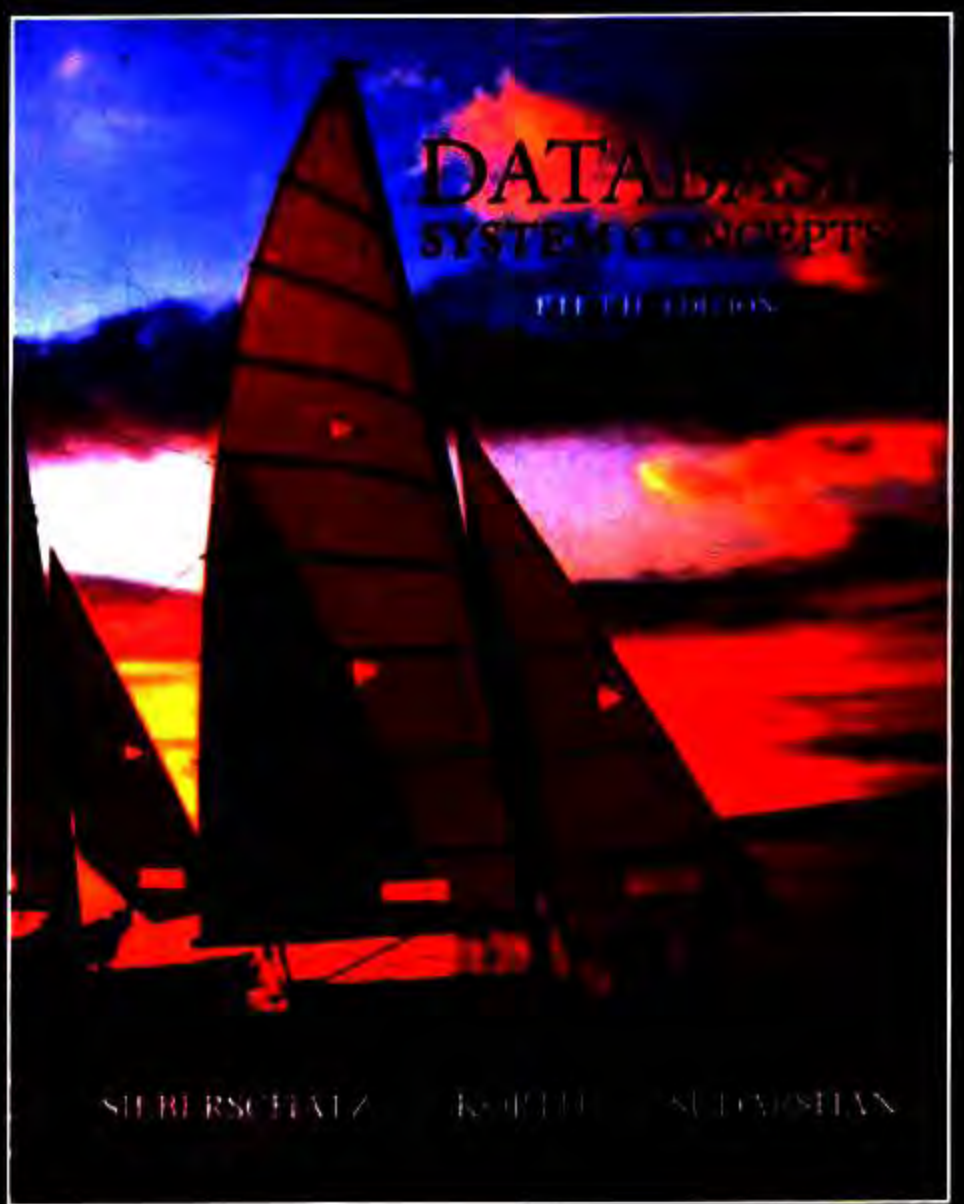
计 算 机 科 学 丛 书

原书第5版

数据库系统概念

Abraham Silberschatz 耶鲁大学
Henry F. Korth 里海大学
S. Sudarshan 印度理工学院
著
杨冬青 马秀莉 唐世渭 等译
北京大学



Database System Concepts

Fifth Edition

机械工业出版社
China Machine Press

Preface

Database management has evolved from a specialized computer application to a central component of a modern computing environment, and, as a result, knowledge about database systems has become an essential part of an education in computer science. In this text, we present the fundamental concepts of database management. These concepts include aspects of database design, database languages, and database-system implementation.

This text is intended for a first course in databases at the junior or senior undergraduate, or first-year graduate, level. In addition to basic material for a first course, the text contains advanced material that can be used for course supplements, or as introductory material for an advanced course.

We assume only a familiarity with basic data structures, computer organization, and a high-level programming language such as Java, C, or Pascal. We present concepts as intuitive descriptions, many of which are based on our running example of a bank enterprise. Important theoretical results are covered, but formal proofs are omitted. In place of proofs, figures and examples are used to suggest why a result is true. Formal descriptions and proofs of theoretical results may be found in research papers and advanced texts that are referenced in the bibliographical notes.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial or experimental database systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular database system. Details of particular database systems are discussed in Part 9, "Case Studies."

In this, the fifth edition of *Database System Concepts*, we have retained the overall style of the prior editions while evolving the content and organization to reflect the changes that are occurring in the way databases are designed, managed, and used. We have also taken into account trends in the teaching of database concepts and made adaptations to facilitate these trends where appropriate. Before we describe the content of the book in detail, we highlight some of the features of the fifth edition.

- **Earlier coverage of SQL.** Many instructors use SQL as a key component of term projects (see our Web site, www.db-book.com, for sample projects). In order to give students ample time for the projects, particularly for universities and colleges on the quarter system, it is essential to teach SQL as early as possible. With this in mind, we have undertaken several changes in organization:
  1. Deferring the presentation of the entity-relationship model to Part 2, entitled "Database Design."
  2. Streamlining the introduction of the relational model by deferring coverage of the relational calculus to Chapter 5, while retaining coverage of the relational algebra in Chapter 2.
  3. Devoting two early chapters to SQL. Chapter 3 covers basic SQL features including data definition and manipulation. Chapter 4 covers more advanced features, including integrity constraints, dynamic SQL, and procedural constructs. New material in this chapter includes expanded coverage of JDBC, procedural constructs in SQL, recursion in SQL, and new features from SQL:2003. The chapter also includes a short overview of authorization; detailed coverage of authorization is deferred to Chapter 8.

  These changes allow students to begin writing SQL queries early in the course, and gain familiarity with the use of database systems. This also allows students to develop an intuition about database design that facilitates the teaching of design methodology in Part 2 of the text. We have found that students appreciate database-design issues better with this organization.

- **A new part (Part 2) that is devoted to database design.** Part 2 of the text contains three chapters devoted to the design of databases and database applications. We include here a chapter (Chapter 6) on the entity-relationship model that includes all of the material from the corresponding chapter of the fourth edition (Chapter 2), plus several significant updates. We also present in Chapter 6 a brief overview of the process of database design. Instructors who prefer to begin their course with the E-R model can begin with this chapter without loss of continuity, as we have strived to avoid dependencies on any prior chapter other than Chapter 1.

  Chapter 7, on relational design, presents the material covered in Chapter 7 of the fourth edition, but does so in a new, more readable style. Design concepts from the E-R model are used to build an intuitive overview of relational design issues, in advance of the presentation of the formal approach to design using functional and multivalued dependencies and algorithmic normalization. This chapter also includes a new section on temporal issues in database design.

  Part 2 concludes with a new chapter, Chapter 8, that describes the design and development of database applications, including Web applications, servlets, JSP, triggers, and security issues. In keeping with the increased need to secure software from attacks, coverage of security has been significantly expanded from the fourth edition.

- **Thoroughly revised and updated coverage of object-based databases and XML.** Part 3 includes a heavily revised chapter on object-based databases that emphasizes SQL object-relational features, replacing the separate chapters on object-oriented and object-relational databases from the fourth edition. Some of the introductory material on object-orientation which students are familiar with from earlier courses has been removed, as have syntactic details of the now defunct ODMG standard. However, important concepts underlying object-oriented databases have been retained, including new material on the JDO standard for adding persistence to Java.

  Part 3 includes also a chapter on the design and querying of XML data, which is significantly revised from the corresponding chapter in the fourth edition. It includes enhanced coverage of XML Schema and XQuery, coverage of the SQL/XML standard, and more examples of XML applications including Web services.

- **Reorganized material on data mining and information retrieval.** Data mining and online analytic processing are now centrally important uses of databases—no longer only "advanced topics." We have, therefore, moved our coverage of these topics into a new part, Part 6, containing a chapter on data mining and analysis along with a chapter on information retrieval.

- **New case study covering PostgreSQL.** PostgreSQL is an open-source database system that has gained enormous popularity in the past few years. In addition to being a platform on which to build database applications, the source code can be studied and extended in courses that emphasize database internals. A case study of PostgreSQL is therefore added to Part 9, where it joins three case studies that appeared in the fourth edition (Oracle, IBM DB2, and Microsoft SQL Server). The latter three case studies have been updated to reflect the latest versions of the respective software.

The coverage of topics not listed above, including transaction processing (concurrency and recovery), storage structures, query processing, and distributed and parallel databases are all updated from their fourth-edition counterparts, though their overall organization is relatively unchanged. The coverage of QBE in Chapter 5 has been revised, removing syntactic details of aggregation and updates that do not correspond to any actual implementation, while retaining the key concepts behind QBE.

Organization

The text is organized in nine major parts, plus three appendices.

- **Overview** (Chapter 1). Chapter 1 provides a general overview of the nature and purpose of database systems. We explain how the concept of a database system has developed, what the common features of database systems are, what a database system does for the user, and how a database system interfaces with operating systems. We also introduce an example database application: a banking enterprise consisting of multiple bank branches. This example

is used as a running example throughout the book. This chapter is motivational, historical, and explanatory in nature.

- **Part 1: Relational Databases** (Chapters 2 through 5). Chapter 2 introduces the relational model of data, covering basic concepts as well as the relational algebra. The chapter also provides a brief introduction to integrity constraints. Chapters 3 and 4 focus on the most influential of the user-oriented relational languages: SQL. While Chapter 3 provides a basic introduction to SQL, Chapter 4 describes more advanced features of SQL, including how to interface between a programming language and a database supporting SQL. Chapter 5 covers other relational languages, including the relational calculus, QBE, and Datalog.

  The chapters in this part describe data manipulation: queries, updates, insertions, and deletions, assuming a schema design has been provided. Schema design issues are deferred to Part 2.

- **Part 2: Database Design** (Chapters 6 through 8). Chapter 6 provides an overview of the database-design process, with major emphasis on database design using the entity-relationship data model. The entity-relationship data model provides a high-level view of the issues in database design, and of the problems that we encounter in capturing the semantics of realistic applications within the constraints of a data model. UML class-diagram notation is also covered in this chapter.

  Chapter 7 introduces the theory of relational database design. The theory of functional dependencies and normalization is covered, with emphasis on the motivation and intuitive understanding of each normal form. This chapter begins with an overview of relational design and relies on an intuitive understanding of logical implication of functional dependencies. This allows the concept of normalization to be introduced prior to full coverage of functional-dependency theory, which is presented later in the chapter. Instructors may chose to use only this initial coverage in Sections 7.1 through 7.3 without loss of continuity. Instructors covering the entire chapter will benefit from students having a good understanding of normalization concepts to motivate some of the challenging concepts of functional-dependency theory.

  Chapter 8 covers application design and development. This chapter emphasizes the construction of database applications with Web-based interfaces. In addition, the chapter covers application security.

- **Part 3: Object-Based Databases and XML** (Chapters 9 and 10). Chapter 9 covers object-based databases. The chapter describes the object-relational data model, which extends the relational data model to support complex data types, type inheritance, and object identity. The chapter also describes database access from object-oriented programming languages.

  Chapter 10 covers the XML standard for data representation, which is seeing increasing use in the exchange and storage of complex data. The chapter also describes query languages for XML.

- **Part 4: Data Storage and Querying** (Chapters 11 through 14). Chapter 11 deals with disk, file, and file-system structure. A variety of data-access techniques are presented in Chapter 12, including hashing and B^+-tree indices. Chapters 13 and 14 address query-evaluation algorithms and query optimization. These chapters provide an understanding of the internals of the storage and retrieval components of a database.

- **Part 5: Transaction Management** (Chapters 15 through 17). Chapter 15 focuses on the fundamentals of a transaction-processing system, including transaction atomicity, consistency, isolation, and durability, as well as the notion of serializability.

  Chapter 16 focuses on concurrency control and presents several techniques for ensuring serializability, including locking, timestamping, and optimistic (validation) techniques. The chapter also covers deadlock issues.

  Chapter 17 covers the primary techniques for ensuring correct transaction execution despite system crashes and disk failures. These techniques include logs, checkpoints, and database dumps.

- **Part 6: Data Mining and Information Retrieval** (Chapters 18 and 19). Chapter 18 introduces the concept of a data warehouse and explains data mining and online analytical processing (OLAP), including SQL:1999 support for OLAP and data warehousing. Chapter 19 describes information-retrieval techniques for querying textual data, including hyperlink-based techniques used in Web search engines.

  Part 6 uses the modeling and language concepts from Parts 1 and 2, but does not depend on Parts 3, 4, or 5. It can therefore be incorporated easily into a course that focuses on SQL and on database design.

- **Part 7: Database-System Architecture** (Chapters 20 through 22). Chapter 20 covers computer-system architecture, and describes the influence of the underlying computer system on the database system. We discuss centralized systems, client–server systems, parallel and distributed architectures, and network types in this chapter.

  Chapter 21, on parallel databases, explores a variety of parallelization techniques, including I/O parallelism, interquery and intraquery parallelism, and interoperation and intraoperation parallelism. The chapter also describes parallel-system design.

  Chapter 22 covers distributed database systems, revisiting the issues of database design, transaction management, and query evaluation and optimization, in the context of distributed databases. The chapter also covers issues of system availability during failures and describes the LDAP directory system.

- **Part 8: Other Topics** (Chapters 23 through 25). Chapter 23 covers performance benchmarks, performance tuning, standardization and application migration from legacy systems.

Chapter 24 covers advanced data types and new applications, including temporal data, spatial and geographic data, multimedia data, and issues in the management of mobile and personal databases.

Finally, Chapter 25 deals with advanced transaction processing. Topics covered include transaction-processing monitors, transactional workflows, electronic commerce, high-performance transaction systems, real-time transaction systems, long duration transactions, and transaction management in multidatabase systems.

- **Part 9: Case Studies** (Chapters 26 through 29). In this part we present case studies of four leading database systems, including PostgreSQL, Oracle, IBM DB2, and Microsoft SQL Server. These chapters outline unique features of each of these systems, and describe their internal structure. They provide a wealth of interesting information about the respective products, and help you see how the various implementation techniques described in earlier parts are used in real systems. They also cover several interesting practical aspects in the design of real systems.

- **Online Appendices**. Although most new database applications use either the relational model or the object-relational model, the network and hierarchical data models are still in use in some legacy applications. For the benefit of readers who wish to learn about these data models, we provide appendices describing the network and hierarchical data models, in Appendices A and B respectively; the appendices are available only online (http://www.db-book.com).

  Appendix C describes advanced relational database design, including the theory of multivalued dependencies, join dependencies, and the project-join and domain-key normal forms. This appendix is for the benefit of individuals who wish to study the theory of relational database design in more detail, and instructors who wish to do so in their courses. This appendix, too, is available only online, on the Web page of the book.

The Fifth Edition

The production of this fifth edition has been guided by the many comments and suggestions we received concerning the earlier editions, by our own observations while teaching at Yale University, Lehigh University, and IIT Bombay, and by our analysis of the directions in which database technology is evolving.

Our basic procedure was to rewrite the material in each chapter, bringing the older material up-to-date, adding discussions on recent developments in database technology, and improving descriptions of topics that students found difficult to understand. As in the fourth edition, each chapter has a list of review terms that can help readers review key topics covered in the chapter. Most chapters also have a tools section at the end of the chapter that provides information on software tools related to the topic of the chapter. We have also added new exercises and updated references.

In the fifth edition, we have divided the exercises into two sets: **practice exercises** and **exercises**. The solutions for the practice exercises are publicly available on the Web page of the book. Students are encouraged to solve the practice exercises on their own, and later use the solutions on the Web page to check their own solutions. Solutions to the other exercises are available only to instructors (see "Instructor's Note," below, for information on how to get the solutions).

Instructor's Note

The book contains both basic and advanced material, which might not be covered in a single semester. We have marked several sections as advanced, using the symbol $**$. These sections may be omitted if so desired, without a loss of continuity. Exercises that are difficult (and can be omitted) are also marked using the symbol "$**$."

It is possible to design courses by using various subsets of the chapters. We outline some of the possibilities here:

- Sections of Chapter 4 from Section 4.6 onward may be omitted from an introductory course.
- Chapter 5 can be omitted if students will not be using relational calculus, QBE or Datalog as part of the course.
- Chapters 9 (Object-Based Databases), 10 (XML), and 14 (Query Optimization) can be omitted from an introductory course.
- Both our coverage of transaction processing (Chapters 15 through 17) and our coverage of database-system architecture (Chapters 20 through 22) consist of an overview chapter (Chapters 15 and 20, respectively), followed by chapters with details. You might choose to use Chapters 15 and 20, while omitting Chapters 16, 17, 21, and 22, if you defer these latter chapters to an advanced course.
- Chapters 18 and 19, covering data mining and information retrieval, can be used as self-study material or omitted from an introductory course.
- Chapters 23 through 25 are suitable for an advanced course or for self-study by students.
- The case-study Chapters 26 through 29 are suitable for self-study by students.

Model course syllabi, based on the text, can be found on the Web home page of the book (see the following section).

Web Page and Teaching Supplements

A Web home page for the book is available at the URL:

http://www.db-book.com

The Web page contains:

- Slides covering all the chapters of the book
- Answers to the practice exercises
- Laboratory material
- The three appendices
- An up-to-date errata list
- Supplementary material contributed by users of the book

The following additional material is available only to faculty:

- An instructor manual containing solutions to all exercises in the book
- A question bank containing extra exercises

For more information about how to get a copy of the instructor manual and the question bank, please send electronic mail to customer.service@mcgraw-hill.com. In the United States, you may call 800-338-3987. The McGraw-Hill Web page for this book is

http://www.mhhe.com/silberschatz

Contacting Us and Other Users

We have endeavored to eliminate typos, bugs, and the like from the text. But, as in new releases of software, bugs probably remain; an up-to-date errata list is accessible from the book's home page. We would appreciate it if you would notify us of any errors or omissions in the book that are not on the current list of errata.

We would be glad to receive suggestions on improvements to the books. We also welcome any contributions to the book Web page that could be of use to other readers, such as programming exercises, project suggestions, online labs and tutorials, and teaching tips.

Email should be addressed to db-book@cs.yale.edu. Any other correspondence should be sent to Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT 06520-8285 USA.

We also provide a mailing list through which users of our book can communicate among themselves and with us, and receive updates on the book and other related information. The list is moderated, so you will not receive junk mail on the list. Please follow the mailing list link from the book's home page to subscribe to the mailing list.

Acknowledgments

This edition has benefited from the many useful comments provided to us by the numerous students who have used the prior four editions. In addition, many people

have written or spoken to us about the book, and have offered suggestions and comments. Although we cannot mention all these people here, we especially thank the following:

- Hani Abu-Salem, DePaul University; Jamel R. Alsabbagh, Grand Valley State University; Ramzi Bualuan, Notre Dame University; Zhengxin Chen, University of Nebraska at Omaha; Jan Chomick, SUNY Buffalo University; Qin Ding, Penn State University at Harrisburg; Frantisek Franek, McMaster University; Shashi K. Gadia, Iowa State University; William Hankley, Kansas State University; Randy M. Kaplan, Drexel University; Mark Llewellyn, University of Central Florida; Marty Maskarinec, Western Illinois University; Yiu-Kai Dennis Ng, Brigham Young University; Sunil Prabhakar, Purdue University; Stewart Shen, Old Dominion University; Anita Whitehall, Foothill College; Christopher Wilson, University of Oregon; Weining Zhang, University of Texas at San Antonio; who served as reviewers of the book and whose comments helped us greatly in formulating this fifth edition.
- Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, and Bianca Schroeder (Carnegie Mellon University) for writing the appendix describing the PostgreSQL database system.
- Hakan Jakobsson (Oracle), for the appendix on the Oracle database system,
- Sriram Padmanabhan (IBM), for the appendix describing the IBM DB2 database system.
- Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas and Michael Zwilling (all of Microsoft) for the appendix on the Microsoft SQL Server database system. José Blakeley also for coordinating and editing Chapter 29, and César Galindo-Legaria, Goetz Graefe, Kalen Delaney, and Thomas Casey (all of Microsoft) for their contributions to the previous edition of the Microsoft SQL Server chapter.
- Chen Li and Sharad Mehrotra for providing material on JDBC and security that helped update and extend Chapter 8.
- Valentin Dinu, Goetz Graefe, Bruce Hillyer, Chad Hogg, Nahid Rahman, Patrick Schmid, Jeff Storey, Prem Thomas, Liu Zhenming, and particularly N. L. Sarda for their feedback, which helped us prepare the fifth edition.
- Rami Khouri, Nahid Rahman, and Michael Rys for feedback on draft versions of chapters from the fifth edition.
- Raj Ashar, Janek Bogucki, Gavin M. Bierman, Christian Breimann, Tom Chappell, Y. C. Chin, Laurens Damen, Prasanna Dhandapani, Arvind Hulgeri, Zheng Jiaping, Graham J. L. Kemp, Hae Choon Lee, Sang-Won Lee, Thanh-Duy Nguyen, D. B. Phatak, Juan Altmayer Pizzorno, Rajarshi Rakshit, Greg Riccardi, N. L. Sarda, Max Smolens, Nikhil Sethi, and Tim Wahls for pointing out errors in the fourth edition.

- Marilyn Turnamian, whose excellent secretarial assistance was essential for timely completion of this fifth edition.

The publisher was Betsy Jones. The sponsoring editor was Kelly Lowery. The developmental editor was Melinda D. Bilecki. The project manager was Peggy Selle. The executive marketing manager was Michael Weitz. The marketing manager was Dawn Bercier. The cover illustrator and cover designer was JoAnne Schopler. The freelance copyeditor was George Watson. The freelance proofreader was Judy Gantenbein. The designer was Laurie Janssen The freelance indexer was Tobiah Waldron.

This edition is based on the four previous editions, so we thank once again the many people who helped us with the first four editions, including R. B. Abhyankar, Don Batory, Phil Bernhard, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Michael Carey, Soumen Chakrabarti, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Shashi Gadia, Jim Gray, Le Gruenwald, Eitan M. Gurari, Ron Hitchens, Yannis Ioannidis, Hyoung-Joo Kim, Won Kim, Henry Korth (father of Henry F.), Carol Kroll, Gary Lindstrom, Irwin Levinstein, Ling Liu, Dave Maier, Keith Marzullo, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Hector Garcia-Molina, Ami Motro, Bhagirath Narahari, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, Bruce Porter, Jim Peterson, K. V. Raghavan, Krithi Ramamritham, Mike Reiter, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Sunita Sarawagi, N. L. Sarda, S. Seshadri, Shashi Shekhar, Amit Sheth, Nandit Soparkar, Greg Speegle, Dilys Thomas, and Marianne Winslett.

Marilyn Turnamian and Nandprasad Joshi provided secretarial assistance for the fourth edition, and Marilyn also prepared an early draft of the cover design for the fourth edition. Lyn Dupré copyedited the third edition and Sara Strandtman edited the text of the third edition. Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K. V. Raghavan, Prateek Kapadia, Sara Strandtman, Greg Speegle, and Dawn Bezviner helped to prepare the instructor's manual for earlier editions. The new cover is an evolution of the covers of the first four editions. The idea of using ships as part of the cover concept was originally suggested to us by Bruce Stephan.

Finally, Sudarshan would like to acknowledge his wife, Sita, for her love and support and son Madhur for his love. Hank would like to acknowledge his wife, Joan, and his children, Abby and Joe, for their love and understanding. Avi would like to acknowledge Valerie for her love, patience, and support during the revision of this book.

A. S.
H. F. K.
S. S.

CHAPTER 1

Introduction

Solutions to Practice Exercises

1.1 Two disadvantages associated with database systems are listed below.

a. Setup of the database system requires more knowledge, money, skills, and time.

b. The complexity of the database may result in poor performance.

1.2 Programming language classification:

- Procedural: C, C++, Java, Basic, Fortran, Cobol, Pascal
- Non-procedural: Lisp and Prolog

Note: Lisp and Prolog support some procedural constructs, but the core of both these languages is non-procedural.

In theory, non-procedural languages are easier to learn, because they let the programmer concentrate on *what* needs to be done, rather than *how* to do it. This is not always true in practice, especially if procedural languages are learned first.

1.3 Six major steps in setting up a database for a particular enterprise are:

- Define the high level requirements of the enterprise (this step generates a document known as the system requirements specification.)
- Define a model containing all appropriate types of data and data relationships.
- Define the integrity constraints on the data.
- Define the physical level.
- For each known problem to be solved on a regular basis (e.g., tasks to be carried out by clerks or Web users) define a user interface to carry out the task, and write the necessary application programs to implement the user interface.

- Create/initialize the database.

1.4 Let *tgrid* be a two-dimensional integer array of size $n \times m$.

a.
- The physical level would simply be $m \times n$ (probably consecutive) storage locations of whatever size is specified by the implementation (e.g., 32 bits each).
- The conceptual level is a grid of boxes, each possibly containing an integer, which is *n* boxes high by *m* boxes wide.
- There are $2^{m \times n}$ possible views. For example, a view might be the entire array, or particular row of the array, or all *n* rows but only columns 1 through *i*.

b.
- Consider the following Pascal declarations:

  type *tgrid* = **array**[1..*n*, 1..*m*] **of integer**;
  var *vgrid1*, *vgrid2* : *tgrid*

  Then *tgrid* is a schema, whereas the value of variables *vgrid1* and *vgrid2* are instances.
- To illustrate further, consider the schema **array**[1..2, 1..2] **of integer**. Two instances of this scheme are:

1	16
7	89

17	90
412	8

CHAPTER 2

Relational Model

Solutions to Practice Exercises

2.1 **a.** $\Pi_{person\_name}\ ((employee\ \bowtie\ manages)$
$\bowtie_{(manager\_name = employee2.person\_name \wedge employee.street = employee2.street}$
$\wedge\ employee.city = employee2.city)(\rho_{employee2}\ (employee)))$

b. The following solutions assume that all people work for exactly one company. If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, the problem is more complicated. We give solutions for this more realistic case later.

$\Pi_{person\_name}\ (\sigma_{company\_name \neq}$ "First Bank Corporation"$(works))$

If people may not work for any company:

$\Pi_{person\_name}(employee)\ -\ \Pi_{person\_name}$
$(\sigma_{(company\_name =}$ "First Bank Corporation"$_)(works))$

c. $\Pi_{person\_name}\ (works)\ -\ (\Pi_{works.person\_name}\ (works$
$\bowtie_{(works.salary \leq works2.salary \wedge works2.company\_name =}$ "Small Bank Corporation"$_)$
$\rho_{works2}(works)))$

2.2 **a.** The left outer theta join of $r(R)$ and $s(S)$ $(r\ \mathbin{⟕}_\theta\ s)$ can be defined as
$(r \bowtie_\theta s) \cup ((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The tuple of nulls is of size equal to the number of attributes in S.

b. The right outer theta join of $r(R)$ and $s(S)$ $(r\ ⟖_\theta\ s)$ can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s)))$
The tuple of nulls is of size equal to the number of attributes in R.

c. The full outer theta join of $r(R)$ and $s(S)$ $(r \mathbin{⟗}_\theta s)$ can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s))) \cup$
$((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The first tuple of nulls is of size equal to the number of attributes in R, and the second one is of size equal to the number of attributes in S.

2.3 **a.**
$$employee \leftarrow \Pi_{person\_name, street, ''Newtown''}(\sigma_{person\_name=\text{“Jones”}}(employee)) \cup (employee - \sigma_{person\_name=\text{“Jones”}}(employee))$$

b. The update syntax allows reference to a single relation only. Since this update requires access to both the relation to be updated (*works*) and the *manages* relation, we must use several steps. First we identify the tuples of *works* to be updated and store them in a temporary relation (t_1). Then we create a temporary relation containing the new tuples (t_2). Finally, we delete the tuples in t_1, from *works* and insert the tuples of t_2.

$$t_1 \leftarrow \Pi_{works.person\_name, company\_name, salary}(\sigma_{works.person\_name = manager\_name}(works \times manages))$$
$$t_2 \leftarrow \Pi_{person\_name, company\_name, 1.1 * salary}(t_1)$$
$$works \leftarrow (works - t_1) \cup t_2$$

CHAPTER 3

SQL

Solutions to Practice Exercises

3.1 Note: The *participated* relation relates drivers, cars, and accidents.

a. Note: this is not the same as the total number of accidents in 1989. We must count people with several accidents only once.

```
select   count (distinct name)
from     accident, participated, person
where    accident.report_number = participated.report_number
and      participated.driver_id = person.driver_id
and      date between date '1989-00-00' and date '1989-12-31'
```

b. We assume the driver was "Jones," although it could be someone else. Also, we assume "Jones" owns one Toyota. First we must find the license of the given car. Then the *participated* and *accident* relations must be updated in order to both record the accident and tie it to the given car. We assume values "Berkeley" for *location*, '2001-09-01' for date and *date*, 4007 for *report_number* and 3000 for damage amount.

```
insert into accident
        values (4007, '2001-09-01', 'Berkeley')

insert into participated
        select o.driver_id, c.license, 4007, 3000
        from person p, owns o, car c
        where p.name = 'Jones' and p.driver_id = o.driver_id and
                o.license = c.license and c.model = 'Toyota'
```

c. Since *model* is not a key of the *car* relation, we can either assume that only one of John Smith's cars is a Mazda, or delete all of John Smith's Mazdas (the query is the same). Again assume *name* is a key for *person*.

```
delete car
where model = 'Mazda' and license in
    (select license
     from person p, owns o
     where p.name = 'John Smith' and p.driver_id = o.driver_id)
```

Note: The *owns*, *accident* and *participated* records associated with the Mazda still exist.

3.2 **a.** Query:

```
select e.employee_name, city
from employee e, works w
where w.company_name = 'First Bank Corporation' and
        w.employee_name = e.employee_name
```

b. If people may work for several companies, the following solution will only list those who earn more than $10,000 per annum from "First Bank Corporation" alone.

```
select *
from employee
where employee_name in
    (select employee_name
     from works
     where company_name = 'First Bank Corporation' and salary ¿ 10000)
```

As in the solution to the previous query, we can use a join to solve this one also.

c. The following solution assumes that all people work for exactly one company.

```
select employee_name
from works
where company_name ≠ 'First Bank Corporation'
```

If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, or if people may have jobs with more than one company, the solution is slightly more complicated.

```
select employee_name
from employee
where employee_name not in
    (select employee_name
     from works
     where company_name = 'First Bank Corporation')
```

d. The following solution assumes that all people work for at most one company.

```
select employee_name
from works
where salary > all
    (select salary
     from works
     where company_name = 'Small Bank Corporation')
```

If people may work for several companies and we wish to consider the *total* earnings of each person, the problem is more complex. It can be solved by using a nested subquery, but we illustrate below how to solve it using the **with** clause.

```
with emp_total_salary as
   (select employee_name, sum(salary) as total_salary
    from works
    group by employee_name
   )
select employee_name
from emp_total_salary
where total_salary > all
   (select total_salary
    from emp_total_salary, works
    where works.company_name = 'Small Bank Corporation' and
          emp_total_salary.employee_name = works.employee_name
   )
```

e. The simplest solution uses the **contains** comparison which was included in the original System R Sequel language but is not present in the subsequent SQL versions.

```
select T.company_name
from company T
where (select R.city
       from company R
       where R.company_name = T.company_name)
     contains
       (select S.city
        from company S
        where S.company_name = 'Small Bank Corporation')
```

Below is a solution using standard SQL.

```
select S.company_name
from company S
where not exists ((select city
                   from company
                   where company_name = 'Small Bank Corporation')
                  except
                   (select city
                   from company T
                   where S.company_name = T.company_name))
```

f. Query:

```
select company_name
from works
group by company_name
having count (distinct employee_name) >= all
    (select count (distinct employee_name)
     from works
     group by company_name)
```

g. Query:

```
select company_name
from works
group by company_name
having avg (salary) > (select avg (salary)
                       from works
                       where company_name = 'First Bank Corporation')
```

3.3 **a.** The solution assumes that each person has only one tuple in the *employee* relation.

```
update employee
set city = 'Newton'
where person_name = 'Jones'
```

b. Query:

```
update works T
set T.salary = T.salary * 1.03
where T.employee_name in (select manager_name
                          from manages)
      and T.salary * 1.1 > 100000
      and T.company_name = 'First Bank Corporation'

update works T
set T.salary = T.salary * 1.1
where T.employee_name in (select manager_name
                          from manages)
      and T.salary * 1.1 <= 100000
      and T.company_name = 'First Bank Corporation'
```

SQL-92 provides a **case** operation (see Exercise 3.5), using which we give a more concise solution:

```
update works T
set T.salary = T.salary *
    (case
        when (T.salary * 1.1 > 100000) then 1.03
        else 1.1
    )
where T.employee_name in (select manager_name
                          from manages) and
      T.company_name = 'First Bank Corporation'
```

3.4 Query:

```
select coalesce(a.name, b.name) as name,
       coalesce(a.address, b.address) as address,
       a.title,
       b.salary
from  a full outer join b on a.name = b.name and
                             a.address = b.address
```

3.5 We use the **case** operation provided by SQL-92:

a. To display the grade for each student:

```
select student_id,
       (case
                when score < 40 then 'F',
                when score < 60 then 'C',
                when score < 80 then 'B',
                else 'A'
        end) as grade
from marks
```

b. To find the number of students with each grade we use the following query, where *grades* is the result of the query given as the solution to part 0.a.

select *grade*, **count**(*student_id*)
from *grades*
group by *grade*

3.6 The query selects those values of *p.a1* that are equal to some value of *r1.a1* or *r2.a1* if and only if both *r1* and *r2* are non-empty. If one or both of *r1* and *r2* are empty, the cartesian product of *p*, *r1* and *r2* is empty, hence the result of the query is empty. Of course if *p* itself is empty, the result is as expected, i.e. empty.

3.7 To insert the tuple ("Johnson", 1900) into the view *loan_info*, we can do the following:

$borrower \leftarrow (\text{"Johnson"}, \perp_k) \cup borrower$

$loan \leftarrow (\perp_k, \perp, 1900) \cup loan$

such that $\perp_k$ is a new marked null not already existing in the database.

CHAPTER 4

Advanced SQL

Solutions to Practice Exercises

4.1 Query:

```
create table loan
    (loan_number     char(10),
     branch_name     char(15),
     amount          integer,
     primary key (loan_number),
     foreign key (branch_name) references branch)


create table borrower
    (customer_name  char(20),
     loan_number    char(10),
     primary key (customer_name, loan_number),
     foreign key (customer_name) references customer,
     foreign key (loan_number) references loan)
```

Declaring the pair *customer_name, loan_number* of relation *borrower* as primary key ensures that the relation does not contain duplicates.

4.2 Query:

```
create table employee
    (person_name    char(20),
     street         char(30),
     city           char(30),
     primary key (person_name) )
```

```
create table works
    (person_name    char(20),
     company_name   char(15),
     salary         integer,
     primary key (person_name),
     foreign key (person_name) references employee,
     foreign key (company_name) references company)
```

```
create table company
    (company_name char(15),
     city         char(30),
     primary key (company_name))
```

```
ppcreate table manages
    (person_name    char(20),
     manager_name   char(20),
     primary key (person_name),
     foreign key (person_name) references employee,
     foreign key (manager_name) references employee)
```

Note that alternative datatypes are possible. Other choices for **not null** attributes may be acceptable.

a. check condition for the *works* table:

```
check((employee_name, company_name) in
        (select  e.employee_name, c.company_name
         from    employee e, company c
         where   e.city = c.city
        )
)
```

b. check condition for the *works* table:

```
check(
        salary < all
                (select   manager_salary
                 from     (select manager_name, manages.employee_name as emp_name,
                                  salary as manager_salary
                           from works, manages
                           where works.employee_name = manages.manager_name)
                 where    employee_name = emp_name
                )
)
```

The solution is slightly complicated because of the fact that inside the **select** expression's scope, the outer *works* relation into which the insertion is being performed is inaccessible. Hence the renaming of the *employee_name* attribute to *emp_name*. Under these circumstances, it is more natural to use assertions.

4.3 The tuples of all employees of the manager, at all levels, get deleted as well! This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second level employee tuples, and so on, till all direct and indirect employee tuples are deleted.

4.4 The assertion_name is arbitrary. We have chosen the name *perry*. Note that since the assertion applies only to the Perryridge branch we must restrict attention to only the Perryridge tuple of the *branch* relation rather than writing a constraint on the entire relation.

```
create assertion perry check
   (not exists (select *
                from branch
                where branch_name = 'Perryridge' and
                      assets ≠ (select sum (amount)
                                from loan
                                where branch_name = 'Perryridge')))
```

4.5 Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However not all kinds of queries can be written in SQL. Also nondeclarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.

Embedded SQL has the advantage of programs being less complicated since it avoids the clutter of the ODBC or JDBC function calls, but requires a specialized preprocessor.

CHAPTER 5

Other Relational Languages

Solutions to Practice Exercises

5.1 **a.** $\{t \mid \exists\, q \in r\,(q[A] = t[A])\}$

b. $\{t \mid t \in r \wedge t[B] = 17\}$

c. $\{t \mid \exists p \in r\, \exists q \in s\,(t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = q[D] \wedge t[E] = q[E] \wedge t[F] = q[F])\}$

d. $\{t \mid \exists p \in r\, \exists q \in s\,(t[A] = p[A] \wedge t[F] = q[F] \wedge p[C] = q[D]\}$

5.2 **a.** $\{< t > \mid \exists p,\ q\,(< t, p, q > \in r_1)\}$

b. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge b = 17\}$

c. $\{< a, b, c > \mid < a, b, c > \in r_1 \vee < a, b, c > \in r_2\}$

d. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge < a, b, c > \in r_2\}$

e. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge < a, b, c > \notin r_2\}$

f. $\{< a, b, c > \mid \exists p,\ q\,(< a, b, p > \in r_1 \wedge < q, b, c > \in r_2)\}$

5.3 **a.** $\{< a > \mid \exists\, b\,(< a, b > \in r \wedge b = 7)\}$

i.

r	*A*	*B*
	P.	*17*

ii. $query\,(X)$:- $r\,(X,\ 17)$

b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

i.

r	*A*	*B*
	_*a*	_*b*

s	*A*	*C*
	_*a*	_*c*

result	*A*	*B*	*C*
P.	_*a*	_*b*	_*c*

ii. *query(X, Y, Z) :- r(X, Y), s(X, Z)*

c. $\{< a > \mid \exists c\,(< a, c > \in s \wedge \exists b_1, b_2\,(< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

i.

r	*A*	*B*
	_*a* _*c*	>_*s* _*s*

s	*A*	*C*
	P._*a*	_*c*

ii. $query\,(X)$:- $s\,(X,\, Y),\ r\,(X,\, Z),\ r\,(Y,\, W),\ Z > W$

5.4 **a.** Query:

$query\,(X)$:- $p\,(X)$
$p\,(X)$:- $manages\,(X,$ "Jones"$)$
$p\,(X)$:- $manages\,(X,\, Y),\ p\,(Y)$

b. Query:

query(X, C) :- p(X), employee(X, S, C)
p(X) :- manages(X, "Jones")
p(X) :- manages(X, Y), p(Y)

c. Query:

query(X, Y) :- p(X, W), p(Y, W)
p(X, Y) :- manages(X, Y)
p(X, Y) :- manages(X, Z), p(Z, Y)

d. Query:

query(X, Y) :- *p(X, Y)*
p(X, Y) :- *manages(X, Z), manages(Y, Z)*
p(X, Y) :- *manages(X, V), manages(Y, W), p(V, W)*

5.5 A Datalog rule has two parts, the *head* and the *body*. The body is a comma separated list of *literals*. A *positive literal* has the form $p(t_1,\ t_2, \ldots,\ t_n\ )$ where p is the name of a relation with n attributes, and $t_1,\ t_2, \ldots,\ t_n$ are either constants or variables. A *negative literal* has the form $\neg p(t_1,\ t_2, \ldots,\ t_n)$ where p has n attributes. In the case of arithmetic literals, p will be an arithmetic operator like $>$, $=$ etc.

We consider only safe rules; see Section 5.4.4 for the definition of safety of Datalog rules. Further, we assume that every variable that occurs in an arithmetic literal also occurs in a positive non-arithmetic literal.

Consider first a rule without any negative literals. To express the rule as an extended relational-algebra view, we write it as a join of all the relations referred to in the (positive) non-arithmetic literals in the body, followed by a selection. The selection condition is a conjunction obtained as follows. If $p_1\,(X,\ Y),\ p_2\,(Y,\ Z)$ occur in the body, where p_1 is of the schema $(A,\ B)$ and p_2 is of the schema $(C,\ D)$, then $p_1.B\ =\ p_2.C$ should belong to the conjunction. The arithmetic literals can then be added to the condition.

As an example, the Datalog query

query(X, Y) :- *works(X, C, S1), works(Y, C, S2), S1 > S2, manages(X, Y)*

becomes the following relational-algebra expression:

$$E_1\ =\ \sigma_{(w1.company\_name = w2.company\_name\ \wedge\ w1.salary > w2.salary\ \wedge\ manages.person\_name = w1.person\_name\ \wedge\ manages.manager\_name = w2.person\_name)} (\rho_{w1}(works)\ \times\ \rho_{w2}(works)\ \times\ manages)$$

Now suppose the given rule has negative literals. First suppose that there are no constants in the negative literals; recall that all variables in a negative literal must also occur in a positive literal. Let $\neg q(X, Y)$ be the first negative literal, and let it be of the schema $(E,\ F)$. Let E_i be the relational algebra expression obtained after all positive and arithmetic literals have been handled. To handle this negative literal, we generate the expression

$$E_j\ \ =\ \ E_i \bowtie (\Pi_{A_1, A_2}(E_i) - q)$$

where A_1 and A_2 are the attribute names of two columns in E_i which correspond to X and Y respectively.

Now let us consider constants occurring in a negative literal. Consider a negative literal of the form $\neg q(a, b, Y)$ where a and b are constants. Then, in the above expression defining E_j we replace q by $\sigma_{A_1 = a \wedge A_2 = b}(q)$.

Proceeding in a similar fashion, the remaining negative literals are processed, finally resulting in an expression E_w.

Finally the desired attributes are projected out of the expression. The attributes in E_w corresponding to the variables in the head of the rule become the projection attributes.

Thus our example rule finally becomes the view:

create view *query* **as**

$$\Pi_{w1.person\_name,\, w2.person\_name}(E_2)$$

If there are multiple rules for the same predicate, the relational-algebra expression defining the view is the union of the expressions corresponding to the individual rules.

The above conversion can be extended to handle rules that satisfy some weaker forms of the safety conditions, and where some restricted cases where the variables in arithmetic predicates do not appear in a positive non-arithmetic literal.

CHAPTER 6

Database Design and the E-R Model

Solutions to Practice Exercises

6.1 See Figure 6.1

6.2 See Figure 6.2.
In the answer given here, the main entity sets are *student*, *course*, *course_offering*, and *instructor*. The entity set *course_offering* is a weak entity set dependent on *course*. The assumptions made are :

a. A class meets only at one particular place and time. This E-R diagram cannot model a class meeting at different places at different times.
b. There is no guarantee that the database does not have two classes meeting at the same place and time.

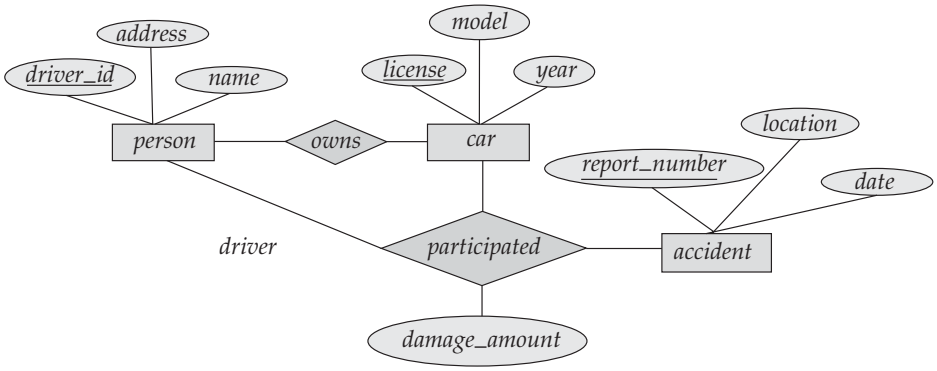


Figure 6.1 E-R diagram for a car insurance company.

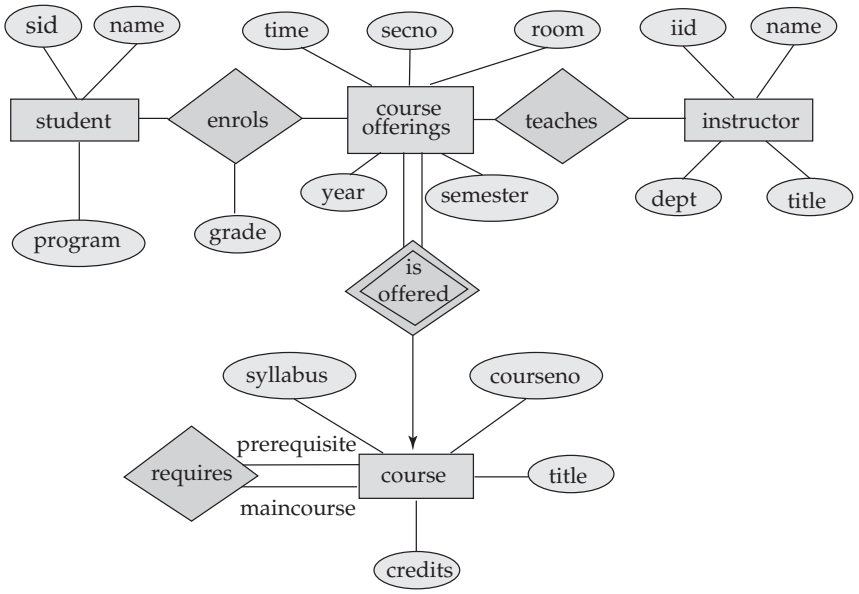


Figure 6.2 E-R diagram for a university.

6.3 **a.** See Figure 6.3
b. See Figure 6.4

6.4 See Figure 6.5

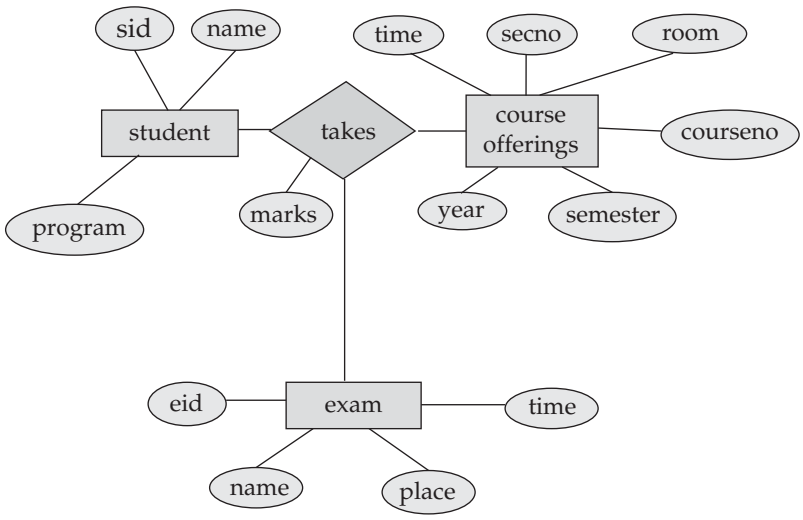


Figure 6.3 E-R diagram for marks database.

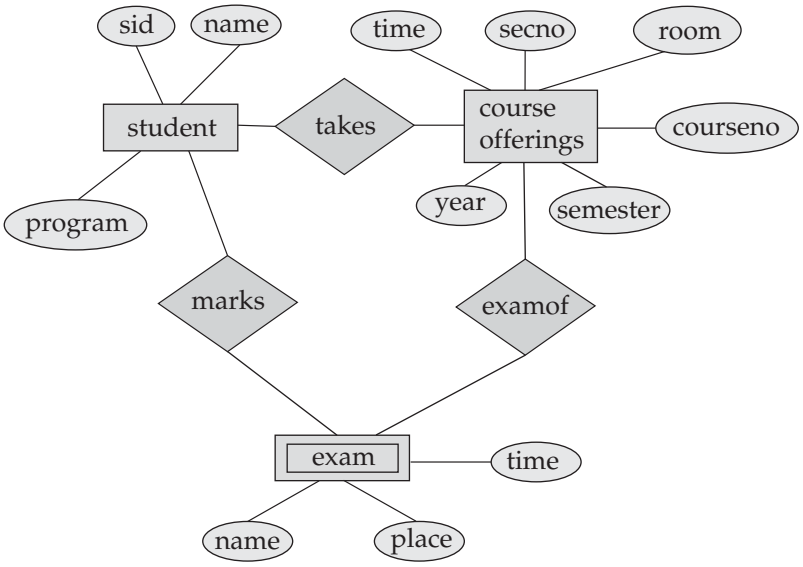


Figure 6.4 Another E-R diagram for marks database.

6.5 By using one entity set many times we are missing relationships in the model. For example, in the E-R diagram in Figure 6.6: the students taking classes are the same students who are athletes, but this model will not show that.

6.6 **a.** See Figure 6.7

b. The additional entity sets are useful if we wish to store their attributes as part of the database. For the *course* entity set, we have chosen to include three attributes. If only the primary key (*c_number*) were included, and if courses have only one section, then it would be appropriate to replace the *course* (and *section*) entity sets by an attribute (*c_number*) of *exam*. The reason it is undesirable to have multiple attributes of *course* as attributes of *exam* is that it would then be difficult to maintain data on the courses, particularly if a course has no exam or several exams. Similar remarks apply to the *room* entity set.

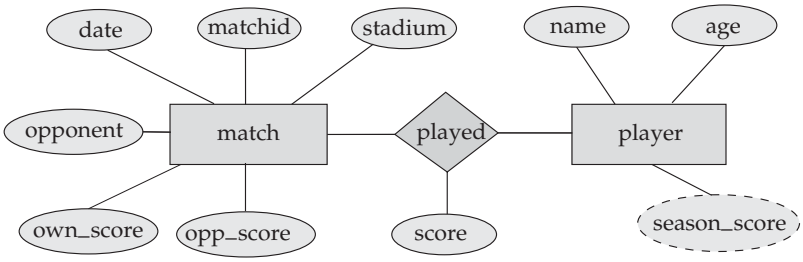


Figure 6.5 E-R diagram for favourite team statistics.

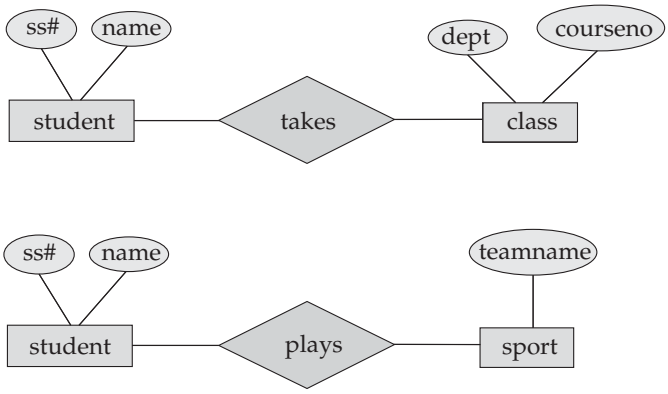


Figure 6.6 E-R diagram with entity duplication.

6.7 **a.** The criteria to use are intuitive design, accurate expression of the real-world concept and efficiency. A model which clearly outlines the objects and relationships in an intuitive manner is better than one which does not, because it is easier to use and easier to change. Deciding between an attribute and an entity set to represent an object, and deciding between an entity set and relationship set, influence the accuracy with which the real-world concept is expressed. If the right design choice is not made, inconsistency and/or loss of information will result. A model which can be implemented in an efficient manner is to be preferred for obvious reasons.

b. Consider three different alternatives for the problem in Exercise 6.2.

- See Figure 6.8
- See Figure 6.9
- See Figure 6.10

Each alternative has merits, depending on the intended use of the database. Scheme 6.8 has been seen earlier. Scheme 6.10 does not require a separate entity for *prerequisites*. However, it will be difficult to store all the prerequi-

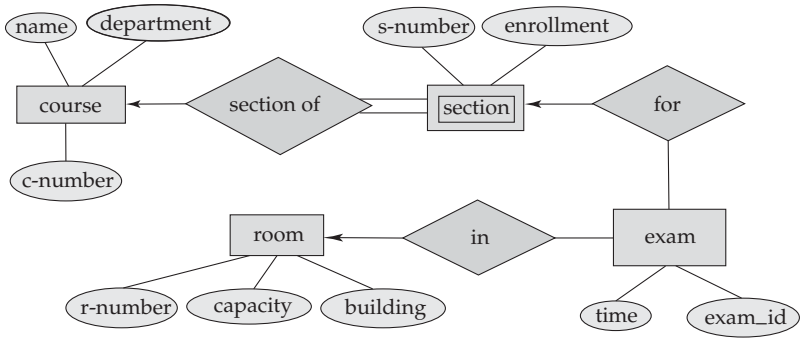


Figure 6.7 E-R diagram for exam scheduling.

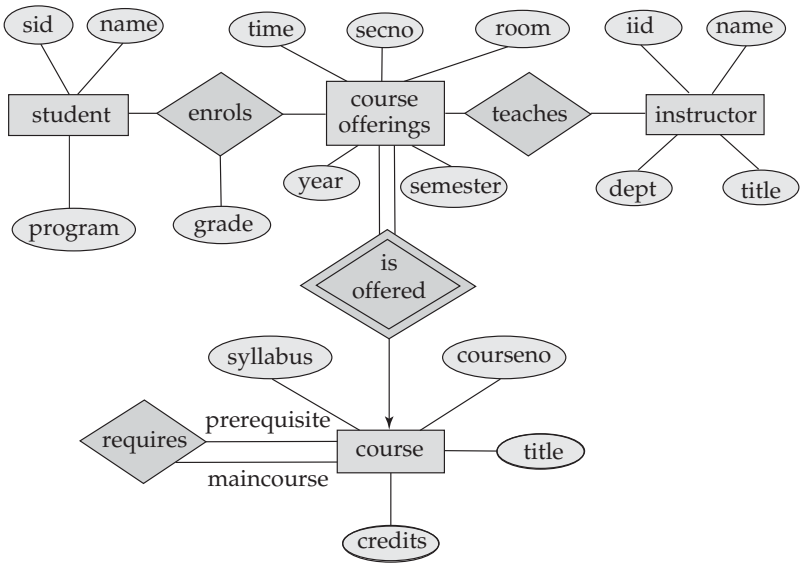


Figure 6.8 E-R diagram for University(a) .

sites(being a multi-valued attribute). Scheme 6.9 treats prerequisites as well as classrooms as separate entities, making it useful for gathering data about prerequisites and room usage. Scheme 6.8 is in between the others, in that it treats prerequisites as separate entities but not classrooms. Since a registrar's office probably has to answer general questions about the number of classes a student is taking or what are all the prerequisites of a course, or where a specific class meets, scheme 6.9 is probably the best choice.

6.8 **a.** If a pair of entity sets are connected by a path in an E-R diagram, the entity sets are related, though perhaps indirectly. A disconnected graph implies that there are pairs of entity sets that are unrelated to each other. If we split the graph into connected components, we have, in effect, a separate database corresponding to each connected component.

b. As indicated in the answer to the previous part, a path in the graph between a pair of entity sets indicates a (possibly indirect) relationship between the two entity sets. If there is a cycle in the graph then every pair of entity sets on the cycle are related to each other in at least two distinct ways. If the E-R diagram is acyclic then there is a unique path between every pair of entity sets and, thus, a unique relationship between every pair of entity sets.

6.9 **a.** Let $E = \{e_1, e_2\}$, $A = \{a_1, a_2\}$, $B = \{b_1\}$, $C = \{c_1\}$, $R_A = \{(e_1, a_1), (e_2, a_2)\}$, $R_B = \{(e_1, b_1)\}$, and $R_C = \{(e_1, c_1)\}$. We see that because of the tuple (e_2, a_2), no instance of R exists which corresponds to E, R_A, R_B and R_C.

b. See Figure 6.11. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B and C.

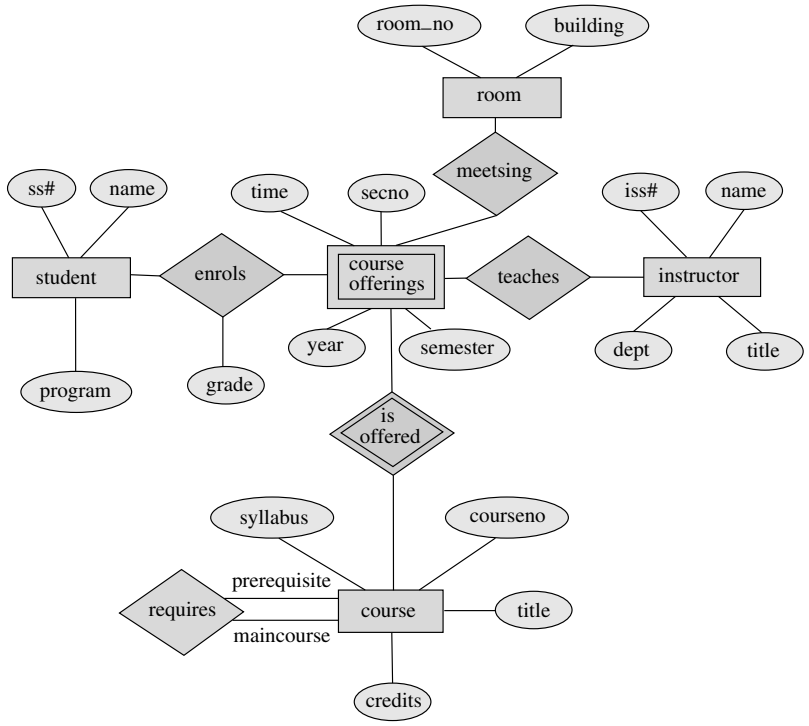


Figure 6.9 E-R diagram for University(b).

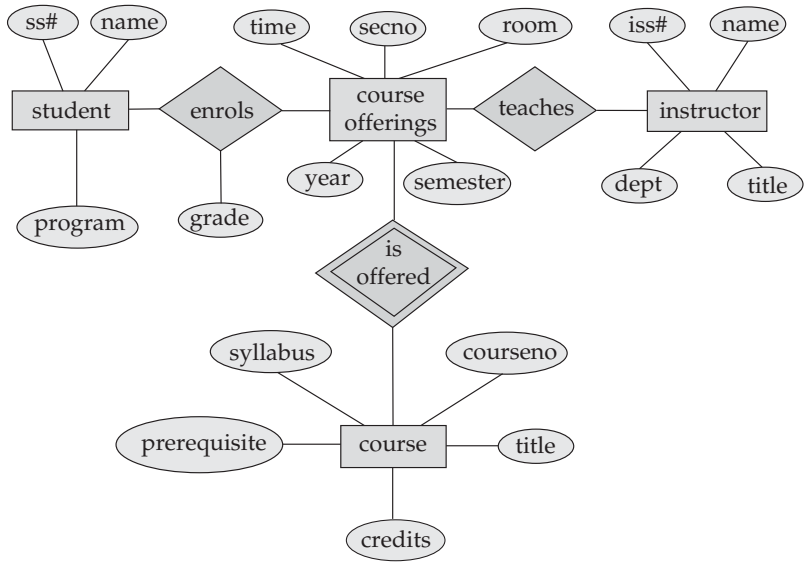


Figure 6.10 E-R diagram for University(c).

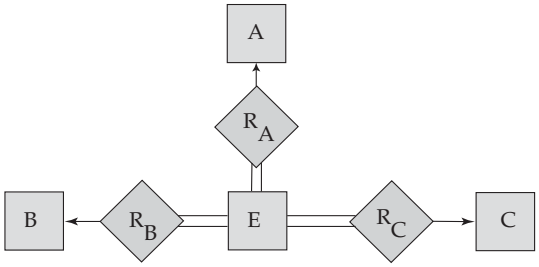


Figure 6.11 E-R diagram to Exercise 6.9b.

c. Suppose A totally participates in the relationhip R, then introduce a total participation constraint between A and R_A.

d. Consider E as a weak entity set and R_A, R_B and R_C as its identifying relationship sets. See Figure 6.12.

6.10 The primary key of a weak entity set can be inferred from its relationship with the strong entity set. If we add primary key attributes to the weak entity set, they will be present in both the entity set and the relationship set and they have to be the same. Hence there will be redundancy.

6.11 *A* inherits all the attributes of *X* plus it may define its own attributes. Similarly *C* inherits all the attributes of *Y* plus its own attributes. *B* inherits the attributes of both *X* and *Y*. If there is some attribute *name* which belongs to both *X* and *Y*, it may be referred to in *B* by the qualified name *X.name* or *Y.name*.

6.12 In this example, we assume that both banks have the shared identifiers for customers, such as the social security number. We see the general solution in the next exercise.

Each of the problems mentioned does have potential for difficulties.

a. *branch_name* is the primary-key of the *branch* entity set. Therefore while merging the two banks' entity sets, if both banks have a branch with the same name, one of them will be lost.

b. customers participate in the relationship sets *cust_banker*, *borrower* and *depositor*. While merging the two banks' *customer* entity sets, duplicate tuples

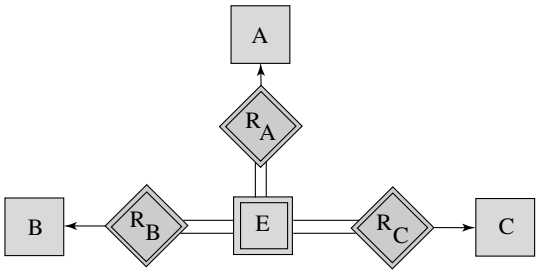


Figure 6.12 E-R diagram to Exercise 6.9d.

of the same customer will be deleted. Therefore those relations in the three mentioned relationship sets which involved these deleted tuples will have to be updated. Note that if the tabular representation of a relationship set is obtained by taking a union of the primary keys of the participating entity sets, no modification to these relationship sets is required.

c. The problem caused by *loans* or *accounts* with the same number in both the banks is similar to the problem caused by branches in both the banks with the same *branch_name*.

To solve the problems caused by the merger, no schema changes are required. Merge the *customer* entity sets removing duplicate tuples with the same *social_security* field. Before merging the *branch* entity sets, prepend the old bank name to the *branch_name* attribute in each tuple. The *employee* entity sets can be merged directly, and so can the *payment* entity sets. No duplicate removal should be performed. Before merging the *loan* and *account* entity sets, whenever there is a number common in both the banks, the old number is replaced by a new unique number, in one of the banks.

Next the relationship sets can be merged. Any relation in any relationship set which involves a tuple which has been modified earlier due to the merger, is itself modified to retain the same meaning. For example let 1611 be a loan number common in both the banks prior to the merger, and let it be replaced by a new unique number 2611 in one of the banks, say bank 2. Now all the relations in *borrower*, *loan_branch* and *loan_payment* of bank 2 which refer to loan number 1611 will have to be modified to refer to 2611. Then the merger with bank 1's corresponding relationship sets can take place.

6.13 This is a case in which the schemas of the two banks differ, so the merger becomes more difficult. The identifying attribute for persons in the US is *social-security*, and in Canada it is *social-insurance*. Therefore the merged schema cannot use either of these. Instead we introduce a new attribute *person_id*, and use this uniformly for everybody in the merged schema. No other change to the schema is required. The values for the *person_id* attribute may be obtained by several ways. One way would be to prepend a country code to the old *social-security* or *social-insurance* values ("U" and "C" respectively, for instance), to get the corresponding *person_id* values. Another way would be to assign fresh numbers starting from 1 upwards, one number to each *social-security* and *social-insurance* value in the old databases.

Once this has been done, the actual merger can proceed as according to the answer to the previous question. If a particular relationship set, say *borrower*, involves only US customers, this can be expressed in the merged database by specializing the entity-set *customer* into *us_customer* and *canada_customer*, and making only *us_customer* participate in the merged *borrower*. Similarly *employee* can be specialized if needed.

CHAPTER 7

Relational-Database Design

Solutions to Practice Exercises

7.1 A decomposition $\{R_1, R_2\}$ is a lossless-join decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since *A* is a candidate key (see Practice Exercise 7.6), Therefore $R_1 \cap R_2 \rightarrow R_1$.

7.2 The nontrivial functional dependencies are: $A \rightarrow B$ and $C \rightarrow B$, and a dependency they logically imply: $AC \rightarrow B$. There are 19 trivial functional dependencies of the form $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$. C does not functionally determine *A* because the first and third tuples have the same *C* but different *A* values. The same tuples also show *B* does not functionally determine *A*. Likewise, *A* does not functionally determine *C* because the first two tuples have the same *A* value and different *C* values. The same tuples also show *B* does not functionally determine *C*.

7.3 Let $Pk(r)$ denote the primary key attribute of relation r.

- The functional dependencies Pk(*account*) $\rightarrow Pk$ (*customer*) and Pk(*customer*) $\rightarrow Pk$(*account*) indicate a one-to-one relationship because any two tuples with the same value for account must have the same value for customer, and any two tuples agreeing on customer must have the same value for account.
- The functional dependency Pk(*account*) $\rightarrow Pk$(*customer*) indicates a many-to-one relationship since any account value which is repeated will have the same customer value, but many account values may have the same customer value.

7.4 To prove that:

$$\text{if } \alpha \rightarrow \beta \text{ and } \alpha \rightarrow \gamma \text{ then } \alpha \rightarrow \beta\gamma$$

Following the hint, we derive:

$\alpha \rightarrow \beta$	given
$\alpha\alpha \rightarrow \alpha\beta$	augmentation rule
$\alpha \rightarrow \alpha\beta$	union of identical sets
$\alpha \rightarrow \gamma$	given
$\alpha\beta \rightarrow \gamma\beta$	augmentation rule
$\alpha \rightarrow \beta\gamma$	transitivity rule and set union commutativity

7.5 Proof using Armstrong's axioms of the Pseudotransitivity Rule:
if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$.

$\alpha \rightarrow \beta$	given
$\alpha\gamma \rightarrow \gamma\beta$	augmentation rule and set union commutativity
$\gamma\beta \rightarrow \delta$	given
$\alpha\gamma \rightarrow \delta$	transitivity rule

7.6 Note: It is not reasonable to expect students to enumerate all of F^+. Some shorthand representation of the result should be acceptable as long as the nontrivial members of F^+ are found.

Starting with $A \rightarrow BC$, we can conclude: $A \rightarrow B$ and $A \rightarrow C$.

Since $A \rightarrow B$ and $B \rightarrow D$, $A \rightarrow D$	(decomposition, transitive)
Since $A \rightarrow CD$ and $CD \rightarrow E$, $A \rightarrow E$	(union, decomposition, transitive)
Since $A \rightarrow A$, we have	(reflexive)
$A \rightarrow ABCDE$ from the above steps	(union)
Since $E \rightarrow A$, $E \rightarrow ABCDE$	(transitive)
Since $CD \rightarrow E$, $CD \rightarrow ABCDE$	(transitive)
Since $B \rightarrow D$ and $BC \rightarrow CD$, $BC \rightarrow ABCDE$	(augmentative, transitive)
Also, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow D$, etc.	

Therefore, any functional dependency with A, E, BC, or CD on the left hand side of the arrow is in F^+, no matter which other attributes appear in the FD. Allow * to represent any set of attributes in R, then F^+ is $BD \rightarrow B$, $BD \rightarrow D$, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow BD$, $B \rightarrow D$, $B \rightarrow B$, $B \rightarrow BD$, and all FDs of the form $A* \rightarrow \alpha$, $BC* \rightarrow \alpha$, $CD* \rightarrow \alpha$, $E* \rightarrow \alpha$ where α is any subset of $\{A, B, C, D, E\}$. The candidate keys are A, BC, CD, and E.

7.7 The given set of FDs F is:

$A \rightarrow BC$
$CD \rightarrow E$
$B \rightarrow D$
$E \rightarrow A$

The left side of each FD in F is unique. Also none of the attributes in the left side or right side of any of the FDs is extraneous. Therefore the canonical cover F_c is equal to F.

7.8 The algorithm is correct because:

- If A is added to $result$ then there is a proof that $\alpha \rightarrow A$. To see this, observe that $\alpha \rightarrow \alpha$ trivially so α is correctly part of $result$. If $A \notin \alpha$ is added to $result$ there must be some FD $\beta \rightarrow \gamma$ such that $A \in \gamma$ and β is already a subset of $result$. (Otherwise $fdcount$ would be nonzero and the **if** condition would be false.) A full proof can be given by induction on the depth of recursion for an execution of **addin**, but such a proof can be expected only from students with a good mathematical background.
- If $A \in \alpha^+$, then A is eventually added to $result$. We prove this by induction on the length of the proof of $\alpha \rightarrow A$ using Armstrong's axioms. First observe that if procedure **addin** is called with some argument β, all the attributes in β will be added to $result$. Also if a particular FD's *fdcount* becomes 0, all the attributes in its tail will definitely be added to $result$. The base case of the proof, $A \in \alpha \Rightarrow A \in \alpha^+$, is obviously true because the first call to **addin** has the argument α. The inductive hypotheses is that if $\alpha \rightarrow A$ can be proved in n steps or less then $A \in result$. If there is a proof in $n+1$ steps that $\alpha \rightarrow A$, then the last step was an application of either reflexivity, augmentation or transitivity on a fact $\alpha \rightarrow \beta$ proved in n or fewer steps. If reflexivity or augmentation was used in the $(n+1)^{st}$ step, A must have been in $result$ by the end of the n^{th} step itself. Otherwise, by the inductive hypothesis $\beta \subseteq result$. Therefore the dependency used in proving $\beta \rightarrow \gamma$, $A \in \gamma$ will have $fdcount$ set to 0 by the end of the n^{th} step. Hence A will be added to $result$.

To see that this algorithm is more efficient than the one presented in the chapter note that we scan each FD once in the main program. The resulting array $appears$ has size proportional to the size of the given FDs. The recursive calls to **addin** result in processing linear in the size of $appears$. Hence the algorithm has time complexity which is linear in the size of the given FDs. On the other hand, the algorithm given in the text has quadratic time complexity, as it may perform the loop as many times as the number of FDs, in each loop scanning all of them once.

7.9 **a.** The query is given below. Its result is non-empty if and only if $b \rightarrow c$ does not hold on r.

select b
from r
group by b
having count(**distinct** c) > 1

b.

```
create assertion b-to-c check
    (not exists
        (select b
         from r
         group by b
         having count(distinct c) > 1
        )
    )
```

7.10 Consider some tuple t in u.

Note that $r_i = \Pi_{R_i}(u)$ implies that $t[R_i] \in r_i$, $1 \leq i \leq n$. Thus,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] \in r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

By the definition of natural join,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] = \Pi_\alpha (\sigma_\beta (t[R_1] \times t[R_2] \times \ldots \times t[R_n]))$$

where the condition β is satisfied if values of attributes with the same name in a tuple are equal and where $\alpha = U$. The cartesian product of single tuples generates one tuple. The selection process is satisfied because all attributes with the same name must have the same value since they are projections from the same tuple. Finally, the projection clause removes duplicate attribute names.

By the definition of decomposition, $U = R_1 \cup R_2 \cup \ldots \cup R_n$, which means that all attributes of t are in $t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n]$. That is, t is equal to the result of this join.

Since t is any arbitrary tuple in u,

$$u \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

7.11 The dependency $B \rightarrow D$ is not preserved. F_1, the restriction of F to (A, B, C) is $A \rightarrow ABC$, $A \rightarrow AB$, $A \rightarrow AC$, $A \rightarrow BC$, $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$, $AB \rightarrow AC$, $AB \rightarrow ABC$, $AB \rightarrow BC$, $AB \rightarrow AB$, $AB \rightarrow A$, $AB \rightarrow B$, $AB \rightarrow C$, AC (same as AB), BC (same as AB), ABC (same as AB). F_2, the restriction of F to (C, D, E) is $A \rightarrow ADE$, $A \rightarrow AD$, $A \rightarrow AE$, $A \rightarrow DE$, $A \rightarrow A$, $A \rightarrow D$, $A \rightarrow E$, $D \rightarrow D$, E (same as A), AD, AE, DE, ADE (same as A). $(F_1 \cup F_2)^+$ is easily seen not to contain $B \rightarrow D$ since the only FD in $F_1 \cup F_2$ with B as the left side is $B \rightarrow B$, a trivial FD. We shall see in Practice Exercise 7.13 that $B \rightarrow D$ is indeed in F^+. Thus $B \rightarrow D$ is not preserved. Note that $CD \rightarrow ABCDE$ is also not preserved.

A simpler argument is as follows: F_1 contains no dependencies with D on the right side of the arrow. F_2 contains no dependencies with B on the left side of the arrow. Therefore for $B \rightarrow D$ to be preserved there must be an FD $B \rightarrow \alpha$ in F_1^+ and $\alpha \rightarrow D$ in F_2^+ (so $B \rightarrow D$ would follow by transitivity). Since the intersection of the two schemes is A, $\alpha = A$. Observe that $B \rightarrow A$ is not in F_1^+ since $B^+ = BD$.

7.12 Let F be a set of functional dependencies that hold on a schema R. Let $\sigma = \{R_1, R_2, \ldots, R_n\}$ be a dependency-preserving 3NF decomposition of R. Let X be a candidate key for R.

Consider a legal instance r of R. Let $j = \Pi_X(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$. We want to prove that $r = j$.

We claim that if t_1 and t_2 are two tuples in j such that $t_1[X] = t_2[X]$, then $t_1 = t_2$. To prove this claim, we use the following inductive argument –
Let $F' = F_1 \cup F_2 \cup \ldots \cup F_n$, where each F_i is the restriction of F to the schema R_i in σ. Consider the use of the algorithm given in Figure 7.9 to compute the closure of X under F'. We use induction on the number of times that the for loop in this algorithm is executed.

- *Basis* : In the first step of the algorithm, $result$ is assigned to X, and hence given that $t_1[X] = t_2[X]$, we know that $t_1[result] = t_2[result]$ is true.
- *Induction Step* : Let $t_1[result] = t_2[result]$ be true at the end of the k th execution of the for loop.
  Suppose the functional dependency considered in the $k + 1$ th execution of the for loop is $\beta \rightarrow \gamma$, and that $\beta \subseteq result$. $\beta \subseteq result$ implies that $t_1[\beta] = t_2[\beta]$ is true. The facts that $\beta \rightarrow \gamma$ holds for some attribute set R_i in σ, and that $t_1[R_i]$ and $t_2[R_i]$ are in $\Pi_{R_i}(r)$ imply that $t_1[\gamma] = t_2[\gamma]$ is also true. Since γ is now added to $result$ by the algorithm, we know that $t_1[result] = t_2[result]$ is true at the end of the $k + 1$ th execution of the for loop.

Since σ is dependency-preserving and X is a key for R, all attributes in R are in $result$ when the algorithm terminates. Thus, $t_1[R] = t_2[R]$ is true, that is, $t_1 = t_2$ – as claimed earlier.

Our claim implies that the size of $\Pi_X(j)$ is equal to the size of j. Note also that $\Pi_X(j) = \Pi_X(r) = r$ (since X is a key for R). Thus we have proved that the size of j equals that of r. Using the result of Practice Exercise 7.10, we know that $r \subseteq j$. Hence we conclude that $r = j$.

Note that since X is trivially in 3NF, $\sigma \cup \{X\}$ is a dependency-preserving lossless-join decomposition into 3NF.

7.13 Given the relation $R' = (A, B, C, D)$ the set of functional dependencies $F' = A \rightarrow B, C \rightarrow D, B \rightarrow C$ allows three distinct BCNF decompositions.

$$R_1 = \{(A, B), (C, D), (B, C)\}$$

is in BCNF as is

$$R_2 = \{(A, B), (C, D), (A, C)\}$$

$$R_2 = \{(A, B), (C, D), (A, C)\}$$

$$R_3 = \{(B, C), (A, D), (A, B)\}$$

7.14 Suppose R is in 3NF according to the textbook definition. We show that it is in 3NF according to the definition in the exercise. Let A be a nonprime attribute in

R that is transitively dependent on a key α for R. Then there exists $\beta \subseteq R$ such that $\beta \rightarrow A$, $\alpha \rightarrow \beta$, $A \notin \alpha$, $A \notin \beta$, and $\beta \rightarrow \alpha$ does not hold. But then $\beta \rightarrow A$ violates the textbook definition of 3NF since

- $A \notin \beta$ implies $\beta \rightarrow A$ is nontrivial
- Since $\beta \rightarrow \alpha$ does not hold, β is not a superkey
- A is not any candidate key, since A is nonprime

Now we show that if R is in 3NF according to the exercise definition, it is in 3NF according to the textbook definition. Suppose R is not in 3NF according the the textbook definition. Then there is an FD $\alpha \rightarrow \beta$ that fails all three conditions. Thus

- $\alpha \rightarrow \beta$ is nontrivial.
- α is not a superkey for R.
- Some A in $\beta - \alpha$ is not in any candidate key.

This implies that A is nonprime and $\alpha \rightarrow A$. Let γ be a candidate key for R. Then $\gamma \rightarrow \alpha$, $\alpha \rightarrow \gamma$ does not hold (since α is not a superkey), $A \notin \alpha$, and $A \notin \gamma$ (since A is nonprime). Thus A is transitively dependent on γ, violating the exercise definition.

7.15 Referring to the definitions in Practice Exercise 7.14, a relation schema R is said to be in 3NF if there is no non-prime attribute A in R for which A is transitively dependent on a key for R.

We can also rewrite the definition of 2NF given here as :
"A relation schema R is in 2NF if no non-prime attribute A is partially dependent on any candidate key for R."

To prove that every 3NF schema is in 2NF, it suffices to show that if a non-prime attribute A is partially dependent on a candidate key α, then A is also transitively dependent on the key α.

Let A be a non-prime attribute in R. Let α be a candidate key for R. Suppose A is partially dependent on α.

- From the definition of a partial dependency, we know that for some proper subset β of α, $\beta \rightarrow A$.
- Since $\beta \subset \alpha$, $\alpha \rightarrow \beta$. Also, $\beta \rightarrow \alpha$ does not hold, since α is a candidate key.
- Finally, since A is non-prime, it cannot be in either β or α.

Thus we conclude that $\alpha \rightarrow A$ is a transitive dependency. Hence we have proved that every 3NF schema is also in 2NF.

7.16 The relation schema $R = (A, B, C, D, E)$ and the set of dependencies

$$\begin{aligned} A &\rightarrow\!\!\!\!\rightarrow BC \\ B &\rightarrow\!\!\!\!\rightarrow CD \\ E &\rightarrow\!\!\!\!\rightarrow AD \end{aligned}$$

constitute a BCNF decomposition, however it is clearly not in 4NF. (It is BCNF because all FDs are trivial).

CHAPTER 8

Application Design and Development

Solutions to Practice Exercises

8.1 The CGI interface starts a new process to service each request, which has a significant operating system overhead. On the other hand, servelets are run as threads of an existing process, avoiding this overhead. Further, the process running threads could be the Web server process itself, avoiding interprocess communication which can be expensive. Thus, for small to moderate sized tasks, the overhead of Java is less than the overheads saved by avoiding process creating and communication.

For tasks involving a lot of CPU activity, this may not be the case, and using CGI with a C or C++ program may give better performance.

8.2 Most computers have limits on the number of simultaneous connections they can accept. With connectionless protocols, connections are broken as soon as the request is satisfied, and therefore other clients can open connections. Thus more clients can be served at the same time. A request can be routed to any one of a number of different servers to balance load, and if a server crashes another can take over without the client noticing any problem.

The drawback of connectionless protocols is that a connection has to be reestablished every time a request is sent. Also, session information has to be sent each time in form of cookies or hidden fields. This makes them slower than the protocols which maintain connections in case state information is required.

8.3 Caching can be used to improve performance by exploiting the commonalities between transactions.

a. If the application code for servicing each request needs to open a connection to the database, which is time consuming, then a pool of open connections may be created before hand, and each request uses one from those.

b. The results of a query generated by a request can be cached. If same request comes agian, or generates the same query, then the cached result can be used instead of connecting to database again.

c. The final webpage generated in response to a request can be cached. If the same request comes again, then the cached page can be outputed.

8.4 For inserting into the materialized view *branch_cust* we must set a database trigger on an insert into *depositor* and *account*. We assume that the database system uses *immediate* binding for rule execution. Further, assume that the current version of a relation is denoted by the relation name itself, while the set of newly inserted tuples is denoted by qualifying the relation name with the prefix – **inserted**.

The active rules for this insertion are given below –

define trigger *insert_into_branch_cust_via_depositor*
after insert on *depositor*
referencing new table as *inserted* **for each statement**
insert into *branch_cust*
  select *branch_name, customer_name*
  from *inserted, account*
  where *inserted.account_number = account.account_number*

define trigger *insert_into_branch_cust_via_account*
after insert on *account*
referencing new table as *inserted* **for each statement**
insert into *branch_cust*
  select *branch_name, customer_name*
  from *depositor*, *inserted*
  where *depositor.account_number = inserted.account_number*

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch_cust*.

The deletion of a tuple from *branch_cust* is similar to insertion, except that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly deleted set of tuples by qualifying the relation name with the keyword **deleted**.

define trigger *delete_from_branch_cust_via_depositor*
after delete on *depositor*
referencing old table as *deleted* **for each statement**
delete from *branch_cust*
  select *branch_name, customer_name*
  from *deleted*, *account*
  where *deleted.account_number = account.account_number*

```
define trigger delete_from_branch_cust_via_account
after delete on account
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from depositor, deleted
    where depositor.account_number = deleted.account_number
```

8.5 Query:

```
create trigger check-delete-trigger after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
    ( select customer_name from depositor
    where account_number <> orow.account_number )
end
```

8.6 The key problem with digital certificates (when used offline, without contacting the certificate issuer) is that there is no way to withdraw them.

For instance (this actually happened, but names of the parties have been changed) person C claims to be an employee of company X and get a new public key certified by the certifying authority A. Suppose the authority A incorrectly believed that C was acting on behalf of company X, it gives C a certificate $cert$. Now, C can communicate with person Y, who checks the certificate $cert$ presenetd by C, and believes the public key contained in $cert$ really belongs to X. Now C would communicate with Y using the public key, and Y trusts the communication is from company X.

Person Y may now reveal confidential information to C, or accept purchase order from C, or execute programs certified by C, based on the public key, thinking he is actually communicating with company X. In each case there is potential for harm to Y.

Even if A detects the impersonation, as long as Y does not check with A (the protocol does not require this check), there is no way for Y to find out that the certificate is forged.

If X was a certification authority itself, further levels of fake certificates can be created. But certificates that are not part of this chain would not be affected.

8.7 A scheme for storing passwords would be to encrypt each password, and then use a hash index on the user-id. The user-id can be used to easily access the encrypted password. The password being used in a login attempt is then encrypted and compared with the stored encryption of the correct password. An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist!

CHAPTER 9

Object-Based Databases

Solutions to Practice Exercises

9.1 For this problem, we use table inheritance. We assume that **MyDate**, **Color** and **DriveTrainType** are pre-defined types.

```
create type Vehicle
    (vehicle_id integer,
     license_number char(15),
     manufacturer char(30),
     model char(30),
     purchase_date MyDate,
     color Color)

create table vehicle of type Vehicle

create table truck
    (cargo_capacity integer)
    under vehicle

create table sportsCar
    (horsepower integer
     renter_age_requirement integer)
    under vehicle

create table van
    (num_passengers integer)
    under vehicle
```

```
create table offRoadVehicle
    (ground_clearance real
     driveTrain DriveTrainType)
    under vehicle
```

9.2 **a.** No Answer.

b. Queries in SQL:1999.

i. Program:

```
select ename
from emp as e, e.ChildrenSet as c
where 'March' in
        (select birthday.month
         from c
        )
```

ii. Program:

```
select e.ename
from emp as e, e.SkillSet as s, s.ExamSet as x
where s.type = 'typing' and x.city = 'Dayton'
```

iii. Program:

```
select distinct s.type
from emp as e, e.SkillSet as s
```

9.3 **a.** The corresponding SQL:1999 schema definition is given below. Note that the derived attribute *age* has been translated into a method.

```
create type Name
    (first_name varchar(15),
     middle_initial char,
     last_name varchar(15))
create type Street
    (street_name varchar(15),
     street_number varchar(4),
     apartment_number varchar(7))
create type Address
    (street Street,
     city varchar(15),
     state varchar(15),
     zip_code char(6))
create table customer
    (name Name,
     customer_id varchar(10),
     address Adress,
     phones  char(7)  array[10],
     dob date)
```

method integer *age*()

b. **create function** *Name* (*f* **varchar**(15), *m* **char**, *l* **varchar**(15))
returns *Name*
begin
set *first_name* = *f*;
set *middle_initial* = *m*;
set *last_name* = *l*;
end
create function *Street* (*sname* **varchar**(15), *sno* **varchar**(4), *ano* **varchar**(7))
returns *Street*
begin
set *street_name* = *sname*;
set *street_number* = *sno*;
set *apartment_number* =*ano*;
end
create function *Address* (*s Street*, *c* **varchar**(15), *sta* **varchar**(15), *zip* **varchar**(6))
returns *Address*
begin
set *street* = *s*;
set *city* = *c*;
set *state* =*sta*;
set *zip_code* =*zip*;
end

9.4 **a.** The schema definition is given below. Note that backward references can be addedbut they are not so important as in OODBS because queries can be written in SQL and joins can take care of integrity constraints.

create type *Employee*
(*person_name* **varchar(30)**,
street **varchar(15)**,
city **varchar(15)**)
create type *Company*
(*company_name* **varchar(15)**,
(*city* **varchar(15)**)
create table *employee* of *Employee*
create table *company* of *Company*
create type *Works*
(*person* **ref**(*Employee*) **scope** *employee*,
comp **ref**(*Company*) **scope** *company*,
salary **int**)
create table *works* of *Works*
create type *Manages*
(*person* **ref**(*Employee*) **scope** *employee*,
(*manager* **ref**(*Employee*) **scope** *employee*)

create table *manages* of *Manages*

b. i. **select** *comp*− >*name*
from *works*
group by *comp*
having count(*person*) ≥ **all(select count(***person*)
from *works*
group by *comp*)

ii. **select** *comp*− >*name*
from *works*
group by *comp*
having sum(*salary*) ≤ **all(select sum(***salary*)
from *works*
group by *comp*)

iii. **select** *comp*− >*name*
from *works*
group by *comp*
having avg(*salary*) > **(select avg(***salary*)
from *works*
where *comp*− >*company_name*="First Bank Corporation")

9.5 **a.** A computer-aided design system for a manufacturer of airplanes:
An OODB system would be suitable for this. That is because CAD requires complex data types, and being computation oriented, CAD tools are typically used in a programming language environment needing to access the database.

b. A system to track contributions made to candidates for public office:
A relational system would be apt for this, as data types are expected to be simple, and a powerful querying mechanism is essential.

c. An information system to support the making of movies:
Here there will be extensive use of multimedia and other complex data types. But queries are probably simple, and thus an object relational system is suitable.

9.6 An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it, and these messages are implemented by the methods.

CHAPTER 10

XML

Solutions to Practice Exercises

10.1 **a.** The XML representation of data using attributes is shown in Figure 10.1.
b. The DTD for the bank is shown in Figure 10.2.

10.2 Query:

```
<!DOCTYPE db [
    <!ELEMENT emp (ename, children*, skills*)>
    <!ELEMENT children (name, birthday)>
    <!ELEMENT birthday (day, month, year)>
    <!ELEMENT skills (type, exams+)>
    <!ELEMENT exams (year, city)>
    <!ELEMENT ename( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT day( #PCDATA )>
    <!ELEMENT month( #PCDATA )>
    <!ELEMENT year( #PCDATA )>
    <!ELEMENT type( #PCDATA )>
    <!ELEMENT city( #PCDATA )>
] >
```

10.3 Code:

```
/db/emp/skills/type
```

```
<bank>
    <account account-number="A-101" branch-name="Downtown"
            balance="500">
    </account>
    <account account-number="A-102" branch-name="Perryridge"
            balance="400">
    </account>
    <account account-number="A-201" branch-name="Brighton"
            balance="900">
    </account>
    <customer customer-name="Johnson" customer-street="Alma"
            customer-city="Palo Alto">
    </customer>
    <customer customer-name="Hayes" customer-street="Main"
            customer-city="Harrison">
    </customer>
    <depositor account-number="A-101" customer-name="Johnson">
    </depositor>
    <depositor account-number="A-201" customer-name="Johnson">
    </depositor>
    <depositor account-number="A-102" customer-name="Hayes">
    </depositor>
</bank>
```

Figure 10.1 XML representation.

```
<!DOCTYPE bank [
    <!ELEMENT account >
    <!ATTLIST account
        account-number ID #REQUIRED
        branch-name CDATA #REQUIRED
        balance CDATA #REQUIRED >
    <!ELEMENT customer >
    <!ATTLIST customer
        customer-name ID #REQUIRED
        customer-street CDATA #REQUIRED
        customer-street CDATA #REQUIRED >
    <!ELEMENT depositor >
    <!ATTLIST depositor
        account-number IDREF #REQUIRED
        customer-name IDREF #REQUIRED >
] >
```

Figure 10.2 The DTD for the bank.

10.4 Query:

```
for $b in distinct (/bank/account/branch-name)
return
<branch-total>
      <branch-name> $b/text() </branch-name>
      let $s := sum (/bank/account[branch-name=$b]/balance)
      return <total-balance> $s </total-balance>
</branch-total>
```

10.5 Query:

```
      <lojoin>
      for $b in /bank/account,
            $c in /bank/customer,
            $d in /bank/depositor
      where $a/account-number = $d/account-number
            and $c/customer-name = $d/customer-name
      return <cust-acct> $c $a </cust-acct>
      |
      for $c in /bank/customer,
      where every $d in /bank/depositor satisfies
      (not ($c/customer-name=$d/customer-name))
      return <cust-acct> $c </cust-acct>
      </lojoin>
```

10.6 The answer in XQuery is

```
<bank-2>
      for $c in /bank/customer
      return
            <customer>
                  <customer-name> $c/* </customer-name>
                  for $a in $c/id(@accounts)
                  return $a
            </customer>
</bank-2>
```

10.7 Realtion schema:

book (bid, title, year, publisher, place)
article (artid, title, journal, year, number, volume, pages)
book_author (bid, first_name,last_name, order)
article_author (artid, first_name,last_name, order)

10.8 The answer is shwn in Figure 10.3.

```
nodes(1,element,bank,–)
nodes(2,element,account,–)
nodes(3,element,account,–)
nodes(4,element,account,–)
nodes(5,element,customer,–)
nodes(6,element,customer,–)
nodes(7,element,depositor,–)
nodes(8,element,depositor,–)
nodes(9,element,depositor,–)
child(2,1) child(3,1) child(4,1)
child(5,1) child(6,1)
child(7,1) child(8,1) child(9,1)
nodes(10,element,account-number,A-101)
nodes(11,element,branch-name,Downtown)
nodes(12,element,balance,500)
child(10,2) child(11,2) child(12,2)
nodes(13,element,account-number,A-102)
nodes(14,element,branch-name,Perryridge)
nodes(15,element,balance,400)
child(13,3) child(14,3) child(15,3)
nodes(16,element,account-number,A-201)
nodes(17,element,branch-name,Brighton)
nodes(18,element,balance,900)
child(16,4) child(17,4) child(18,4)
nodes(19,element,customer-name,Johnson)
nodes(20,element,customer-street,Alma)
nodes(21,element,customer-city,Palo Alto)
child(19,5) child(20,5) child(21,5)
nodes(22,element,customer-name,Hayes)
nodes(23,element,customer-street,Main)
nodes(24,element,customer-city,Harrison)
child(22,6) child(23,6) child(24,6)
nodes(25,element,account-number,A-101)
nodes(26,element,customer-name,Johnson)
child(25,7) child(26,7)
nodes(27,element,account-number,A-201)
nodes(28,element,customer-name,Johnson)
child(27,8) child(28,8)
nodes(29,element,account-number,A-102)
nodes(30,element,customer-name,Hayes)
child(29,9) child(30,9)
```

Figure 10.3 Relational Representation of XML Data as Trees.

10.9 **a.** The answer is shown in Figure 10.4.

b. Show how to map this DTD to a relational schema.

part(partid,name)
subpartinfo(partid, subpartid, qty)

Attributes partid and subpartid of subpartinfo are foreign keys to part.

c. No answer

```
<parts>
    <part>
        <name> bicycle </name>
        <subpartinfo>
            <part>
                <name> wheel </name>
                <subpartinfo>
                    <part>
                        <name> rim </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> spokes </name>
                    </part>
                    <qty> 40 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> tire </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> brake </name>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> gear </name>
            </part>
            <qty> 3 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> frame </name>
            </part>
            <qty> 1 </qty>
        </subpartinfo>
    </part>
</parts>
```

Figure 10.4 Example Parts Data in XML.

CHAPTER 11

Storage and File Structure

Solutions to Practice Exercises

11.1 This arrangement has the problem that P_i and B_{4i-3} are on the same disk. So if that disk fails, reconstruction of B_{4i-3} is not possible, since data and parity are both lost.

11.2 **a.** To ensure atomicity, a block write operation is carried out as follows:

i. Write the information onto the first physical block.
ii. When the first write completes successfully, write the same information onto the second physical block.
iii. The output is declared completed only after the second write completes successfully.

During recovery, each pair of physical blocks is examined. If both are identical and there is no detectable partial-write, then no further actions are necessary. If one block has been partially rewritten, then we replace its contents with the contents of the other block. If there has been no partial-write, but they differ in content, then we replace the contents of the first block with the contents of the second, or vice versa. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates both copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

b. The idea is similar here. For any block write, the information block is written first followed by the corresponding parity block. At the time of

recovery, each set consisting of the n^{th} block of each of the disks is considered. If none of the blocks in the set have been partially-written, and the parity block contents are consistent with the contents of the information blocks, then no further action need be taken. If any block has been partially-written, it's contents are reconstructed using the other blocks. If no block has been partially-written, but the parity block contents do not agree with the information block contents, the parity block's contents are reconstructed.

11.3 **a.** MRU is preferable to LRU where $R_1 \bowtie R_2$ is computed by using a nested-loop processing strategy where each tuple in R_2 must be compared to each block in R_1. After the first tuple of R_2 is processed, the next needed block is the first one in R_1. However, since it is the least recently used, the LRU buffer management strategy would replace that block if a new block was needed by the system.

b. LRU is preferable to MRU where $R_1 \bowtie R_2$ is computed by sorting the relations by join values and then comparing the values by proceeding through the relations. Due to duplicate join values, it may be necessary to "back-up" in one of the relations. This "backing-up" could cross a block boundary into the most recently used block, which would have been replaced by a system using MRU buffer management, if a new block was needed.

Under MRU, some unused blocks may remain in memory forever. In practice, MRU can be used only in special situations like that of the nested-loop strategy discussed in example 0.a

11.4 **a.** Although moving record 6 to the space for 5, and moving record 7 to the space for 6, is the most straightforward approach, it requires moving the most records, and involves the most accesses.

b. Moving record 7 to the space for 5 moves fewer records, but destroys any ordering in the file.

c. Marking the space for 5 as deleted preserves ordering and moves no records, but requires additional overhead to keep track of all of the free space in the file. This method may lead to too many "holes" in the file, which if not compacted from time to time, will affect performance because of reduced availability of contiguous free records.

11.5 (We use "↑ i" to denote a pointer to record "i".)
The original file of Figure 11.8.

header				↑ 1
record 0	A-102	Perryridge	400	
record 1				↑ 4
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4		Perryridge		↑ 6
record 5	A-201		900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

a. The file after **insert** (Brighton, A-323, 1600).

header				↑ 4
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

b. The file after **delete** record 2.

header				↑ 2
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2				↑ 4
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

The free record chain could have alternatively been from the header to 4, from 4 to 2, and finally from 2 to 6.

c. The file after **insert** (Brighton, A-626, 2000).

header				↑ 4
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2	A-626	Brighton	2000	
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

11.6 Instance of relations:

course relation

course_name	room	instructor	
Pascal	CS-101	Calvin, B	c_1
C	CS-102	Calvin, B	c_2
Lisp	CS-102	Kess, J	c_3

course_name	student_name	grade
Pascal	Carper, D	A
Pascal	Merrick, L	A
Pascal	Mitchell, N	B
Pascal	Bliss, A	C
Pascal	Hames, G	C
C	Nile, M	A
C	Mitchell, N	B
C	Carper, D	A
C	Hurly, I	B
C	Hames, G	A
Lisp	Bliss, A	C
Lisp	Hurly, I	B
Lisp	Nile, M	D
Lisp	Stars, R	A
Lisp	Carper, D	A

Block 0 contains: c_1, e_1, e_2, e_3, e_4, and e_5
Block 1 contains: c_2, e_6, e_7, e_8, e_9 and e_{10}
Block 2 contains: c_3, e_{11}, e_{12}, e_{13}, e_{14}, and e_{15}

11.7 **a.** Everytime a record is inserted/deleted, check if the usage of the block has changed levels. In that case, update the corrosponding bits. Note that we don't need to access the bitmaps at all unless the usage crosses a boundary, so in most of the cases there is no overhead.

b. When free space for a large record or a set of records is sought, then multiple free list entries may have to be scanned before finding a proper sized one, so overheads are much higher. With bitmaps, one page of bitmap can store free info for many pages, so I/O spent for finding free space is minimal. Similarly, when a whole block or a large part of it is deleted, bitmap technique is more convenient for updating free space information.

CHAPTER 12

Indexing and Hashing

Solutions to Practice Exercises

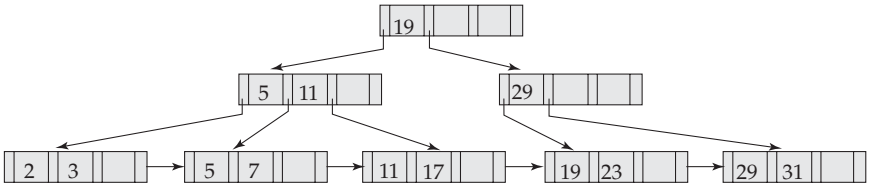
12.1 Reasons for not keeping several search indices include:

- **a.** Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
- **b.** Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).
- **c.** Each extra index requires additional storage space.
- **d.** For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist.

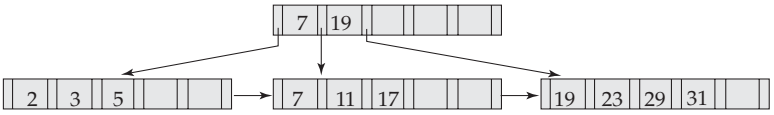
12.2 In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

12.3 The following were generated by inserting values into the B^+-tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil n/2 \rceil$ values/pointers.

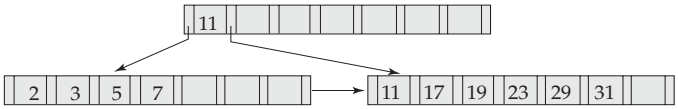
a.



b.

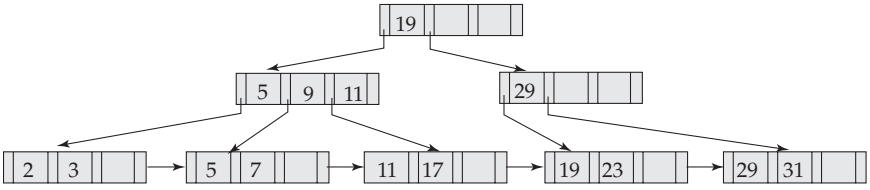


c.

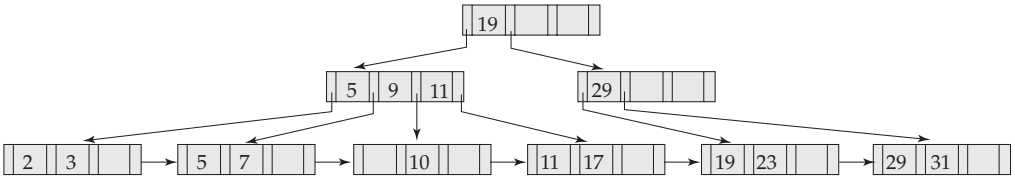


12.4 • With structure 12.3.a:

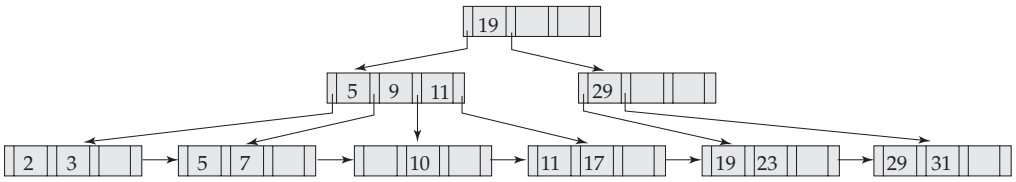
Insert 9:



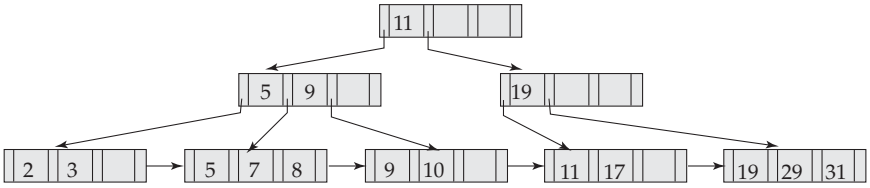
Insert 10:



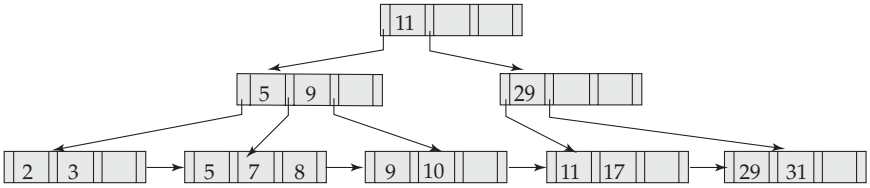
Insert 8:



Delete 23:

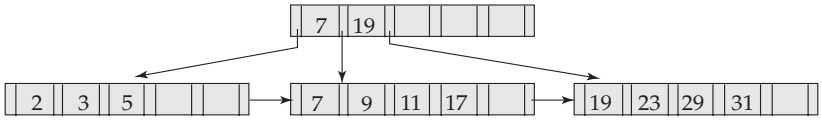


Delete 19:

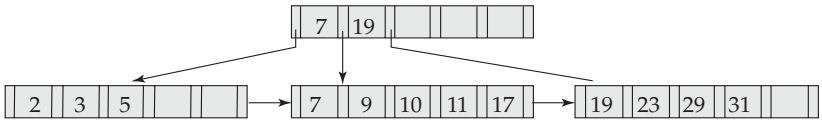


- With structure 12.3.b:

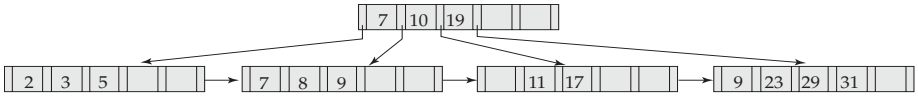
  Insert 9:



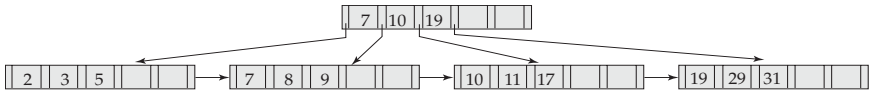
  Insert 10:



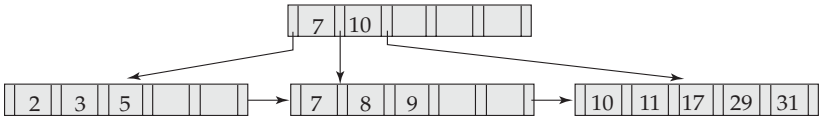
Insert 8:



Delete 23:

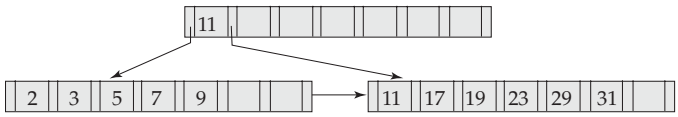


Delete 19:

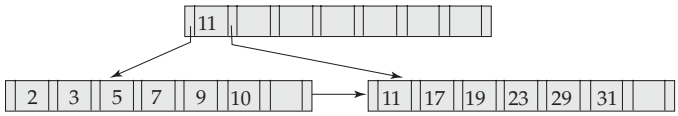


- With structure 12.3.c:

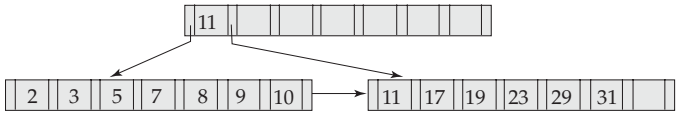
  Insert 9:



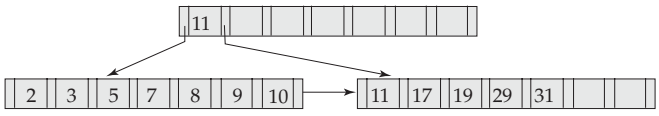
  Insert 10:



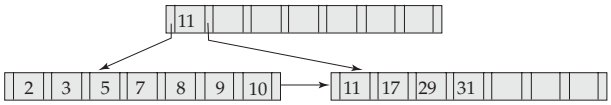
  Insert 8:



  Delete 23:



Delete 19:



12.5 If there are *K* search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $log_{\lfloor (m-1)n/m \rfloor}(K)$

12.6 The algorithm for insertion into a B-tree is:

Locate the leaf node into which the new key-pointer pair should be inserted. If there is space remaining in that leaf node, perform the insertion at the correct location, and the task is over. Otherwise insert the key-pointer pair conceptually into the correct location in the leaf node, and then split it along the middle. The middle key-pointer pair does not go into either of the resultant nodes of the split operation. Instead it is inserted into the parent node, along with the tree pointer to the new child. If there is no space in the parent, a similar procedure is repeated.

The deletion algorithm is:

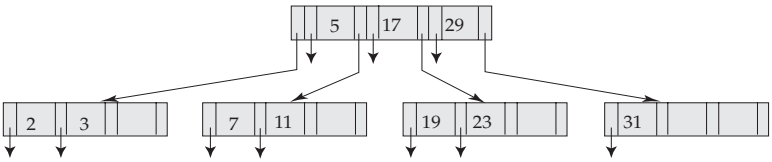
Locate the key value to be deleted, in the B-tree.

a. If it is found in a leaf node, delete the key-pointer pair, and the record from the file. If the leaf node contains less than $\lceil n/2 \rceil - 1$ entries as a result of this deletion, it is either merged with its siblings, or some entries are redistributed to it. Merging would imply a deletion, whereas redistribution would imply change(s) in the parent node's entries. The deletions may ripple upto the root of the B-tree.

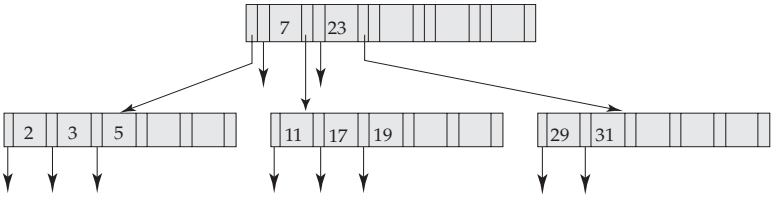
b. If the key value is found in an internal node of the B-tree, replace it and its record pointer by the smallest key value in the subtree immediately to its right and the corresponding record pointer. Delete the actual record in the database file. Then delete that smallest key value-pointer pair from the subtree. This deletion may cause further rippling deletions till the root of the B-tree.

Below are the B-trees we will get after insertion of the given key values. We assume that leaf and non-leaf nodes hold the same number of search key values.

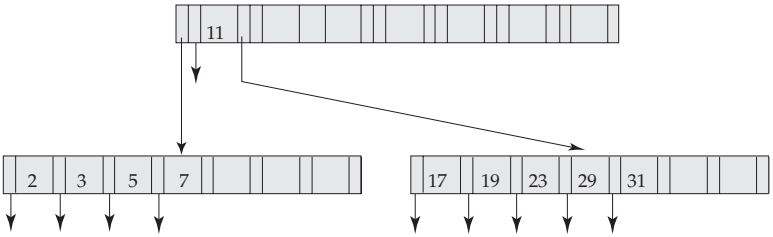
a.



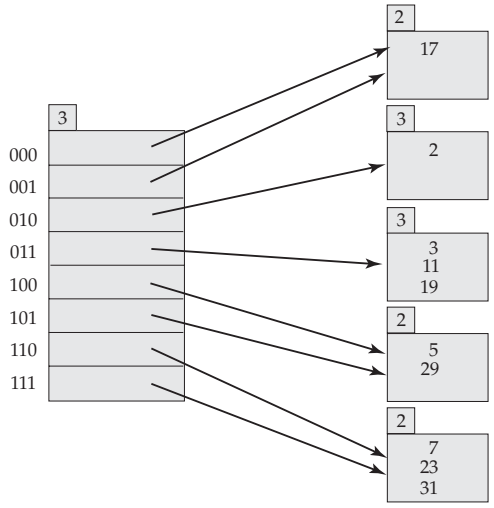
b.



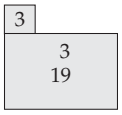
c.



12.7 Extendable hash structure

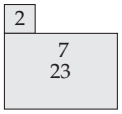


12.8 **a.** Delete 11: From the answer to Exercise 12.7, change the third bucket to:

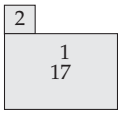


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

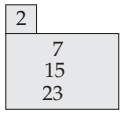
b. Delete 31: From the answer to 12.7, change the last bucket to:



c. Insert 1: From the answer to 12.7, change the first bucket to:



d. Insert 15: From the answer to 12.7, change the last bucket to:



12.9 Let i denote the number of bits of the hash value used in the hash table. Let **bsize** denote the maximum capacity of each bucket.

```
delete(value K_l)
begin
    j = first i high-order bits of h(K_l);
    delete value K_l from bucket j;
    coalesce(bucket j);
end

coalesce(bucket j)
begin
    i_j = bits used in bucket j;
    k = any bucket with first (i_j − 1) bits same as that
        of bucket j while the bit i_j is reversed;
    i_k = bits used in bucket k;
    if(i_j ≠ i_k)
        return; /* buckets cannot be merged */
    if(entries in j + entries in k > bsize)
        return; /* buckets cannot be merged */
    move entries of bucket k into bucket j;

    decrease the value of i_j by 1;
    make all the bucket-address-table entries,
    which pointed to bucket k, point to j;

    coalesce(bucket j);
end
```

Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket j differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

When merge is successful, further merging may be possible, which is handled by a recursive call to *coalesce* at the end of the function.

12.10 If the hash table is currently using i bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly i.

Consider a bucket j with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to i, then reset the count to 1. If the bucket is being

split and i_j is one less that i, then increase the count by 1. It the bucket if being coalesced, and i_j is equal to i then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.

However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e. a count for each value of common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when sufficient number of entries at the end of count array become 0.

12.11 We reproduce the account relation of Figure 12.25 below.

A-217	Brighton	750
A-101	Downtown	500
A-1 10	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Bitmaps for *branch_name*

Brighton	1	0	0	0	0	0	0	0	0
Downtown	0	1	1	0	0	0	0	0	0
Mianus	0	0	0	1	0	0	0	0	0
Perryridge	0	0	0	0	1	1	1	0	0
Redwood	0	0	0	0	0	0	0	1	0
Round hill	0	0	0	0	0	0	0	0	1

Bitmaps for *balance*

L_1	0	0	0	0	0	0	0	0	0
L_2	0	0	0	0	1	0	0	0	1
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0

where, level L_1 is below 250, level L_2 is from 250 to below 500, L_3 from 500 to below 750 and level L_4 is above 750.

To find all accounts in Downtown with a balance of 500 or more, we find the union of bitmaps for levels L_3 and L_4 and then intersect it with the bitmap for Downtown.

Downtown	0	1	1	0	0	0	0	0	0
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0
$L_3 \cup L_4$	1	1	1	1	0	1	1	1	0
Downtown	0	1	1	0	0	0	0	0	0
Downtown $\cap (L_3 \cup L_4)$	0	1	1	0	0	0	0	0	0

Thus, the required tuples are A-101 and A-110.

12.12 No answer

CHAPTER 13

Query Processing

Solutions to Practice Exercises

13.1 Query:

$$\Pi_{T.branch\_name}((\Pi_{branch\_name,\ assets}(\rho_T(branch))) \bowtie_{T.assets>S.assets} (\Pi_{assets}\ (\sigma_{(branch\_city='Brooklyn')}(\rho_S(branch)))))$$

This expression performs the theta join on the smallest amount of data possible. It does this by restricting the right hand side operand of the join to only those branches in Brooklyn, and also eliminating the unneeded attributes from both the operands.

13.2 We will refer to the tuples (kangaroo, 17) through (baboon, 12) using tuple numbers t_1 through t_{12}. We refer to the j^{th} run used by the i^{th} pass, as r_{ij}. The initial sorted runs have three blocks each. They are:

$$\begin{aligned} r_{11} &= \{t_3, t_1, t_2\} \\ r_{12} &= \{t_6, t_5, t_4\} \\ r_{13} &= \{t_9, t_7, t_8\} \\ r_{14} &= \{t_{12}, t_{11}, t_{10}\} \end{aligned}$$

Each pass merges three runs. Therefore the runs after the end of the first pass are:

$$\begin{aligned} r_{21} &= \{t_3, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\} \\ r_{22} &= \{t_{12}, t_{11}, t_{10}\} \end{aligned}$$

At the end of the second pass, the tuples are completely sorted into one run:

$$r_{31} = \{t_{12}, t_3, t_{11}, t_{10}, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}$$

13.3 r_1 needs 800 blocks, and r_2 needs 1500 blocks. Let us assume M pages of memory. If $M > 800$, the join can easily be done in 1500 + 800 disk accesses, using even plain nested-loop join. So we consider only the case where $M \leq 800$ pages.

a. Nested-loop join:
Using r_1 as the outer relation we need $20000 * 1500 + 800 = 30,000,800$ disk accesses, if r_2 is the outer relation we need $45000 * 800 + 1500 = 36,001,500$ disk accesses.

b. Block nested-loop join:
If r_1 is the outer relation, we need $\lceil \frac{800}{M-1} \rceil * 1500 + 800$ disk accesses, if r_2 is the outer relation we need $\lceil \frac{1500}{M-1} \rceil * 800 + 1500$ disk accesses.

c. Merge-join:
Assuming that r_1 and r_2 are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil log_{M-1}(1500/M) \rceil + 2) + 800(2\lceil log_{M-1}(800/M) \rceil + 2)$ disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

d. Hash-join:
We assume no overflow occurs. Since r_1 is smaller, we use it as the build relation and r_2 as the probe relation. If $M > 800/M$, i.e. no need for recursive partitioning, then the cost is $3(1500 + 800) = 6900$ disk accesses, else the cost is $2(1500 + 800)\lceil log_{M-1}(800) - 1 \rceil + 1500 + 800$ disk accesses.

13.4 If there are multiple tuples in the inner relation with the same value for the join attributes, we may have to access that many blocks of the inner relation for each tuple of the outer relation. That is why it is inefficient. To reduce this cost we can perform a join of the outer relation tuples with just the secondary index leaf entries, postponing the inner relation tuple retrieval. The result file obtained is then sorted on the inner relation addresses, allowing an efficient physical order scan to complete the join.

Hybrid merge–join requires the outer relation to be sorted. The above algorithm does not have this requirement, but for each tuple in the outer relation it needs to perform an index lookup on the inner relation. If the outer relation is much larger than the inner relation, this index lookup cost will be less than the sorting cost, thus this algorithm will be more efficient.

13.5 We can store the entire smaller relation in memory, read the larger relation block by block and perform nested loop join using the larger one as the outer relation. The number of I/O operations is equal to $b_r + b_s$, and memory requirement is $min(b_r, b_s) + 2$ pages.

13.6 **a.** Use the index to locate the first tuple whose *branch_city* field has value "Brooklyn". From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch_city* field is anything other than "Brooklyn".

c. This query is equivalent to the query

$$\sigma_{(branch\_city \geq \text{'Brooklyn'} \wedge assets < 5000)}(branch)$$

Using the *branch-city* index, we can retrieve all tuples with *branch-city* value greater than or equal to "Brooklyn" by following the pointer chains from the first "Brooklyn" tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

13.7 Let *outer* be the iterator which returns successive tuples from the pipelined outer relation. Let *inner* be the iterator which returns successive tuples of the inner relation having a given value at the join attributes. The *inner* iterator returns these tuples by performing an index lookup. The functions **IndexedNLJoin::open**, **IndexedNLJoin::close** and **IndexedNLJoin::next** to implement the indexed nested-loop join iterator are given below. The two iterators *outer* and *inner*, the value of the last read outer relation tuple t_r and a flag $done_r$ indicating whether the end of the outer relation scan has been reached are the state information which need to be remembered by **IndexedNLJoin** between calls.

```
IndexedNLJoin::open()
begin
    outer.open();
    inner.open();
    done_r := false;
    if(outer.next() ≠ false)
        move tuple from outer's output buffer to t_r;
    else
        done_r := true;
end
```

```
IndexedNLJoin::close()
begin
    outer.close();
    inner.close();
end
```

```
boolean IndexedNLJoin::next()
begin
    while(¬done_r)
    begin
        if(inner.next(t_r[JoinAttrs]) ≠ false)
        begin
            move tuple from inner's output buffer to t_s;
            compute t_r ⋈ t_s and place it in output buffer;
            return true;
        end
        else
            if(outer.next() ≠ false)
            begin
                move tuple from outer's output buffer to t_r;
                rewind inner to first tuple of s;
            end
            else
                done_r := true;
    end
    return false;
end
```

13.8 Suppose $r(T \cup S)$ and $s(S)$ be two relations and $r \div s$ has to be computed.

For sorting based algorithm, sort relation s on S. Sort relation r on (T, S). Now, start scanning r and look at the T attribute values of the first tuple. Scan r till tuples have same value of T. Also scan s simultaneously and check whether every tuple of s also occurs as the S attribute of r, in a fashion similar to merge join. If this is the case, output that value of T and proceed with the next value of T. Relation s may have to be scanned multiple times but r will only be scanned once. Total disk accesses, after sorting both the relations, will be $|r| + N * |s|$, where N is the number of distinct values of T in r.

We assume that for any value of T, all tuples in r with that T value fit in memory, and consider the general case at the end. Partition the relation r on attributes in T such that each partition fits in memory (always possible because of our assumption). Consider partitions one at a time. Build a hash table on the tuples, at the same time collecting all distinct T values in a separate hash table. For each value of T, Now, for each value V_T of T, each value s of S, probe the hash table on (V_T, s). If any of the values is absent, discard the value V_T, else output the value V_T.

In the case that not all r tuples with one value for T fit in memory, partition r and s on the S attributes such that the condition is satisfied, run the algorithm on each corresponding pair of partitions r_i and s_i. Output the intersection of the T values generated in each partition.

13.9 Seek overhead is reduced, but the the number of runs that can be merged in a pass decreases potentially leading to more passes Should choose a value of b_b that minimizes overall cost.

CHAPTER 14

Query Optimization

Solutions to Practice Exercises

14.1 **a.** $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

Let us rename $(E_1 \bowtie_\theta (E_2 - E_3))$ as R_1, $(E_1 \bowtie_\theta E_2)$ as R_2 and $(E_1 \bowtie_\theta E_3)$ as R_3. It is clear that if a tuple t belongs to R_1, it will also belong to R_2. If a tuple t belongs to R_3, $t[E_3$'s attributes] will belong to E_3, hence t cannot belong to R_1. From these two we can say that

$$\forall t,\ t \in R_1 \ \Rightarrow\ t \in (R_2 - R_3)$$

It is clear that if a tuple t belongs to $R_2 - R_3$, then $t[R_2$'s attributes] $\in E_2$ and $t[R_2$'s attributes] $\notin E_3$. Therefore:

$$\forall t,\ t \in (R_2 - R_3) \ \Rightarrow\ t \in R_1$$

The above two equations imply the given equivalence.

This equivalence is helpful because evaluation of the right hand side join will produce many tuples which will finally be removed from the result. The left hand side expression can be evaluated more efficiently.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.

θ uses only attributes from A. Therefore if any tuple t in the output of ${}_A\mathcal{G}_F(E)$ is filtered out by the selection of the left hand side, all the tuples in E whose value in A is equal to $t[A]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E)) \ \Rightarrow\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E))$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E)) \ \Rightarrow\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E))$$

The above two equations imply the given equivalence.

This equivalence is helpful because evaluation of the right hand side avoids performing the aggregation on groups which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$ where θ uses only attributes from E_1.

θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 \supset\!\bowtie E_2)$ is filtered out by the selection of the left hand side, all the tuples in E_1 whose value is equal to $t[E_1]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2) \ \Rightarrow\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2 \ \Rightarrow\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2)$$

The above two equations imply the given equivalence.

This equivalence is helpful because evaluation of the right hand side avoids producing many output tuples which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

14.2 **a.** $R = \{(1,2)\},\ S = \{(1,3)\}$

The result of the left hand side expression is $\{(1)\}$, whereas the result of the right hand side expression is empty.

b. $R = \{(1,2),(1,5)\}$

The left hand side expression has an empty result, whereas the right hand side one has the result $\{(1,2)\}$.

c. Yes, on replacing the max by the min, the expressions will become equivalent. Any tuple that the selection in the rhs eliminates would not pass the selection on the lhs if it were the minimum value, and would be eliminated anyway if it were not the minimum value.

d. $R = \{(1,2)\},\ S = \{(2,3)\},\ T = \{(1,4)\}$. The left hand expression gives $\{(1,2,null,4)\}$ whereas the the right hand expression gives $\{(1,2,3,null)\}$.

e. Let R be of the schema (A,B) and S of (A,C). Let $R = \{(1,2)\},\ S = \{(2,3)\}$ and let θ be the expression $C = 1$. The left side expression's result is empty, whereas the right side expression results in $\{(1,2,null)\}$.

14.3 **a.** We define the multiset versions of the relational-algebra operators here. Given multiset relations r_1 and r_2,

i. σ

Let there be c_1 copies of tuple t_1 in r_1. If t_1 satisfies the selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$, otherwise there are none.

ii. Π

For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

iii. $\times$

If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, then there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

iv. $\bowtie$

The output will be the same as a cross product followed by a selection.

v. $-$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 - c_2$ copies of t in $r_1 - r_2$, provided that $c_1 - c_2$ is positive.

vi. $\cup$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 + c_2$ copies of t in $r_1 \cup r_2$.

vii. $\cap$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $min(c_1, c_2)$ copies of t in $r_1 \cap r_2$.

b. All the equivalence rules 1 through 7.b of section 14.2.1 hold for the multiset version of the relational-algebra defined in the first part.

There exist equivalence rules which hold for the ordinary relational-algebra, but do not hold for the multiset version. For example consider the rule :-

$$A \cap B \quad = \quad A \cup B \; - \; (A - B) \; - \; (B - A)$$

This is clearly valid in plain relational-algebra. Consider a multiset instance in which a tuple t occurs 4 times in A and 3 times in B. t will occur 3 times in the output of the left hand side expression, but 6 times in the output of the right hand side expression. The reason for this rule to not hold in the multiset version is the asymmetry in the semantics of multiset union and intersection.

14.4

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because E is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute C for relation r_2 and on E for r_3. Then for each tuple in r_1, we do the following:
  - **a.** Use the index for r_2 to look up at most one tuple which matches the C value of r_1.
  - **b.** Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

14.5 The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r_1, $1500/V(C, r_2) = 15/11$ tuples (on the average) of r_2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r_3 to yield a result of approximately 10,227 tuples ($15000/11 \times 750/100 = 10227$). A good strategy should join r_1 and r_2 first, since

the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

14.6 **a.** Use the index to locate the first tuple whose *branch_city* field has value "Brooklyn". From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch_city* field is anything other than "Brooklyn".

c. This query is equivalent to the query:

$$\sigma_{(branch\_city \geq \text{'Brooklyn'} \wedge assets < 5000)}(branch).$$

Using the *branch_city* index, we can retrieve all tuples with *branch_city* value greater than or equal to "Brooklyn" by following the pointer chains from the first "Brooklyn" tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

14.7 Each join order is a complete binary tree (every non-leaf node has exactly two children) with the relations as the leaves. The number of different complete binary trees with n leaf nodes is $\frac{1}{n}\binom{2(n-1)}{(n-1)}$. This is because there is a bijection between the number of complete binary trees with n leaves and number of binary trees with $n-1$ nodes. Any complete binary tree with n leaves has $n-1$ internal nodes. Removing all the leaf nodes, we get a binary tree with $n-1$ nodes. Conversely, given any binary tree with $n-1$ nodes, it can be converted to a complete binary tree by adding n leaves in a unique way. The number of binary trees with $n-1$ nodes is given by $\frac{1}{n}\binom{2(n-1)}{(n-1)}$, known as the Catalan number. Multiplying this by $n!$ for the number of permutations of the n leaves, we get the desired result.

14.8 Consider the dynamic programming algorithm given in Section 14.4.2. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x = 2$. Thus the total cost is less than 3^n.

14.9 The derivation of time taken is similar to the general case, except that instead of considering $2^{k+1}-2$ subsets of size less than or equal to k for A, we only need to consider $k+1$ subsets of size exactly equal to k. That is because the right hand operand of the topmost join has to be a single relation. Therefore the total cost for finding the best join order for all subsets of size $k+1$ is $\binom{n}{k+1}(k+1)$, which is equal to $n\binom{n-1}{k}$. Summing over all k from 1 to $n-1$ using the binomial expansion of $(1+x)^{n-1}$ with $x = 1$, gives a total cost of less than $n2^{n-1}$.

14.10 **a.** The nested query is as follows:

```
select   S.acount_number
from     account S
where    S.branch_name like 'B%' and
         S.balance =
         (select max(T.balance)
         from account T
         where T.branch_name = S.branch_name)
```

b. The decorrelated query is as follows:

```
create table t1 as
        select branch_name, max(balance)
        from account
        group by branch_name
select  account_number
from    account, t1
where   account.branch_name like 'B%' and
        account.branch_name = t1.branch_name and
        account.balance = t1.balance
```

c. In general, consider the queries of the form:

```
select   ···
from     L1
where    P1 and
         A1 op
         (select f(A2)
         from L2
         where P2)
```

where, f is some aggregate function on attributes A_2, and *op* is some boolean binary operator. It can be rewritten as

```
create table t1 as
        select f(A2),V
        from L2
        where P2^1
        group by V
select  ···
from    L1, t1
where   P1 and P2^2 and
        A1 op t1.A2
```

where P_2^1 contains predicates in P_2 without selections involving correlation variables, and P_2^2 introduces the selections involving the correlation variables. V contains all the attributes that are used in the selections involving correlation variables in the nested query.

CHAPTER 15

Transactions

Solutions to Practice Exercises

15.1 Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

15.2 There are several steps in the creation of a file. A storage area is assigned to the file in the file system, a unique i-number is given to the file and an i-node entry is inserted into the i-list. Deletion of file involves exactly opposite steps.

For the file system user in UNIX, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor though, many of the internal file system actions need to have transaction semantics. All the steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

15.3 Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured. In contrast, most users of file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

15.4 If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average responce time will increase. Also when the transaction is reading data from disk, CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur.

15.5 Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

15.6 There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, T_1, T_2, T_3, T_4, T_5.

15.7 A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.

CHAPTER 16

Concurrency Control

Solutions to Practice Exercises

16.1 Suppose two-phase locking does not ensure serializability. Then there exists a set of transactions $T_0,\ T_1\ ...\ T_{n-1}$ which obey 2PL and which produce a non-serializable schedule. A non-serializable schedule implies a cycle in the precedence graph, and we shall show that 2PL cannot produce such cycles. Without loss of generality, assume the following cycle exists in the precedence graph: $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{n-1} \rightarrow T_0$. Let α_i be the time at which T_i obtains its last lock (i.e. T_i's lock point). Then for all transactions such that $T_i \rightarrow T_j$, $\alpha_i\ <\ \alpha_j$. Then for the cycle we have

$$\alpha_0\ <\ \alpha_1\ <\ \alpha_2\ <\ ...\ <\ \alpha_{n-1}\ <\ \alpha_0$$

Since $\alpha_0\ <\ \alpha_0$ is a contradiction, no such cycle can exist. Hence 2PL cannot produce non-serializable schedules. Because of the property that for all transactions such that $T_i \rightarrow T_j$, $\alpha_i\ <\ \alpha_j$, the lock point ordering of the transactions is also a topological sort ordering of the precedence graph. Thus transactions can be serialized according to their lock points.

16.2 **a.** Lock and unlock instructions:

T_{31}: **lock-S**(A)
read(A)
lock-X(B)
read(B)
if $A\ =\ 0$
then $B\ :=\ B\ +\ 1$
write(B)
unlock(A)
unlock(B)

T_{32}: **lock-S**(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

b. Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T_{31}	T_{32}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

The transactions are now deadlocked.

16.3 Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

16.4 The proof is in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

16.5 Consider the tree-structured database graph given below.

Schedule possible under tree protocol but not under 2PL:

T_1	T_2
lock *(A)*	
lock *(B)*	
unlock *(A)*	
	lock *(A)*
lock *(C)*	
unlock *(B)*	
	lock *(B)*
	unlock *(A)*
	unlock *(B)*
unlock *(C)*	

Schedule possible under 2PL but not under tree protocol:

T_1	T_2
lock *(A)*	
	lock *(B)*
lock *(C)*	
	unlock *(B)*
unlock *(A)*	
unlock *(C)*	

16.6 The proof is in Kedem and Silberschatz, "Locking Protocols: From Exclusive to Shared Locks," JACM Vol. 30, 4, 1983.

16.7 The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979.

16.8 The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979.

16.9 The access protection mechanism can be used to implement page level locking. Consider reads first. A process is allowed to read a page only after it read-locks the page. This is implemented by using `mprotect` to initially turn off read permissions to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler associated with protection violation then requests a read lock on the page, and after the lock is acquired, it uses `mprotect` to allow read access to the page by the process, and finally allows the process to continue. Write access is handled similarly.

16.10 The proof is in Korth, "Locking Primitives in a Database System," JACM Vol. 30, 1983.

16.11 It would make no difference. The write protocol is such that the most recent transaction to write an item is also the one with the largest timestamp to have done so.

16.12 If a transaction needs to access a large a set of items, multiple granularity locking requires fewer locks, whereas if only one item needs to be accessed, the single lock granularity system allows this with just one lock. Because all the desired data items are locked and unlocked together in the multiple granularity scheme, the locking overhead is low, but concurrency is also reduced.

16.13 In the concurrency control scheme of Section 16.3 choosing **Start**(T_i) as the timestamp of T_i gives a subset of the schedules allowed by choosing **Validation**(T_i) as the timestamp. Using **Start**(T_i) means that whoever started first must finish first. Clearly transactions could enter the validation phase in the same order in which they began executing, but this is overly restrictive. Since choosing **Validation**(T_i) causes fewer nonconflicting transactions to restart, it gives the better response times.

16.14

- Two-phase locking: Use for simple applications where a single granularity is acceptable. If there are large read-only transactions, multiversion protocols would do better. Also, if deadlocks must be avoided at all costs, the tree protocol would be preferable.
- Two-phase locking with multiple granularity locking: Use for an application mix where some applications access individual records and others access whole relations or substantial parts thereof. The drawbacks of 2PL mentioned above also apply to this one.
- The tree protocol: Use if all applications tend to access data items in an order consistent with a particular partial order. This protocol is free of deadlocks, but transactions will often have to lock unwanted nodes in order to access the desired nodes.
- Timestamp ordering: Use if the application demands a concurrent execution that is equivalent to a particular serial ordering (say, the order of arrival), rather than *any* serial ordering. But conflicts are handled by roll-back of transactions rather than waiting, and schedules are not recoverable. To make them recoverable, additional overheads and increased response time have to be tolerated. Not suitable if there are long read-only transactions, since they will starve. Deadlocks are absent.
- Validation: If the probability that two concurrently executing transactions conflict is low, this protocol can be used advantageously to get better concurrency and good response times with low overheads. Not suitable under high contention, when a lot of wasted work will be done.
- Multiversion timestamp ordering: Use if timestamp ordering is appropriate but it is desirable for read requests to never wait. Shares the other disadvantages of the timestamp ordering protocol.
- Multiversion two-phase locking: This protocol allows read-only transactions to always commit without ever waiting. Update transactions follow 2PL, thus allowing recoverable schedules with conflicts solved by waiting rather than roll-back. But the problem of deadlocks comes back, though read-only transactions cannot get involved in them. Keeping multiple ver-

sions adds space and time overheads though, therefore plain 2PL may be preferable in low conflict situations.

16.15 A transaction waits on *a.* disk I/O and *b.* lock acquisition. Transactions generally wait on disk reads and not on disk writes as disk writes are handled by the buffering mechanism in asynchronous fashion and transactions update only the in-memory copy of the disk blocks.

The technique proposed essentially separates the waiting times into two phases. The first phase – where transaction is executed without acquiring any locks and without performing any writes to the database – accounts for almost all the waiting time on disk I/O as it reads all the data blocks it needs from disk if they are not already in memory. The second phase—the transaction re-execution with strict two-phase locking—accounts for all the waiting time on acquiring locks. The second phase may, though rarely, involve a small waiting time on disk I/O if a disk block that the transaction needs is flushed to memory (by buffer manager) before the second phase starts.

The technique may increase concurrency as transactions spend almost no time on disk I/O with locks held and hence locks are held for shorter time. In the first phase the transaction reads all the data items required—and not already in memory—from disk. The locks are acquired in the second phase and the transaction does almost no disk I/O in this phase. Thus the transaction avoids spending time in disk I/O with locks held.

The technique may even increase disk throughput as the disk I/O is not stalled for want of a lock. Consider the following scenario with strict two-phase locking protocol: A transaction is waiting for a lock, the disk is idle and there are some item to be read from disk. In such a situation disk bandwidth is wasted. But in the proposed technique, the transaction will read all the required item from the disk without acquiring any lock and the disk bandwidth may be properly utilized.

Note that the proposed technique is most useful if the computation involved in the transactions is less and most of the time is spent in disk I/O and waiting on locks, as is usually the case in disk-resident databases. If the transaction is computation intensive, there may be wasted work. An optimization is to save the updates of transactions in a temporary buffer, and instead of reexecuting the transaction, to compare the data values of items when they are locked with the values used earlier. If the two values are the same for all items, then the buffered updates of the transaction are executed, instead of reexecuting the entire transaction.

16.16 Consider two transactions T_1 and T_2 shown below.

T_1	T_2
write(p)	
	read(p)
	read(q)
write(q)	

Let TS(T_1) < TS(T_2) and let the timestamp test at each operation except write(q) be successful. When transaction T_1 does the timestamp test for write(q) it finds that TS(T_1) < R-timestamp(q), since TS(T_1) < TS(T_2) and R-timestamp(q) = TS(T_2). Hence the writeoperation fails and transaction T_1 rolls back. The cascading results in transaction T_2 also being rolled back as it uses the value for item p that is written by transaction T_1.

If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

16.17 In the text, we considered two approaches to dealing with the phantom phenomenon by means of locking. The coarser granularity approach obviously works for timestamps as well. The B^+-tree index based approach can be adapted to timestamping by treating index buckets as data items with timestamps associated with them, and requiring that all read accesses use an index. We now show that this simple method works. Suppose a transaction T_i wants to access all tuples with a particular range of search-key values, using a B^+-tree index on that search-key. T_i will need to read all the buckets in that index which have key values in that range. It can be seen that any delete or insert of a tuple with a key-value in the same range will need to write one of the index buckets read by T_i. Thus the logical conflict is converted to a conflict on an index bucket, and the phantom phenomenon is avoided.

16.18 Note: The tree-protocol of Section 16.1.5 which is referred to in this question, is different from the multigranularity protocol of Section 16.4 and the B^+-tree concurrency protocol of Section 16.9.

One strategy for early lock releasing is given here. Going down the tree from the root, if the currently visited node's child is not full, release locks held on all nodes except the current node, request an X-lock on the child node, after getting it release the lock on the current node, and then descend to the child. On the other hand, if the child is full, retain all locks held, request an X-lock on the child, and descend to it after getting the lock. On reaching the leaf node, start the insertion procedure. This strategy results in holding locks only on the full index tree nodes from the leaf upwards, until and including the first non-full node.

An optimization to the above strategy is possible. Even if the current node's child is full, we can still release the locks on all nodes but the current one. But after getting the X-lock on the child node, we split it right away. Releasing the lock on the current node and retaining just the lock on the appropriate split child, we descend into it making it the current node. With this optimization, at any time at most two locks are held, of a parent and a child node.

CHAPTER 17

Recovery System

Solutions to Practice Exercises

17.1

- The recovery scheme using a log with deferred updates has the following advantages over the recovery scheme with immediate updates:
  - **a.** The scheme is easier and simpler to implement since fewer operations and routines are needed, i.e., no UNDO.
  - **b.** The scheme requires less overhead since no extra I/O operations need to be done until commit time (log records can be kept in memory the entire time).
  - **c.** Since the old values of data do not have to be present in the log-records, this scheme requires less log storage space.
- The disadvantages of the deferred modification scheme are :
  - **a.** When a data item needs to accessed, the transaction can no longer directly read the correct page from the database buffer, because a previous write by the same transaction to the same data item may not have been propagated to the database yet. It might have updated a local copy of the data item and deferred the actual database modification. Therefore finding the correct version of a data item becomes more expensive.
  - **b.** This scheme allows less concurrency than the recovery scheme with immediate updates. This is because write-locks are held by transactions till commit time.
  - **c.** For long transaction with many updates, the memory space occupied by log records and local copies of data items may become too high.

17.2 The first phase of recovery is to undo the changes done by the failed transactions, so that all data items which have been modified by them get back the

values they had before the *first* of the failed transactions started. If several of the failed transactions had modified the same data item, forward processing of log-records for undo-list transactions would make the data item get the value which it had before the *last* failed transaction to modify that data item started. This is clearly wrong, and we can see that reverse prcessing gets us the desired result.

The second phase of recovery is to redo the changes done by committed transactons, so that all data items which have been modified by them are restored to the value they had after the *last* of the committed transactions finished. It can be seen that only forward processing of log-records belonging to redo-list transactions can guarantee this.

17.3 Interactive transactions are more difficult to recover from than batch transactions because some actions may be irrevocable. For example, an output (write) statement may have fired a missile, or caused a bank machine to give money to a customer. The best way to deal with this is to try to do all output statements at the end of the transaction. That way if the transaction aborts in the middle, no harm will be have been done.

17.4

- Consider the a bank account A with balance \$100. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. The log records corresponding to the updates of A by transactions T_1 and T_2 would be $< T_1, A, 100, 110 >$ and $< T_2, A, 110, 120 >$ resp.

  Say, we wish to undo transaction T_1. The normal transaction undo mechanism will replaces the value in question—A in this example—by the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism the resulting balance would be \$100 and we would, in effect, undo both transactions, whereas we intend to undo only transaction T_1.
- Let the erroneous transaction be T_e.
  - ☐ Identify the latest checkpoint, say C, in the log before the log record $< T_e$, *START*$>$.
  - ☐ Redo all log records starting from the checkpoint C till the log record $< T_e$, *COMMIT*$>$. Some transaction—apart from transaction T_e—would be active at the commit time of transaction T_e. Let S_1 be the set of such transactions.
  - ☐ Rollback T_e and the transactions in the set S_1.
  - ☐ Scan the log further starting from the log record $< T_e$, *COMMIT*$>$ till the end of the log. Note the transactions that were started after the commit point of T_e. Let the set of such transactions be S_2. Re-execute the transactions in set S_1 and S_2 logically.

- Consider again an example from the first item. Let us assume that both transactions are undone and the balance is reverted back to the original value \$100.

  Now we wish to redo transaction T_2. If we redo the log record $< T_2, A, 110, 120 >$ corresponding to transaction T_2 the balance would become \$120 and we would, in effect, redo both transactions, whereas we intend to redo only transaction T_2.

17.5 This is implemented by using `mprotect` to initially turn off access to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler accociated with protection violation then requests a write lock on the page, and after the lock is acquired, it writes the initial contents (before-image) of the page to the log. It then uses `mprotect` to allow write access to the page by the process, and finally allows the process to continue. When the transaction is ready to commit, and before it releases the lock on the page, it writes the contents of the page (after-image) to the log. These before- and after- images can be used for recovery after a crash.

This scheme can be optimized to not write the whole page to log for undo logging, provided the program pins the page in memory.

17.6 We can maintain the LSNs of such pages in an array in a separate disk page. The LSN entry of a page on the disk is the sequence number of the latest log record reflected on the disk. In the normal case, as the LSN of a page resides in the page itself, the page and its LSN are in consistent state. But in the modified scheme as the LSN of a page resides in a separate page it may not be written to the disk at a time when the actual page is written and thus the two may not be in consistent state.

If a page is written to the disk before its LSN is updated on the disk and the system crashes then, during recovery, the page LSN read from the LSN array from the disk is older than the sequence number of the log record reflected to the disk. Thus some updates on the page will be redone unnecessarily but this is fine as updates are idempotent. But if the page LSN is written to the disk to before the actual page is written and the system crashes then some of the updates to the page may be lost. The sequence number of the log record corresponding to the latest update to the page that made to the disk is older than the page LSN in the LSN array and all updates to the page between the two LSNs are lost.

Thus the LSN of a page should be written to the disk only after the page has been written and; we can ensure this as follows: before writing a page containing the LSN array to the disk, we should flush the corresponding pages to the disk. (We can maintain the page LSN at the time of the last flush of each page in the buffer separately, and avoid flushing pages that have been flushed already.)

CHAPTER 18

Data Analysis and Mining

Solutions to Practice Exercises

18.1 query:

groupby rollup(a), **rollup**(b), **rollup**(c), **rollup**(d)

18.2 We assume that multiple students do not have the same marks since otherwise the question is not deterministic; the query below deterministically returns all students with the same marks as the n student, so it may return more than n students.

select *student*, **sum**(*marks*) **as** *total*,
    rank() **over** (**order by** (*total*) **desc**) **as** *trank*
from S
groupby *student*
having *trank* $\leq n$

18.3 query:

select *t1.account-number*, *t1.date-time*, *sum(t2.value)*
from *transaction* as *t1*, *transaction* as *t2*
where *t1.account-number* = *t2.account-number* **and**
    t2.date-time < *t1.date-time*
groupby *t1.account-number*, *t1.date-time*
order by *t1.account-number*, *t1.date-time*

18.4 query:

```
(select color, size, sum(number)
 from sales
 groupby color, size
)
union
(select color, 'all', sum(number)
 from sales
 groupby color
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', 'all', sum(number)
 from sales
)
```

18.5 In a destination-driven architecture for gathering data, data transfers from the data sources to the data-warehouse are based on demand from the warehouse, whereas in a source-driven architecture, the transfers are initiated by each source.

The benefits of a source-driven architecture are

- Data can be propagated to the destination as soon as it becomes available. For a destination-driven architecture to collect data as soon as it is available, the warehouse would have to probe the sources frequently, leading to a high overhead.
- The source does not have to keep historical information. As soon as data is updated, the source can send an update message to the destination and forget the history of the updates. In contrast, in a destination-driven architecture, each source has to maintain a history of data which have not yet been collected by the data warehouse. Thus storage requirements at the source are lower for a source-driven architecture.

On the other hand, a destination-driven architecture has the following advantages.

- In a source-driven architecture, the source has to be active and must handle error conditions such as not being able to contact the warehouse for some time. It is easier to implement passive sources, and a single active warehouse. In a destination-driven architecure, each source is required to provide only a basic functionality of executing queries.

- The warehouse has more control on when to carry out data gathering activities, and when to process user queries; it is not a good idea to perform both simultaneously, since they may conflict on locks.

18.6 Consider the following pair of rules and their confidence levels :

No.	Rule	Conf.
1.	$\forall\ persons\ \ P,\ 10000 < P.salary \leq 20000 \Rightarrow P.credit = good$	60%
2.	$\forall\ persons\ \ P,\ 20000 < P.salary \leq 30000 \Rightarrow P.credit = good$	90%

The new rule has to be assigned a confidence-level which is between the confidence-levels for rules 1 and 2. Replacing the original rules by the new rule will result in a loss of confidence-level information for classifying persons, since we cannot distinguish the confidence levels of perople earning between 10000 and 20000 from those of people earning between 20000 and 30000. Therefore we can combine the two rules without loss of information only if their confidences are the same.

18.7 query:

select *store-id*, *city*, *state*, *country*,
date, *month*, *quarter*, *year*,
sum(*number*), **sum**(*price*)
from *sales*, *store*, *date*
where *sales.store-id* = *store.store-id* **and**
sales.date = *date.date*
groupby rollup(*country*, *state*, *city*, *store-id)*,
rollup(*year*, *quarter*, *month*, *date*)

CHAPTER 19

Information Retrieval

Solutions to Practice Exercises

19.1 We do not consider the questions containing neither of the keywords as their relevance to the keywords is zero. The number of words in a question include stop words. We use the equations given in Section 19.2.1 to compute relevance; the log term in the equation is assumed to be to the base 2.

Q#	#wo--rds	# "SQL"	#"rela--tion"	"SQL" term freq.	"relation" term freq.	"SQL" relv.	"relation" relv.	Tota relv.
1	84	1	1	0.0170	0.0170	0.0002	0.0002	0.0004
4	22	0	1	0.0000	0.0641	0.0000	0.0029	0.0029
5	46	1	1	0.0310	0.0310	0.0006	0.0006	0.0013
6	22	1	0	0.0641	0.0000	0.0029	0.0000	0.0029
7	33	1	1	0.0430	0.0430	0.0013	0.0013	0.0026
8	32	1	3	0.0443	0.1292	0.0013	0.0040	0.0054
9	77	0	1	0.0000	0.0186	0.0000	0.0002	0.0002
14	30	1	0	0.0473	0.0000	0.0015	0.0000	0.0015
15	26	1	1	0.0544	0.0544	0.0020	0.0020	0.0041

19.2 Let S be a set of n keywords. An algorithm to find all documents that contain at least k of these keywords is given below :

This algorithm calculates a reference count for each document identifier. A reference count of i for a document identifier d means that at least i of the keywords in S occur in the document identified by d. The algorithm maintains a

list of records, each having two fields – a document identifier, and the reference count for this identifier. This list is maintained sorted on the document identifier field.

```
initialize the list L to the empty list;
for (each keyword c in S) do
begin
   D := the list of documents identifiers corresponding to c;
   for (each document identifier d in D) do
      if (a record R with document identifier as d is on list L) then
            R.reference_count := R.reference_count + 1;
      else begin
            make a new record R;
            R.document_id := d;
            R.reference_count := 1;
            add R to L;
      end;
end;
for (each record R in L) do
   if (R.reference_count >= k) then
      output R;
```

Note that execution of the second *for* statement causes the list D to "merge" with the list L. Since the lists L and D are sorted, the time taken for this merge is proportional to the sum of the lengths of the two lists. Thus the algorithm runs in time (at most) proportional to n times the sum total of the number of document identifiers corresponding to each keyword in S.

19.3 No answer

19.4 No answer

19.5 No answer

CHAPTER 20

Database System Architectures

Solutions to Practice Exercises

20.1 The drawbacks would be that two interprocess messages would be required to acquire locks, one for the request and one to confirm grant. Interprocess communication is much more expensive than memory access, so the cost of locking would increase. The process storing the shared structures could also become a bottleneck.

The benefit of this alternative is that the lock table is protected better from erroneous updates since only one process can access it.

20.2 With powerful clients, it still makes sense to have a client-server system, rather than a fully centralized system. If the data-server architecture is used, the powerful clients can off-load all the long and compute intensive transaction processing work from the server, freeing it to perform only the work of satisfying read-write requests. even if the transaction-server model is used, the clients still take care of the user-interface work, which is typically very compute-intensive.

A fully distributed system might seem attractive in the presence of powerful clients, but client-server systems still have the advantage of simpler concurrency control and recovery schemes to be implemented on the server alone, instead of having these actions distributed in all the machines.

20.3 **a.** We assume that objects are smaller than a page and fit in a page. If the interconnection link is slow it is better to choose object shipping, as in page shipping a lot of time will be wasted in shipping objects that might never be needed. With a fast interconnection though, the communication overheads and latencies, not the actual volume of data to be shipped, becomes the bottle neck. In this scenario page shipping would be preferable.

b. Two benefits of an having an object-cache rather than a page-cache, even if page shipping is used, are:-

i. When a client runs out of cache space, it can replace objects without replacing entire pages. The reduced caching granularity might result in better cache-hit ratios.

ii. It is possible for the server to ask clients to return some of the locks which they hold, but don't need (lock de-escalation). Thus there is scope for greater concurrency. If page caching is used, this is not possible.

20.4 Since the part which cannot be parallelized takes 20% of the total running time, the best speedup we can hope for has to be less than 5.

20.5 With the central server, each site does not have to remember which site to contact when a particular data item is to be requested. The central server alone needs to remember this, so data items can be moved around easily, depending on which sites access which items most frequently. Other house-keeping tasks are also centralized rather than distributed, making the system easier to develop and maintain. Of course there is the disadvantage of a total shutdown in case the server becomes unavailable. Even if it is running, it may become a bottleneck because every request has to be routed via it.

CHAPTER 21

Parallel Databases

Solutions to Practice Exercises

21.1 If there are few tuples in the queried range, then each query can be processed quickly on a single disk. This allows parallel execution of queries with reduced overhead of initiating queries on multiple disks.

On the other hand, if there are many tuples in the queried range, each query takes a long time to execute as there is no parallelism within its execution. Also, some of the disks can become hot-spots, further increasing response time.

Hybrid range partitioning, in which small ranges (a few blocks each) are partitioned in a round-robin fashion, provides the benefits of range partitioning without its drawbacks.

21.2 **a.** When there are many small queries, inter-query parallelism gives good throughput. Parallelizing each of these small queries would increase the initiation overhead, without any significant reduction in response time.

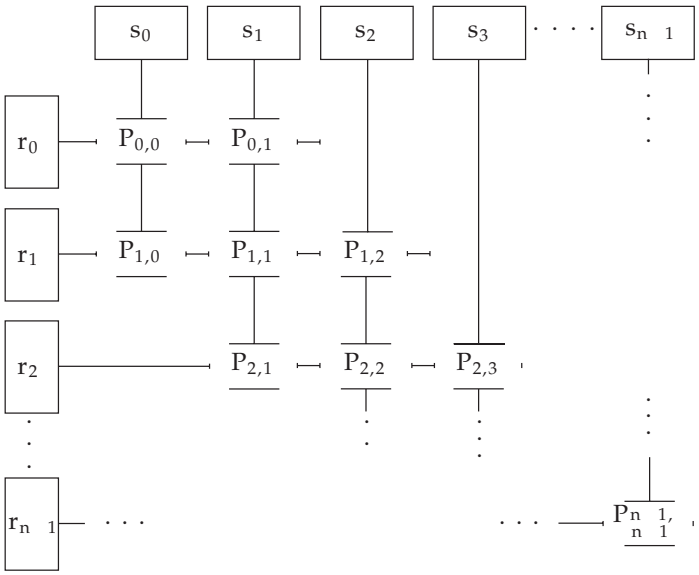
b. With a few large queries, intra-query parallelism is essential to get fast response times. Given that there are large number of processors and disks, only intra-operation parallelism can take advantage of the parallel hardware – for queries typically have few operations, but each one needs to process a large number of tuples.

21.3 **a.** The speed-up obtained by parallelizing the operations would be offset by the data transfer overhead, as each tuple produced by an operator would have to be transferred to its consumer, which is running on a different processor.

b. In a shared-memory architecture, transferring the tuples is very efficient. So the above argument does not hold to any significant degree.

c. Even if two operations are independent, it may be that they both supply their outputs to a common third operator. In that case, running all three on the same processor may be better than transferring tuples across processors.

21.4 Relation r is partitioned into n partitions, $r_0, r_1, \ldots, r_{n-1}$, and s is also partitioned into n partitions, $s_0, s_1, \ldots, s_{n-1}$. The partitions are replicated and assigned to processors as shown below.



Each fragment is replicated on 3 processors only, unlike in the general case where it is replicated on n processors. The number of processors required is now approximately $3n$, instead of n^2 in the general case. Therefore given the same number of processors, we can partition the relations into more fragments with this optimization, thus making each local join faster.

21.5 **a.** A partitioning vector which gives 5 partitions with 20 tuples in each partition is: $[21, 31, 51, 76]$. The 5 partitions obtained are $1-20$, $21-30$, $31-50$, $51-75$ and $76-100$. The assumption made in arriving at this partitioning vector is that within a histogram range, each value is equally likely.

b. Let the histogram ranges be called $h_1, h_2, \ldots, h_h$, and the partitions $p_1, p_2, \ldots, p_p$. Let the frequencies of the histogram ranges be $n_1, n_2, \ldots, n_h$. Each partition should contain N/p tuples, where $N = \Sigma_{i=1}^{h} n_i$.

To construct the load balanced partitioning vector, we need to determine the value of the k_1^{th} tuple, the value of the k_2^{th} tuple and so on, where $k_1 = N/p$, $k_2 = 2N/p$ etc, until k_{p-1}. The partitioning vector will then be $[k_1, k_2, \ldots, k_{p-1}]$. The value of the k_i^{th} tuple is determined as follows. First determine the histogram range h_j in which it falls. Assuming all values in

a range are equally likely, the k_i^{th} value will be

$$s_j + (e_j - s_j) * \frac{k_{ij}}{n_j}$$

where

s_j : first value in h_j
e_j : last value in h_j
k_{ij} : $k_i - \Sigma_{l=1}^{j-1} n_l$

21.6 **a.** The copies of the data items at a processor should be partitioned across multiple other processors, rather than stored in a single processor, for the following reasons:

- to better distribute the work which should have been done by the failed processor, among the remaining processors.
- Even when there is no failure, this technique can to some extent deal with hot-spots created by read only transactions.

b. RAID level 0 itself stores an extra copy of each data item (mirroring). Thus this is similar to mirroring performed by the database itself, except that the database system does not have to bother about the details of performing the mirroring. It just issues the write to the RAID system, which automatically performs the mirroring.

RAID level 5 is less expensive than mirroring in terms of disk space requirement, but writes are more expensive, and rebuilding a crashed disk is more expensive.

CHAPTER 22

Distributed Databases

Solutions to Practice Exercises

22.1 Data transfer on a local-area network (LAN) is much faster than on a wide-area network (WAN). Thus replication and fragmentation will not increase throughput and speed-up on a LAN, as much as in a WAN. But even in a LAN, replication has its uses in increasing reliability and availability.

22.2 **a.** The types of failure that can occur in a distributed system include
   i. Computer failure (site failure).
   ii. Disk failure.
   iii. Communication failure.

b. The first two failure types can also occur on centralized systems.

22.3 A proof that 2PC guarantees atomic commits/aborts inspite of site and link failures, follows. The main idea is that after all sites reply with a <**ready** T> message, only the co-ordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a site can happen only after it ascertains the co-ordinator's decision, either directly from the co-ordinator, or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

a. A site can abort a transaction T (by writing an <**abort** T> log record) only under the following circumstances:

   i. It has not yet written a <**ready** T> log-record. In this case, the co-ordinator could not have got, and will not get a <**ready** T> or <**commit** T> message from this site. Therefore only an abort decision can be made by the co-ordinator.

ii. It has written the <**ready** T> log record, but on inquiry it found out that some other site has an <**abort** T> log record. In this case it is correct for it to abort, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually aborting.

iii. It is itself the co-ordinator. In this case also no site could have committed, or will commit in the future, because commit decisions can be made only by the co-ordinator.

b. A site can commit a transaction T (by writing an <**commit** T> log record) only under the following circumstances:

i. It has written the <**ready** T> log record, and on inquiry it found out that some other site has a <**commit** T> log record. In this case it is correct for it to commit, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually committing.

ii. It is itself the co-ordinator. In this case no other participating site can abort/ would have aborted, because abort decisions are made only by the co-ordinator.

22.4 Site A cannot distinguish between the three cases until communication has resumed with site B. The action which it performs while B is inaccessible must be correct irrespective of which of these situations has actually occurred, and must be such that B can re-integrate consistently into the distributed system once communication is restored.

22.5 We can have a scheme based on sequence numbers similar to the scheme based on timestamps. We tag each message with a sequence number that is unique for the (sending site, receiving site) pair. The number is increased by 1 for each new message sent from the sending site to the receiving site.

The receiving site stores and acknowledges a received message only if it has received all lower numbered messages also; the message is stored in the *received-messages* relation.

The sending site retransmits a message until it has received an ack from the receiving site containing the sequence number of the transmitted message, or a higher sequence number. Once the acknowledgment is received, it can delete the message from its send queue.

The receiving site discards all messages it receives that have a lower sequence number than the latest stored message from the sending site. The receiving site discards from *received-messages* all but the (number of the) most recent message from each sending site (message can be discarded only after being processed locally).

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of mes-

sages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

22.6 Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Let one of the sites, say s, be down when T_1 is executed and transaction t_2 reads the balance from site s. One can see that the balance at the primary site would be \$110 at the end.

22.7 In remote backup systems all transactions are performed at the primary site and the data is replicated at the remote backup site. The remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.

The distributed systems offer greater availability by having multiple copies of the data at different sites whereas the remote backup systems offer lesser availability at lower cost and execution overhead.

In a distributed system, transaction code runs at all the sites whereas in a remote backup system it runs only at the primary site. The distributed system transactions follow two-phase commit to have the data in consistent state whereas a remote backup system does not follow two-phase commit and avoids related overhead.

22.8 Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Suppose the copy of the balance at one of the sites, say s, is not consistent – due to lazy replication strategy – with the primary copy after transaction T_1 is executed and let transaction T_2 read this copy of the balance. One can see that the balance at the primary site would be \$110 at the end.

22.9 Let us say a cycle $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_m \rightarrow T_i$ exists in the graph built by the controller. The edges in the graph will either be local edges of the from (T_k, T_l) or distributed edges of the form (T_k, T_l, n). Each local edge (T_k, T_l) definitely implies that T_k is waiting for T_l. Since a distributed edge (T_k, T_l, n) is inserted into the graph only if T_k's request has reached T_l and T_l cannot immediately release the lock, T_k is indeed waiting for T_l. Therefore every edge in the cycle indeed represents a transaction waiting for another. For a detailed proof that this imlies a deadlock refer to Stuart et al. [1984].

We now prove the converse implication. As soon as it is discovered that T_k is waiting for T_l:

a. a local edge (T_k, T_l) is added if both are on the same site.

b. The edge (T_k, T_l, n) is added in both the sites, if T_k and T_l are on different sites.

Therefore, if the algorithm were able to collect all the local wait-for graphs at the same instant, it would definitely discover a cycle in the constructed graph, in case there is a circular wait at that instant. If there is a circular wait at the instant when the algorithm began execution, none of the edges participating in that cycle can disappear until the algorithm finishes. Therefore, even though the algorithm cannot collect all the local graphs at the same instant, any cycle which existed just before it started will anyway be detected.

22.10 **a.** i. Send the query $\Pi_{name}(employee)$ to the Boca plant.
ii. Have the Boca location send back the answer.

b. i. Compute average at New York.
ii. Send answer to San Jose.

c. i. Send the query to find the highest salaried employee to Toronto, Edmonton, Vancouver, and Montreal.
ii. Compute the queries at those sites.
iii. Return answers to San Jose.

d. i. Send the query to find the lowest salaried employee to New York.
ii. Compute the query at New York.
iii. Send answer to San Jose.

22.11 The result is as follows.

$r \ltimes s =$

A	B	C
1	2	3
5	3	2

22.12 The reasons are:

a. Directory access protocols are simplified protocols that cater to a limited type of access to data.

b. Directory systems provide a simple mechanism to name objects in a hierarchical fashion which can be used in a distributed directory system to specify what information is stored in each of the directory servers. The directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

CHAPTER 23

Application Development and Administration

Solutions to Practice Exercises

23.1 No answer.

23.2 No answer.

23.3 No answer.

23.4 **a.** Let there be 100 transactions in the system. The given mix of transaction types would have 25 transactions each of type A and B, and 50 transactions of type C. Thus the time taken to execute transactions only of type A is 0.5 seconds and that for transactions only of type B or only of type C is 0.25 seconds. Given that the transactions do not interfere, the total time taken to execute the 100 transactions is $0.5 + 0.25 + 0.25 = 1$ second. i.e, the average overall transaction throughput is 100 *transactions per second*.

b. One of the most important causes of transaction interference is lock contention. In the previous example, assume that transactions of type A and B are update transactions, and that those of type C are queries. Due to the speed mismatch between the processor and the disk, it is possible that a transaction of type A is holding a lock on a "hot" item of data and waiting for a disk write to complete, while another transaction (possibly of type B or C) is waiting for the lock to be released by A. In this scenario some CPU cycles are wasted. Hence, the observed throughput would be lower than the calculated throughput.

Conversely, if transactions of type A and type B are disk bound, and those of type C are CPU bound, and there is no lock contention, observed throughput may even be better than calculated.

Lock contention can also lead to deadlocks, in which case some transaction(s) will have to be aborted. Transaction aborts and restarts (which may

also be used by an optimistic concurrency control scheme) contribute to the observed throughput being lower than the calculated throughput.

Factors such as the limits on the sizes of data-structures and the variance in the time taken by book-keeping functions of the transaction manager may also cause a difference in the values of the observed and calculated throughput.

23.5 In the absence of an anticipatory standard it may be difficult to reconcile between the differences among products developed by various organizations. Thus it may be hard to formulate a reactionary standard without sacrificing any of the product development effort. This problem has been faced while standardizing pointer syntax and access mechanisms for the ODMG standard.

On the other hand, a reactionary standard is usually formed after extensive product usage, and hence has an advantage over an anticipatory standard - that of built-in pragmatic experience. In practice, it has been found that some anticipatory standards tend to be over-ambitious. SQL-3 is an example of a standard that is complex and has a very large number of features. Some of these features may not be implemented for a long time on any system, and some, no doubt, will be found to be inappropriate.

CHAPTER 24

Advanced Data Types and New Applications

Solutions to Practice Exercises

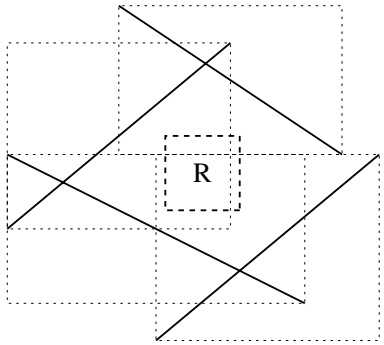
24.1 A temporal database models the changing states of some aspects of the real world. The time intervals related to the data stored in a temporal database may be of two types - *valid time* and *transaction time*. The valid time for a fact is the set of intervals during which the fact is true in the real world. The transaction time for a data object is the set of time intervals during which this object is part of the physical database. Only the transaction time is system dependent and is generated by the database system.

Suppose we consider our sample bank database to be bitemporal. Only the concept of valid time allows the system to answer queries such as - "What was Smith's balance two days ago?". On the other hand, queries such as - "What did we record as Smith's balance two days ago?" can be answered based on the transaction time. The difference between the two times is important. For example, suppose, three days ago the teller made a mistake in entering Smith's balance and corrected the error only yesterday. This error means that there is a difference between the results of the two queries (if both of them are executed today).

24.2 The given query is not a range query, since it requires only searching for a point. This query can be efficiently answered by a B-tree index on the pair of attributes (x, y).

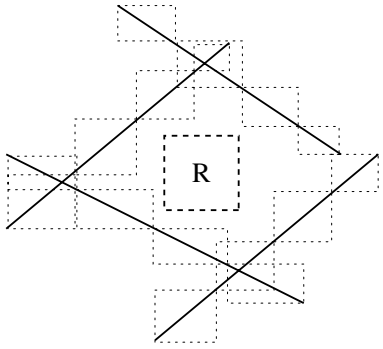
24.3 Suppose that we want to search for the nearest neighbor of a point P in a database of points in the plane. The idea is to issue multiple region queries centered at P. Each region query covers a larger area of points than the previous query. The procedure stops when the result of a region query is non-empty. The distance from P to each point within this region is calculated and the set of points at the smallest distance is reported.

24.4 Large bounding boxes tend to overlap even where the region of overlap does not contain any information. The following figure:



shows a region R within which we have to locate a segment. Note that even though none of the four segments lies in R, due to the large bounding boxes, we have to check each of the four bounding boxes to confirm this.

A significant improvement is observed in the follwoing figure:



where each segment is split into multiple pieces, each with its own bounding box. In the second case, the box R is not part of the boxes indexed by the R-tree. In general, dividing a segment into smaller pieces causes the bounding boxes to be smaller and less wasteful of area.

24.5 Following is a recursive procedure for computing spatial join of two R-trees.

```
SpJoin(node n1, node n2)
begin
    if(the bounding boxes of n1 and n2 do not intersect)
        return;
    if(both n1 and n2 are leaves)
        output all pairs of entries (e1, e2) such that
            e1 ∈ n1 and e2 ∈ n2, and e1 and e2 overlap;
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    for each ns1 in NS1 and ns2 in NS2;
        SpJoin(ns1, ns2);
end
```

24.6 The concepts of RAID can be used to improve reliability of the broadcast of data over wireless systems. Each block of data that is to be broadcast is split into *units* of equal size. A checksum value is calculated for each unit and appended to the unit. Now, parity data for these units is calculated. A checksum for the parity data is appended to it to form a parity unit. Both the data units and the parity unit are then broadcast one after the other as a single transmission.

On reception of the broadcast, the receiver uses the checksums to verify whether each unit is received without error. If one unit is found to be in error, it can be reconstructed from the other units.

The size of a unit must be chosen carefully. Small units not only require more checksums to be computed, but the chance that a burst of noise corrupts more than one unit is also higher. The problem with using large units is that the probability of noise affecting a unit increases; thus there is a tradeoff to be made.

24.7 We can distinguish two models of broadcast data. In the case of a pure broadcast medium, where the receiver cannot communicate with the broadcaster, the broadcaster transmits data with periodic cycles of retransmission of the entire data, so that new receivers can catch up with all the broadcast information. Thus, the data is broadcast in a continuous cycle. This period of the cycle can be considered akin to the worst case rotational latency in a disk drive. There is no concept of seek time here. The value for the cycle latency depends on the

application, but is likely to be at least of the order of seconds, which is much higher than the latency in a disk drive.

In an alternative model, the receiver can send requests back to the broadcaster. In this model, we can also add an equivalent of disk access latency, between the receiver sending a request, and the broadcaster receiving the request and responding to it. The latency is a function of the volume of requests and the bandwidth of the broadcast medium. Further, queries may get satisfied without even sending a request, since the broadcaster happened to send the data either in a cycle or based on some other receivers request. Regardless, latency is likely to be at least of the order of seconds, again much higher than the corresponding values for a hard disk.

A typical hard disk can transfer data at the rate of 1 to 5 megabytes per second. In contrast, the bandwidth of a broadcast channel is typically only a few kilobytes per second. Total latency is likely to be of the order of seconds to hundreds or even thousands of seconds, compared to a few milliseconds for a hard disk.

24.8 Let C be the computer onto which the central database is loaded. Each mobile computer (host) i stores, with its copy of each document d, a version-vector – that is a set of version numbers $V_{d,i,j}$, with one entry for each other host j that stores a copy of the document d, which it could possibly update.

Host A updates document 1 while it is disconnected from C. Thus, according to the version vector scheme, the version number $V_{1,A,A}$ is incremented by one.

Now, suppose host A re-connects to C. This pair exchanges version-vectors and finds that the version number $V_{1,A,A}$ is greater than $V_{1,C,A}$ by 1, (assuming that the copy of document 1 stored host A was updated most recently only by host A). Following the version-vector scheme, the version of document 1 at C is updated and the change is reflected by an increment in the version number $V_{1,C,A}$. Note that these are the only changes made by either host.

Similarly, when host B connects to host C, they exchange version-vectors, and host B finds that $V_{1,B,A}$ is one less than $V_{1,C,A}$. Thus, the version number $V_{1,B,A}$ is incremented by one, and the copy of document 1 at host B is updated.

Thus, we see that the version-vector scheme ensures proper updating of the central database for the case just considered. This argument can be very easily generalized for the case where multiple off-line updates are made to copies of document 1 at host A as well as host B and host C. The argument for off-line updates to document 2 is similar.

CHAPTER 25

Advanced Transaction Processing

Solutions to Practice Exercises

25.1 **a.** The tasks in a workflow have dependencies based on their status. For example the starting of a task may be conditional on the outcome (such as commit or abort) of some other task. All the tasks cannot execute independently and concurrently, using 2PC just for atomic commit.

b. Once a task gets over, it will have to expose its updates, so that other tasks running on the same processing entity don't have to wait for long. 2PL is too strict a form of concurrency control, and is not appropriate for workflows.

c. Workflows have their own consistency requirements; that is, failure-atomicity. An execution of a workflow must finish in an *acceptable termination state*. Because of this, and because of early exposure of uncommitted updates, the recovery procedure will be quite different. Some form of logical logging and compensation transactions will have to be used. Also to perform forward recovery of a failed workflow, the recovery routines need to restore the state information of the scheduler and tasks, not just the updated data items. Thus simple WAL cannot be used.

25.2

- Loading the entire database into memory in advance can provide transactions which need high-speed or realtime data access the guarantee that once they start they will not have to wait for disk accesses to fetch data. However no transaction can run till the entire database is loaded.
- The advantage in loading on demand is that transaction processing can start rightaway; however transactions may see long and unpredictable delays in disk access until the entire database is loaded into memory.

25.3 A high-performance system is not necessarily a real-time system. In a high performance system, the main aim is to execute each transaction as quickly as

possible, by having more resources and better utilization. Thus average speed and response time are the main things to be optimized. In a real-time system, speed is not the central issue. Here *each* transaction has a deadline, and taking care that it finishes within the deadline or takes as little extra time as possible, is the critical issue.

25.4 In the presence of long-duration transactions, trying to ensure serializability has several drawbacks:-

a. With a waiting scheme for concurrency control, long-duration transactions will force long waiting times. This means that response time will be high, concurrency will be low, so throughput will suffer. The probability of deadlocks is also increased.

b. With a time-stamp based scheme, a lot of work done by a long-running transaction will be wasted if it has to abort.

c. Long duration transactions are usually interactive in nature, and it is very difficult to enforce serializability with interactiveness.

Thus the serializability requirement is impractical. Some other notion of database consistency has to be used in order to support long duration transactions.

25.5 Each thread can be modeled as a transaction T which takes a message from the queue and delivers it. We can write transaction T as a multilevel transaction with subtransactions T_1 and T_2. Subtransaction T_1 removes a message from the queue and subtransaction T_2 delivers it. Each subtransaction releases locks once it completes, allowing other transactions to access the queue. If transaction T_2 fails to deliver the message, transaction T_1 will be undone by invoking a compensating transaction which will restore the message to the queue.

25.6 Consider the advanced recovery algorithm of Section 17.8. The redo pass, which repeats history, is the same as before. We discuss below how the undo pass is handled.

- **Recovery with nested transactions**:

  Each subtransaction needs to have a unique TID, because a failed subtransaction might have to be independently rolled back and restarted.

  If a subtransaction fails, the recovery actions depend on whether the unfinished upper-level transaction should be aborted or continued. If it should be aborted, all finished and unfinished subtransactions are undone by a backward scan of the log (this is possible because the locks on the modified data items are not released as soon as a subtransaction finishes). If the nested transaction is going to be continued, just the failed transaction is undone, and then the upper-level transaction continues.

  In the case of a system failure, depending on the application, the entire nested-transaction may need to be aborted, or, (for e.g., in the case of long duration transactions) incomplete subtransactions aborted, and the nested transaction resumed. If the nested-transaction must be aborted, the rollback can be done in the usual manner by the recovery algorithm, during the undo pass. If the nested-transaction must be restarted, any incomplete

subtransactions that need to be rolled back can be rolled back as above. To restart the nested-transaction, state information about the transaction, such as locks held and execution state, must have been noted on the log, and must restored during recovery. Mini-batch transactions (discussed in Section 23.1.8) are an example of nested transactions that must be restarted.

- **Recovery with multi-level transactions**:
  In addition to what is done in the previous case, we have to handle the problems caused by exposure of updates performed by committed subtransactions of incomplete upper-level transactions. A committed subtransaction may have released locks that it held, so the compensating transaction has to reacquire the locks. This is straightforward in the case of transaction failure, but is more complicated in the case of system failure.

  The problem is, a lower level subtransaction a of a higher level transaction A may have released locks, which have to be reacquired to compensate A during recovery. Unfortunately, there may be some other lower level subtransaction b of a higher level transaction B that started and acquired the locks released by a, before the end of A. Thus undo records for b may precede the operation commit record for A. But if b had not finished at the time of the system failure, it must first be rolled back and its locks released, to allow the compensating transaction of A to reacquire the locks.

  This complicates the undo pass; it can no longer be done in one backward scan of the log. Multilevel recovery is described in detail in David Lomet, "MLR: A Recovery Method for Multi-Level Systems", ACM SIGMOD Conf. on the Management of Data 1992, San Diego.

25.7 **a.** We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the co-ordinator, and requiring that the lock be requested on the data item which resides on the currently elected co-ordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

25.8 **a.** The same system as in the answer to Exercise 25.7 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.

Contents

Chapter 3 SQL

Chapter 4 Advanced SQL

Chapter 5 Other Relational Languages

PART 2 ■ DATABASE DESIGN

Chapter 6 Database Design and the E-R Model

Chapter 7 Relational Database Design

Chapter 8 Application Design and Development

PART 3 ■ OBJECT-BASED DATABASES AND XML

Chapter 9 Object-Based Databases

Chapter 10 XML

PART 4 ■ DATA STORAGE AND QUERYING

Chapter 11 Storage and File Structure

Chapter 12 Indexing and Hashing

Chapter 13 Query Processing

Chapter 14 Query Optimization

PART 5 ■ TRANSACTION MANAGEMENT

Chapter 15 Transactions

Chapter 16 Concurrency Control

Chapter 17 Recovery System

PART 6 ■ DATA MINING AND INFORMATION RETRIEVAL

Chapter 18 Data Analysis and Mining

Chapter 19 Information Retrieval

PART 7 ■ SYSTEM ARCHITECTURE

Chapter 20 Database-System Architectures

Chapter 21 Parallel Databases

Chapter 22 Distributed Databases

PART 8 ■ OTHER TOPICS

Chapter 23 Advanced Application Development

Chapter 24 Advanced Data Types and New Applications

Chapter 25 Advanced Transaction Processing

PART 9 ■ CASE STUDIES

Chapter 26 PostgreSQL

Chapter 27 Oracle

Chapter 28 IBM DB2 Universal Database

Chapter 29 Microsoft SQL Server

PART 10 ■ APPENDICES

Appendix A Network Model (contents online)

Appendix B Hierarchical Model (contents online)

Appendix C Advanced Relational Database Design (contents online)

Bibliography 1101

Index 1129

INSTRUCTOR'S MANUAL TO ACCOMPANY

Database System Concepts

Fifth Edition

Abraham Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
Indian Institute of Technology, Bombay

Contents

Preface

This volume is an instructor's manual for the 5^{th} edition of *Database System Concepts* by Abraham Silberschatz, Henry F. Korth and S. Sudarshan. It contains answers to the exercises at the end of each chapter of the book. Before providing answers to the exercises for each chapter, we include a few remarks about the chapter. The nature of these remarks vary. They include explanations of the inclusion or omission of certain material, and remarks on how we teach the chapter in our own courses. The remarks also include suggestions on material to skip if time is at a premium, and tips on software and supplementary material that can be used for programming exercises.

The Web home page of the book, at http://www.db-book.com, contains a variety of useful information, including up-to-date errata, online appendices describing the network data model, the hierarchical data model, and advanced relational database design, and model course syllabi. We will periodically update the page with supplementary material that may be of use to teachers and students.

We provide a mailing list through which users can communicate among themselves and with us. If you wish to use this facility, please visit the following URL and follow the instructions there to subscribe:

http://mailman.cs.yale.edu/mailman/listinfo/db-book-list

The mailman mailing list system provides many benefits, such as an archive of postings, and several subscription options, including digest and Web only. To send messages to the list, send e-mail to:

db-book-list@cs.yale.edu

We would appreciate it if you would notify us of any errors or omissions in the book, as well as in the instructor's manual. Internet electronic mail should be addressed to db-book@cs.yale.edu. Physical mail may be sent to Avi Silberschatz, Yale University, 51 Prospect Street, New Haven, CT, 06520, USA.

Although we have tried to produce an instructor's manual which will aid all of the users of our book as much as possible, there can always be improvements. These could include improved answers, additional questions, sample test questions, programming projects, suggestions on alternative orders of presentation of the material, additional references, and so on. If you would like to suggest any such improvements to the book or the instructor's manual, we would be glad to hear from you. All contributions that we make use of will, of course, be properly credited to their contributor.

This manual is derived from the manuals for the earlier editions. The manual for the 4^{th} edition was prepared by Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia and Arvind Hulgeri. The manual for the 3^{rd} edition was prepared by K. V. Raghavan with help from Prateek R. Kapadia. Sara Strandtman helped with the instructor manual for the 2^{nd} and 3^{rd} editions, while Greg Speegle and Dawn Bezviner helped us to prepare the instructor's manual for the 1^{st} edition.

A. S.
H. F. K.
S. S.

Instructor Manual Version 5.0.0

CHAPTER 1

Introduction

Exercises

1.5 List four applications which you have used, that most likely used a database system to store persistent data.
Answer: No answer

1.6 List four significant differences between a file-processing system and a DBMS.
Answer: Some main differences between a database management system and a file-processing system are:

- Both systems contain a collection of data and a set of programs which access that data. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.
- A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, whereas data written by one program in a file-processing system may not be readable by another program.
- A database management system is designed to allow flexible access to data (i.e., queries), whereas a file-processing system is designed to allow predetermined access to data (i.e., compiled programs).
- A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file.

1.7 Explain the difference between physical and logical data independence.
Answer:

- Physical data independence is the ability to modify the physical scheme without making it necessary to rewrite application programs. Such modifications include changing from unblocked to blocked record storage, or from sequential to random access files.
- Logical data independence is the ability to modify the conceptual scheme without making it necessary to rewrite application programs. Such a modification might be adding a field to a record; an application program's view hides this change from the program.

1.8 List five responsibilities of a database management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.
Answer: A general purpose database manager (DBM) has five responsibilities:

a. interaction with the file manager.
b. integrity enforcement.
c. security enforcement.
d. backup and recovery.
e. concurrency control.

If these responsibilities were not met by a given DBM (and the text points out that sometimes a responsibility is omitted by design, such as concurrency control on a single-user DBM for a micro computer) the following problems can occur, respectively:

a. No DBM can do without this, if there is no file manager interaction then nothing stored in the files can be retrieved.
b. Consistency constraints may not be satisfied, account balances could go below the minimum allowed, employees could earn too much overtime (e.g., hours > 80) or, airline pilots may fly more hours than allowed by law.
c. Unauthorized users may access the database, or users authorized to access part of the database may be able to access parts of the database for which they lack authority. For example, a high school student could get access to national defense secret codes, or employees could find out what their supervisors earn.
d. Data could be lost permanently, rather than at least being available in a consistent state that existed prior to a failure.
e. Consistency constraints may be violated despite proper integrity enforcement in each transaction. For example, incorrect bank balances might be reflected due to simultaneous withdrawals and deposits, and so on.

1.9 List at least two reasons why database systems support data manipulation using a declarative query language such as SQL, instead of just providing a a library of C or C++ functions to carry out data manipulation.
Answer: No answer

1.10 Explain what problems are caused by the design of the table in Figure 1.5.
Answer: No answer

1.11 What are five main functions of a database administrator?

Answer: Five main functions of a database administrator are:

- To create the scheme definition
- To define the storage structure and access methods
- To modify the scheme and/or physical organization when necessary
- To grant authorization for data access
- To specify integrity constraints

CHAPTER 2

Relational Model

Exercises

2.4 Describe the differences in meaning between the terms *relation* and *relation schema*.

Answer: A relation schema is a type definition, and a relation is an instance of that schema. For example, *student* (*ss#, name*) is a relation schema and

ss#	*name*
123-45-6789	Tom Jones
456-78-9123	Joe Brown

is a relation based on that schema.

2.5 Consider the relational database of Figure 2.35, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who work for First Bank Corporation.
b. Find the names and cities of residence of all employees who work for First Bank Corporation.
c. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
d. Find the names of all employees in this database who live in the same city as the company for which they work.
e. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

Answer:

a. $\Pi_{person\text{-}name}\ (\sigma_{company\text{-}name\ =}$ "First Bank Corporation" $(works))$

employee (*person-name*, *street*, *city*)
works (*person-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*person-name*, *manager-name*)

Figure 2.35. Relational database for Exercises 2.1, 2.3 and 2.9.

b. $\Pi_{person\text{-}name,\, city}\ (employee \bowtie (\sigma_{company\text{-}name\, =}$ "First Bank Corporation" $(works)))$

c. $\Pi_{person\text{-}name,\ street,\ city}\ (\sigma_{(company\text{-}name\, =}$ "First Bank Corporation" $\wedge\ salary > 10000)\ works \bowtie employee)$

d. $\Pi_{person\text{-}name}\ (employee \bowtie works \bowtie company)$

e. Note: Small Bank Corporation will be included in each answer.

$$\Pi_{company\text{-}name}\ (company \div (\Pi_{city}\ (\sigma_{company\text{-}name\, =}\text{ "Small Bank Corporation" }(company))))$$

2.6 Consider the relation of Figure 2.20, which shows the result of the query "Find the names of all customers who have a loan at the bank." Rewrite the query to include not only the name, but also the city of residence for each customer. Observe that now customer Jackson no longer appears in the result, even though Jackson does in fact have a loan from the bank.

a. Explain why Jackson does not appear in the result.

b. Suppose that you want Jackson to appear in the result. How would you modify the database to achieve this effect?

c. Again, suppose that you want Jackson to appear in the result. Write a query using an outer join that accomplishes this desire without your having to modify the database.

Answer: The rewritten query is

$\Pi_{customer\text{-}name,customer\text{-}city,amount}(borrower \bowtie loan \bowtie customer)$

a. Although Jackson does have a loan, no address is given for Jackson in the *customer* relation. Since no tuple in *customer* joins with the Jackson tuple of *borrower*, Jackson does not appear in the result.

b. The best solution is to insert Jackson's address into the *customer* relation. If the address is unknown, null values may be used. If the database system does not support nulls, a special value may be used (such as **unknown**) for Jackson's street and city. The special value chosen must not be a plausible name for an actual city or street.

c. $\Pi_{customer\text{-}name,customer\text{-}city,amount}((borrower \bowtie loan) \supset\!\bowtie customer)$

2.7 Consider the relational database of Figure 2.35. Give an expression in the relational algebra for each request:

a. Give all employees of First Bank Corporation a 10 percent salary raise.

b. Give all managers in this database a 10 percent salary raise, unless the salary would be greater than $100,000. In such cases, give only a 3 percent raise.
c. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

Answer:

a. $works \leftarrow \Pi_{person\text{-}name, company\text{-}name, 1.1*salary}($
$\sigma_{(company\text{-}name=\text{“First Bank Corporation”})}(works))$
$\cup (works - \sigma_{company\text{-}name=\text{“First Bank Corporation”}}(works))$

b. The same situation arises here. As before, t_1, holds the tuples to be updated and t_2 holds these tuples in their updated form.

$t_1 \leftarrow \Pi_{works.person\text{-}name, company\text{-}name, salary}$
$(\sigma_{works.person\text{-}name=manager\text{-}name}(works \times manages))$

$t_2 \leftarrow \Pi_{works.person\text{-}name, company\text{-}name, salary*1.03}$
$(\sigma_{t_1.salary\ *\ 1.1\ >\ 100000}(t_1))$

$t_2 \leftarrow t_2 \cup (\Pi_{works.person\text{-}name, company\text{-}name, salary*1.1}$
$(\sigma_{t_1.salary\ *\ 1.1\ \leq\ 100000}(t_1)))$

$works \leftarrow (works - t_1) \cup t_2$

c. $works \leftarrow works - \sigma_{company-name=\text{“Small Bank Corporation”}}(works)$

2.8 Using the bank example, write relational-algebra queries to find the accounts held by more than two customers in the following ways:

a. Using an aggregate function.
b. Without using any aggregate functions.

Answer:

a. $t_1 \leftarrow {}_{account\text{-}number}\mathcal{G}_{\textbf{count}\ customer\text{-}name}(depositor)$
$\Pi_{account\text{-}number}\left(\sigma_{num\text{-}holders>2}\left(\rho_{account\text{-}holders(account\text{-}number, num\text{-}holders)}(t_1)\right)\right)$

b. $t_1 \leftarrow (\rho_{d1}(depositor) \times \rho_{d2}(depositor) \times \rho_{d3}(depositor))$
$t_2 \leftarrow \sigma_{(d1.account\text{-}number=d2.account\text{-}number=d3.account\text{-}number)}(t_1)$
$\Pi_{d1.account\text{-}number}(\sigma_{(d1.customer\text{-}name\neq d2.customer\text{-}name\ \wedge}$
$_{d2.customer\text{-}name\neq d3.customer\text{-}name\ \wedge d3.customer\text{-}name\neq d1.customer\text{-}name)}(t_2))$

2.9 Consider the relational database of Figure 2.35. Give a relational-algebra expression for each of the following queries:

a. Find the company with the most employees.
b. Find the company with the smallest payroll.
c. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

Answer:

a. $t_1 \leftarrow {}_{company\text{-}name}\mathcal{G}_{\textbf{count-distinct}\ person\text{-}name}(works)$
$t_2 \leftarrow \mathbf{max}_{num\text{-}employees}(\rho_{company\text{-}strength(company\text{-}name,num\text{-}employees)}(t_1))$
$\Pi_{company\text{-}name}(\rho_{t_3(company\text{-}name,num\text{-}employees)}(t_1) \bowtie \rho_{t_4(num\text{-}employees)}(t_2))$

b. $t_1 \leftarrow {}_{company\text{-}name}\mathcal{G}_{\textbf{sum}\ salary}(works)$
$t_2 \leftarrow \mathbf{min}_{payroll}(\rho_{company\text{-}payroll(company\text{-}name,payroll)}(t_1))$
$\Pi_{company\text{-}name}(\rho_{t_3(company\text{-}name,payroll)}(t_1) \bowtie \rho_{t_4(payroll)}(t_2))$

c. $t_1 \leftarrow {}_{company\text{-}name}\mathcal{G}_{\textbf{avg}\ salary}(works)$
$t_2 \leftarrow \sigma_{company\text{-}name\ =\ \text{“First Bank Corporation”}}(t_1)$
$\Pi_{t_3.company\text{-}name}((\rho_{t_3(company\text{-}name,avg\text{-}salary)}(t_1))$
$\bowtie_{t_3.avg\text{-}salary\ >\ first\text{-}bank.avg\text{-}salary}\ (\rho_{first\text{-}bank(company\text{-}name,avg\text{-}salary)}(t_2)))$

2.10 List two reasons why null values might be introduced into the database.
Answer: Nulls may be introduced into the database because the actual value is either unknown or does not exist. For example, an employee whose address has changed and whose new address is not yet known should be retained with a null address. If employee tuples have a composite attribute *dependents*, and a particular employee has no dependents, then that tuple's *dependents* attribute should be given a null value.

2.11 Consider the following relational schema

employee(*empno*, *name*, *office*, *age*)
books(*isbn*, *title*, *authors*, *publishe r*)
loan(*empno*, *isbn*, *date*)

Write the following queries in relational algebra.

a. Find the names of employees who have borrowed a book published by McGraw-Hill.
b. Find the names of employees who have borrowed all books published by McGraw-Hill.
c. Find the names of employees who have borrowed more than five different books published by McGraw-Hill.
d. For each publisher, find the names of employees who have borrowed more than five books of that publisher.

Answer: No answer

CHAPTER 3

SQL

Exercises

3.8 Consider the insurance database of Figure 3.11, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the number of accidents in which the cars belonging to "John Smith" were involved.

b. Update the damage amount for the car with license number "AABB2000" in the accident with report number "AR2197" to $3000.

Answer: Note: The *participated* relation relates drivers, cars, and accidents.

a. SQL query:

```
select   count (distinct *)
from     accident
where    exists
         (select *
          from participated, person
          where participated.driver_id = person.driver_id
                and person.name = 'John Smith'
                and accident.report_number = participated.report_number)
```

b. SQL query:

```
update participated
set damage_amount = 3000
where report_number = "AR2197" and driver_id in
      (select driver_id
       from owns
       where license = "AABB2000")
```

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*driver_id*, *car*, *report_number*, *damage_amount*)

Figure 3.11. Insurance database.

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 3.12. Employee database.

3.9 Consider the employee database of Figure 3.12, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names of all employees who work for First Bank Corporation.
b. Find all employees in the database who live in the same cities as the companies for which they work.
c. Find all employees in the database who live in the same cities and on the same streets as do their managers.
d. Find all employees who earn more than the average salary of all employees of their company.
e. Find the company that has the smallest payroll.

Answer:

a. Find the names of all employees who work for First Bank Corporation.

```
select employee_name
from works
where company_name = 'First Bank Corporation'
```

b. Find all employees in the database who live in the same cities as the companies for which they work.

```
select e.employee_name
from employee e, works w, company c
where e.employee_name = w.employee_name and e.city = c.city and
      w.company_name = c.company _name
```

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

```
select P.employee_name
from employee P, employee R, manages M
where P.employee_name = M.employee_name and
      M.manager_name = R.employee_name and
      P.street = R.street and P.city = R.city
```

d. Find all employees who earn more than the average salary of all employees of their company.

The following solution assumes that all people work for at most one company.

```
select employee_name
from works T
where salary > (select avg (salary)
                from works S
                where T.company_name = S.company_name)
```

e. Find the company that has the smallest payroll.

```
select company_name
from works
group by company_name
having sum (salary) <= all (select sum (salary)
                            from works
                            group by company_name)
```

3.10 Consider the relational database of Figure 3.12. Give an expression in SQL for each of the following queries.

a. Give all employees of First Bank Corporation a 10 percent raise.
b. Give all managers of First Bank Corporation a 10 percent raise.
c. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

Answer:

a. Give all employees of First Bank Corporation a 10-percent raise. (the solution assumes that each person works for at most one company.)

```
update works
set salary = salary * 1.1
where company_name = 'First Bank Corporation'
```

b. Give all managers of First Bank Corporation a 10-percent raise.

```
update works
set salary = salary * 1.1
where employee_name in (select manager_name
                        from manages)
      and company_name = 'First Bank Corporation'
```

c. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

```
delete works
where company_name = 'Small Bank Corporation'
```

3.11 Let the following relation schemas be given:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Let relations $r(R)$ and $s(S)$ be given. Give an expression in SQL that is equivalent to each of the following queries.

a. $\Pi_A(r)$
b. $\sigma_{B=17}(r)$
c. $r \times s$
d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Answer:

a. $\Pi_A(r)$

```
select distinct A
from r
```

b. $\sigma_{B=17}(r)$

```
select *
from r
where B = 17
```

c. $r \times s$

```
select distinct *
from r, s
```

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

```
select distinct A, F
from r, s
where C = D
```

3.12 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in SQL that is equivalent to each of the following queries.

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

Answer:

a. $r_1 \cup r_2$

```
(select *
 from r1)
union
(select *
 from r2)
```

b. $r_1 \cap r_2$

We can write this using the **intersect** operation, which is the preferred approach, but for variety we present an solution using a nested subquery.

```
select *
from r1
where (A, B, C) in (select *
                    from r2)
```

c. $r_1 - r_2$

```
select *
from r1
where (A, B, C) not in (select *
                        from r2)
```

This can also be solved using the **except** clause.

d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

```
select r1.A, r2.B, r3.C
from r1, r2
where r1.B = r2.B
```

3.13 Show that, in SQL, $<>$ **all** is identical to **not in**.
Answer: Let the set S denote the result of an SQL subquery. We compare $(x <>$ **all** $S)$ with $(x$ **not in** $S)$. If a particular value x_1 satisfies $(x_1 <>$ **all** $S)$ then for all elements y of S $x_1 \neq y$. Thus x_1 is not a member of S and must satisfy $(x_1$ **not in** $S)$. Similarly, suppose there is a particular value x_2 which satisfies $(x_2$ **not in** $S)$. It cannot be equal to any element w belonging to S, and hence $(x_2 <>$ **all** $S)$ will be satisfied. Therefore the two expressions are equivalent.

3.14 Consider the relational database of Figure 3.12. Using SQL, define a view consisting of *manager_name* and the average salary of all employees who work for that manager. Explain why the database system should not allow updates to be expressed in terms of this view.
Answer:

```
create view salinfo as
        select manager_name, avg(salary)
        from manages m, works w
        where m.employee_name = w.employee_name
        group by manager_name
```

Updates should not be allowed in this view because there is no way to determine how to change the underlying data. For example, suppose the request is "change the average salary of employees working for Smith to $200". Should everybody who works for Smith have their salary changed to $200? Or should the first (or more, if necessary) employee found who works for Smith have their salary adjusted so that the average is $200? Neither approach really makes sense.

3.15 Write an SQL query, without using a **with** clause, to find all branches where the total account deposit is less than the average total account deposit at all branches,

a. Using a nested query in the **from** clauser.
b. Using a nested query in a **having** clause.

Answer: We output the branch names along with the total account deposit at the branch.

a. Using a nested query in the **from** clauser.

```
select branch_name, tot_balance
from (select branch_name, sum (balance)
      from account
      group by branch_name) as branch_total(branch_name, tot_balance)
where tot_balance ¡
      ( select avg (tot_balance)
      from ( select branch_name, sum (balance)
            from account
            group by branch_name) as branch_total(branch_name, tot_balance)
      )
```

b. Using a nested query in a **having** clause.

```
select branch_name, sum (balance)
from account
group by branch_name
having sum (balance) ¡
      ( select avg (tot_balance)
      from ( select branch_name, sum (balance)
            from account
            group by branch_name) as branch_total(branch_name, tot_balance)
      )
```

3.16 List two reasons why null values might be introduced into the database.
Answer: No Answer

3.17 Show how to express the **coalesce** operation from Exercise 3.4 using the **case** operation.
Answer: No Answer.

3.18 Give an SQL schema definition for the employee database of Figure 3.12. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.
Answer:

```
create domain   company_names char(20)
create domain   city_names char(30)
create domain   person_names char(20)

create table    employee
```

(*employee_name* *person_names*,
street **char**(30),
city *city_names*,
primary key (*employee_name*))

create table *works*
(*employee_name* *person_names*,
company_name *company_names*,
salary **numeric**(8, 2),
primary key (*employee_name*))

create table *company*
(*company_name* *company_names*,
city *city_names*,
primary key (*company_name*))

create table *manages*
(*employee_name* *person_names*,
manager_name *person_names*,
primary key (*employee_name*))

3.19 Using the relations of our sample bank database, write SQL expressions to define the following views:

a. A view containing the account numbers and customer names (but not the balances) for all accounts at the Deer Park branch.
b. A view containing the names and addresses of all customers who have an account with the bank, but do not have a loan.
c. A view containing the name and average account balance of every customer of the Rock Ridge branch.

Answer: No Answer.

3.20 For each of the views that you defined in Exercise 3.19, explain how updates would be performed (if they should be allowed at all).
Answer: No Answer.

3.21 Consider the following relational schema

employee(*empno*, *name*, *office*, *age*)
books(*isbn*, *title*, *authors*, *publisher*)
loan(*empno*, *isbn*, *date*)

Write the following queries in SQL.

a. Print the names of employees who have borrowed any book published by McGraw-Hill.

b. Print the names of employees who have borrowed all books published by McGraw-Hill.
c. For each publisher, print the names of employees who have borrowed more than five books of that publisher.

Answer: No Answer.

3.22 Consider the relational schema

student(*student_id*, *student_name*)
registered(*student_id*, *course_id*)

Write an SQL query to list the student-id and name of each student along with the total number of courses that the student is registered for. Students who are not registered for any course must also be listed, with the number of registered courses shown as 0.
Answer: No Answer.

3.23 Suppose that we have a relation *marks*(*student_id*, *score*). Write an SQL query to find the *dense rank* of each student. That is, all students with the top mark get a rank of 1, those with the next highest mark get a rank of 2, and so on. Hint: Split the task into parts, using the **with** clause.
Answer: No Answer.

CHAPTER 4

Advanced SQL

Exercises

4.7 Referential-integrity constraints as defined in this chapter involve exactly two relations. Consider a database that includes the following relations:

salaried-worker (*name, office, phone, salary*)
hourly-worker (*name, hourly-wage*)
address (*name, street, city*)

Suppose that we wish to require that every name that appears in *address* appear in either *salaried-worker* or *hourly-worker*, but not necessarily in both.

a. Propose a syntax for expressing such constraints.
b. Discuss the actions that the system must take to enforce a constraint of this form.

Answer:

a. For simplicity, we present a variant of the SQL syntax. As part of the **create table** expression for *address* we include

foreign key (*name*) **references** *salaried-worker* or *hourly-worker*

b. To enforce this constraint, whenever a tuple is inserted into the *address* relation, a lookup on the *name* value must be made on the *salaried-worker* relation and (if that lookup failed) on the *hourly-worker* relation (or vice-versa).

4.8 Write a Java function using JDBC metadata features that takes a ResultSet as an input parameter, and prints out the result in tabular form, with appropriate names as column headings.
Answer: No Answer.

4.9 Write a Java function using JDBC metadata features that prints a list of all relations in the database, displaying for each relation the names and types of its attributes.
Answer: No Answer.

4.10 Consider an employee database with two relations

employee (*employee-name*, *street*, *city*)
works (*employee-name*, *company-name*, *salary*)

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

a. Using SQL functions as appropriate.
b. Without using SQL functions.

Answer:

a.
```
create function avg-salary(cname varchar(15))
    returns integer
    declare result integer;
        select avg(salary) into result
        from works
        where works.company-name = cname
    return result;
end
select company-name
from works
where avg-salary(company-name) > avg-salary("First Bank Corporation")
```

b.
```
select company-name
from works
group by company-name
having avg(salary) > (select avg(salary)
                      from works
                      where company-name="First Bank Corporation")
```

4.11 Rewrite the query in Section 4.6.1 that returns the name, street and city of all customers with more than one account, using the **with** clause instead of using a function call.
Answer: No Answer.

4.12 Compare the use of embedded SQL with the use in SQL of functions defined in a general-purpose programming language. Under what circumstances would you use each of these features?
Answer: SQL functions are primarily a mechanism for extending the power of SQL to handle attributes of complex data types (like images), or to perform complex and non-standard operations. Embedded SQL is useful when imperative actions like displaying results and interacting with the user are needed.

These cannot be done conveniently in an SQL only environment. Embedded SQL can be used instead of SQL functions by retrieving data and then performing the function's operations on the SQL result. However a drawback is that a lot of query-evaluation functionality may end up getting repeated in the host language code.

4.13 Modify the recursive query in Figure 4.14 to define a relation

empl_depth(*employee_name*, *manager_name*, *depth*)

where the attribute *depth* indicates how many levels of intermediate managers are there between the employee and the manager. Employees who are directly under a manager would have a depth of 0.
Answer: No Answer.

4.14 Consider the relational schema

part(*part_id*, *name*, *cost*)
subpart(*part_id*, *subpart_id*, *count*)

A tuple $(p_1, p_2, 3)$ in the *subpart* relation denotes that the part with part-id p_2 is a direct subpart of the part with part-id p_1, and p_1 has 3 copies of p_2 Note that p_2 may itself have further subparts. Write a recursive SQL query that outputs the names of all subparts of the part with part-id "P-100".
Answer: No Answer.

4.15 Consider again the relational schema from Exercise 4.14. Write a JDBC function using non-recursive SQL to find the total cost of part "P-100", including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.
Answer: No Answer.

CHAPTER 5

Other Relational Languages

Exercises

5.6 Consider the employee database of Figure 5.14. Give expressions in tuple relational calculus and domain relational calculus for each of the following queries:

a. Find the names of all employees who work for First Bank Corporation.
b. Find the names and cities of residence of all employees who work for First Bank Corporation.
c. Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
d. Find all employees who live in the same city as that in which the company for which they work is located.
e. Find all employees who live in the same city and on the same street as their managers.
f. Find all employees in the database who do not work for First Bank Corporation.
g. Find all employees who earn more than every employee of Small Bank Corporation.
h. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

Answer:

a. Find the names of all employees who work for First Bank Corporation:

i. $\{t \mid \exists\, s \in works\ (t[person\_name] = s[person\_name] \wedge s[company\_name] =$ "First Bank Corporation"$)\}$

ii. $\{ < p > \mid \exists\, c,\ s\ (< p, c, s > \in works \wedge c =$ "First Bank Corporation"$)\}$

b. Find the names and cities of residence of all employees who work for First Bank Corporation:

i. $\{t \mid \exists\, r \in employee\, \exists\, s \in works\, (\, t[person\_name] = r[person\_name] \wedge t[city] = r[city] \wedge r[person\_name] = s[person\_name] \wedge s[company\_name] =$ 'First Bank Corporation'$)\}$

ii. $\{< p, c > \mid \exists\, co,\ sa,\ st\, (< p, co, sa > \in works \wedge < p, st, c > \in employee \wedge co =$ "First Bank Corporation"$)\}$

c. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum:

i. $\{t \mid t \in employee \wedge (\exists\, s \in works\, (\, s[person\_name] = t[person\_name] \wedge s[company\_name] =$ "First Bank Corporation" $\wedge\ s[salary] > 10000))\}$

ii. $\{< p, s, c > \mid < p, s, c > \in employee \wedge \exists\, co,\ sa\, (< p, co, sa > \in works \wedge co =$ 'First Bank Corporation' $\wedge\ sa > 10000)\}$

d. Find the names of all employees in this database who live in the same city as the company for which they work:

i. $\{t \mid \exists\, e \in employee\, \exists\, w \in works\, \exists\, c \in company\ (t[person\_name] = e[person\_name] \wedge e[person\_name] = w[person\_name] \wedge w[company\_name] = c[company\_name] \wedge e[city] = c[city])\}$

ii. $\{< p > \mid \exists\, st,\ c,\ co,\ sa\, (< p, st, c > \in employee \wedge < p, co, sa > \in works \wedge < co, c > \in company)\}$

e. Find the names of all employees who live in the same city and on the same street as do their managers:

i. $\{\, t \mid \exists\, l \in employee\, \exists\, m \in manages\, \exists\, r \in employee\ (l[person\_name] = m[person\_name] \wedge m[manager\_name] = r[person\_name] \wedge l[street] = r[street] \wedge l[city] = r[city] \wedge t[person\_name] = l[person\_name])\}$

ii. $\{< t > \mid \exists\, s,\ c,\ m\, (< t, s, c > \in employee \wedge < t, m > \in manages \wedge < m, s, c > \in employee)\}$

f. Find the names of all employees in this database who do not work for First Bank Corporation:
If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, the problem is more complicated. We give solutions for this more realistic case later.

i. $\{\, t \mid \exists\, w \in works\, (\, w[company\_name] \neq$ "First Bank Corporation" $\wedge\ t[person\_name] = w[person\_name])\}$

ii. $\{\, < p > \mid \exists\, c,\ s\, (< p, c, s > \in works \wedge c \neq$ "First Bank Corporation"$)\}$

If people may not work for any company:

i. $\{ t \mid \exists e \in employee (t[person\_name] = e[person\_name] \wedge \neg \exists w \in works$
$(w[company\_name] =$ "First Bank Corporation"
$\wedge w[person\_name] = t[person\_name]))\}$

ii. $\{ < p > \mid \exists s, c (< p, s, c > \in employee) \wedge \neg \exists x, y$
$(y =$ "First Bank Corporation"$\wedge < p, y, x > \in works)\}$

g. Find the names of all employees who earn more than every employee of Small Bank Corporation:

i. $\{t \mid \exists w \in works (t[person\_name] = w[person\_name] \wedge \forall s \in works$
$(s[company\_name] =$ "Small Bank Corporation" $\Rightarrow w[salary] > s[salary]))\}$

ii. $\{< p > \mid \exists c, s (< p, c, s > \in works \wedge \forall p_2, c_2, s_2$
$(< p_2, c_2, s_2 > \notin works \vee c_2 \neq$ "Small Bank Corporation" $\vee s > s_2))\}$

h. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.
Note: Small Bank Corporation will be included in each answer.

i. $\{t \mid \forall s \in company (s[company\_name] =$ "Small Bank Corporation" $\Rightarrow$
$\exists r \in company (t[company\_name] = r[company\_name] \wedge r[city] = s[city]))\}$

ii. $\{< co > \mid \forall co_2, ci_2 (< co_2, ci_2 > \notin company$
$\vee co_2 \neq$ "Small Bank Corporation" $\vee < co, ci_2 > \in company)\}$

5.7 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write relational-algebra expressions equivalent to the following domain-relational-calculus expressions:

a. $\{< a > \mid \exists b (< a, b > \in r \wedge b = 17)\}$

b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

c. $\{< a > \mid \exists b (< a, b > \in r) \vee \forall c (\exists d (< d, c > \in s) \Rightarrow < a, c > \in s)\}$

d. $\{< a > \mid \exists c (< a, c > \in s \wedge \exists b_1, b_2 (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

Answer:

a. $\Pi_A (\sigma_{B = 17} (r))$

b. $r \bowtie s$

c. $\Pi_A(r) \cup (r \div \sigma_B(\Pi_C(s)))$

d. $\Pi_{r.A} ((r \bowtie s) \bowtie_{c = r2.A \wedge r.B > r2.B} (\rho_{r_2}(r)))$

It is interesting to note that (*d*) is an abstraction of the notorious query "Find all employees who earn more than their manager." Let $R = (emp,\ sal)$, $S = (emp,\ mgr)$ to observe this.

5.8 Repeat Exercise 5.7, writing SQL queries instead of relational-algebra expressions.
Answer: No answer.

5.9 Let $R = (A,\ B)$ and $S = (A,\ C)$, and let $r(R)$ and $s(S)$ be relations. Using the special constant *null*, write tuple-relational-calculus expressions equivalent to each of the following:

a. $r \ltimes\sqsubset s$
b. $r \rtimes\sqsubset s$
c. $r \rtimes\ltimes s$

Answer:

a. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee \exists s \in S(\neg\exists r \in R(r[A] = s[A]) \wedge t[A] = s[A] \wedge t[C] = s[C] \wedge t[B] = null)\}$

b. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee \exists r \in R(\neg\exists s \in S(r[A] = s[A]) \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = null) \vee \exists s \in S(\neg\exists r \in R(r[A] = s[A]) \wedge t[A] = s[A] \wedge t[C] = s[C] \wedge t[B] = null)\}$

c. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee \exists r \in R(\neg\exists s \in S(r[A] = s[A]) \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = null)\}$

5.10 Consider the insurance database of Figure 5.15, where the primary keys are underlined. Construct the following GQBE queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 1989.
b. Find the number of accidents in which the cars belonging to "John Smith" were involved.

Answer: No answer.

5.11 Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.
Answer: No answer.

5.12 Repeat Exercise 5.6 using QBE and Datalog.
Answer:

a. Find the names of all employees who work for First Bank Corporation.

i.

works	*person_name*	*company_name*	*salary*
	P._*x*	First Bank Co	

ii. *query(X)* :- *works(X, "First Bank Corporation", Y)*

b. Find the names and cities of residence of all employees who work for First Bank Corporation.

i.

employee	*person_name*	*street*	*city*
	P._*x*		P._*y*

works	*person_name*	*company_name*	*salary*
	_*x*	First Bank Corporation	

ii. $query\,(X,\,Y)$:- $employee\,(X,\,Z,\,Y)$, $works\,(X,$ "First Bank Corporation", $W)$

c. Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.

If people may work for several companies, the following solutions will only list those who earn more than $10,000 per annum from "First Bank Corporation" alone.

i.

employee	*person_name*	*street*	*city*
	P._x	P._y	P._z

works	*person_name*	*company_name*	*salary*
	_x	First Bank Corporation	>10000

ii.

$$query\,(X,\,Y,\,Z)\text{ :- }lives\,(X,\,Y,\,Z),\ works\,(X,\text{ ``First Bank Corporation''},\ W),\ W\ >\ 10000$$

d. Find all employees who live in the city where the company for which they work is located.

i.

employee	*person_name*	*street*	*city*
	P._x		_y

works	*person_name*	*company_name*	*salary*
	_x	_c	

company	*company_name*	*city*
	_c	_y

ii. $query\,(X)$:- $employee\,(X,\,Y,\,Z),\ works\,(X,\,V,\,W),\ company\,(V,\,Z)$

e. Find all employees who live in the same city and on the same street as their managers.

i.

employee	*person_name*	*street*	*city*
	P._x	_s	_c
	_y	_s	_c

manages	*person_name*	*manager_name*
	_x	_y

ii. $query\,(X)$:- $lives\,(X,\,Y,\,Z),\ manages\,(X,\,V),\ lives\,(V,\,Y,\,Z)$

f. Find all employees in the database who do not work for First Bank Corporation.
The following solutions assume that all people work for exactly one company.

i.

works	*person_name*	*company_name*	*salary*
	P._x	First Bank Co	

ii. $query\,(X)$:- $works\,(X,\, Y,\, Z),\; Y \neq$ "First Bank Corporation"

If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, or if people may have jobs with more than one company, the solutions are slightly more complicated. They are given below :

i.

employee	*person_name*	*street*	*city*
	P._x		

works	*person_name*	*company_name*	*salary*
¬	_x	First Bank Corporation	

ii.

$query\,(X)$:- $employee\,(X,\, Y,\, Z),\; \neg\, p1\,(X)$
$p1\,(X)$:- $works(X,$ "First Bank Corporation", $W)$

g. Find all employees who earn more than every employee of Small Bank Corporation.
The following solutions assume that all people work for at most one company.

i.

works	*person_name*	*company_name*	*salary*
	P._x	Small Bank Co	_y MAX.All. _y

or

works	*person_name*	*company_name*	*salary*
¬	P._x	Small Bank Co	_y > _y

ii.

$query\,(X)$:- $works\,(X,\, Y,\, Z),\; \neg\, p\,(X)$
$p\,(X)$:- $works(X, C, Y1),\; works\,(V,$ "Small Bank Corporation", $Y),\; Y > Y1$

h. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

Note: Small Bank Corporation will be included in each answer.

i.

located_in	*company_name*	*salary*
	Small Bank Corporation	*_x*
	P. *_c*	*_y*
	Small Bank Corporation	*_y*

condition
CNT.ALL.*_y*= CNT.ALL.*_x*=

ii.

query(X) :- company(X, C), not p(X)
p(X) :- company(X, C1), company("Small Bank Corporation", C2), not company(X, C2)

5.13 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give expressions in QBE, and Datalog equivalent to each of the following queries:

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

Answer:

a. $r_1 \cup r_2$

i.

result	*A*	*B*	*C*
P.	*_a*	*_b*	*_c*
P.	*_d*	*_e*	*_f*

r1	*A*	*B*	*C*
	_a	*_b*	*_c*

r2	*A*	*B*	*C*
	_d	*_e*	*_f*

ii.

query(X, Y, Z) :- r_1 (X, Y, Z)
query(X, Y, Z) :- r_2 (X, Y, Z)

b. $r_1 \cap r_2$

i.

r1	A	B	C
	_a	_b	_c

r2	A	B	C
	_d	_e	_f

ii. *query(X, Y, Z)* :- r_1 *(X, Y, Z),* r_2 *(X, Y, Z)*

c. $r_1 - r_2$

i.

r1	A	B	C
	_a	_b	_c

r2	A	B	C
	_d	_e	_f

ii. *query(X, Y, Z)* :- r_1 *(X, Y, Z), not* r_2 *(X, Y, Z)*

d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

i.

result		B	C
P.	_a	_b	_c

r1	A	B	C
	_a	_b	

r2	A	B	C
		_b	_c

ii. *query(X, Y, Z)* :- r_1 *(X, Y, V),* r_2 *(W, Y, Z)*

5.14 Write an extended relational-algebra view equivalent to the Datalog rule

$$p(A, C, D) \text{ :– } q1(A, B),\ q2(B, C),\ q3(4, B),\ D = B + 1.$$

Answer: Let us assume that $q1$, $q2$ and $q3$ are instances of the schema $(A1, A2)$. The relational algebra view is

create view P **as**

$\Pi_{q1.A1,q2.A2,q1.A2+1}(\sigma_{q3.A1=4 \wedge q1.A2=q2.A1 \wedge q1.A2=q3.A2}(q1 \times q2 \times q3))$

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

Figure 5.14. Employee database.

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*driver_id*, *car*, *report_number*, *damage_amount*)

Figure 5.15. Insurance database.

CHAPTER 6

Database Design and the E-R Model

Exercises

6.14 Explain the distinctions among the terms primary key, candidate key, and superkey.

Answer: A *superkey* is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. A superkey may contain extraneous attributes. If *K* is a superkey, then so is any superset of *K*. A superkey for which no proper subset is also a superkey is called a *candidate key*. It is possible that several distinct sets of attributes could serve as candidate keys. The *primary key* is one of the candidate keys that is chosen by the database designer as the principal means of identifying entities within an entity set.

6.15 Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

Answer: See Figure 6.1

6.16 Construct appropriate tables for each of the E-R diagrams in Practice Exercises 6.1 and 6.2.

Answer:

a. Car insurance tables:

person (driver-id, name, address)
car (license, year, model)
accident (report-number, date, location)
participated(driver-id, license, report-number, damage-amount)

b. Hospital tables:

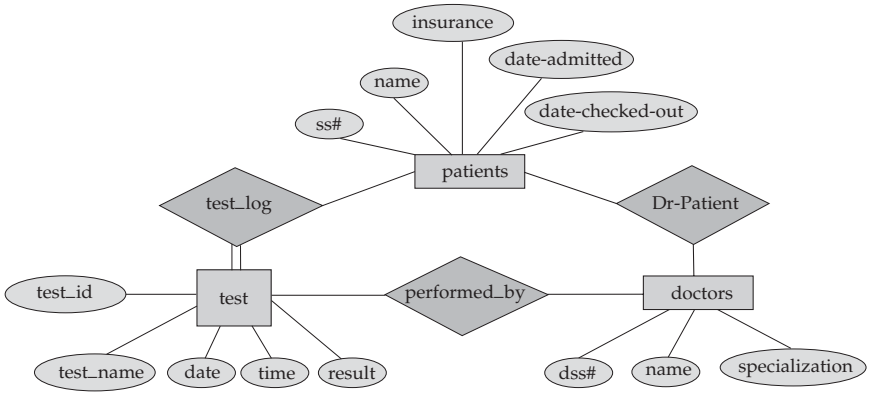


Figure 6.1 E-R diagram for a hospital.

patients (patient-id, name, insurance, date-admitted, date-checked-out)
doctors (doctor-id, name, specialization)
test (testid, testname, date, time, result)
doctor-patient (patient-id, doctor-id)
test-log (testid, patient-id) performed-by (testid, doctor-id)

c. University registrar's tables:

student (student-id, name, program)
course (courseno, title, syllabus, credits)
course-offering (courseno, secno, year, semester, time, room)
instructor (instructor-id, name, dept, title)
enrols (student-id, courseno, secno, semester, year, grade)
teaches (courseno, secno, semester, year, instructor-id)
requires (maincourse, prerequisite)

6.17 Extend the E-R diagram of Practice Exercise 6.4 to track the same information for all teams in a league.
Answer: See Figure 6.2 Note that a player can stay in only one team during a season.

6.18 Explain the difference between a weak and a strong entity set.
Answer: A strong entity set has a primary key. All tuples in the set are distinguishable by that key. A weak entity set has no primary key unless attributes of the strong entity set on which it depends are included. Tuples in a weak entity set are partitioned according to their relationship with tuples in a strong entity set. Tuples within each partition are distinguishable by a discriminator, which is a set of attributes.

6.19 We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?
Answer: We have weak entities for several reasons:

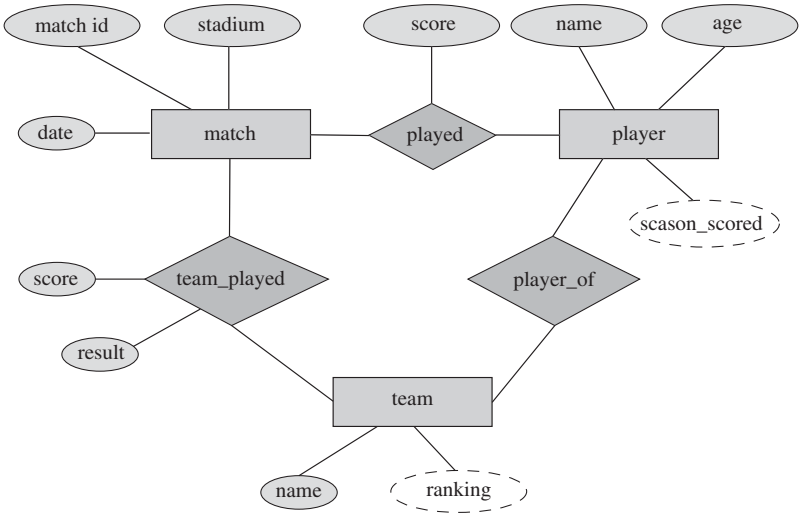


Figure 6.2 E-R diagram for all teams statistics.

- We want to avoid the data duplication and consequent possible inconsistencies caused by duplicating the key of the strong entity.
- Weak entities reflect the logical structure of an entity being dependent on another entity.
- Weak entities can be deleted automatically when their strong entity is deleted.
- Weak entities can be stored physically with their strong entities.

6.20 Define the concept of aggregation. Give two examples of where this concept is useful.

Answer: Aggregation is an abstraction through which relationships are treated as higher-level entities. Thus the relationship between entities A and B is treated as if it were an entity C. Some examples of this are:

a. Employees work for projects. An employee working for a particular project uses various machinery. See Figure 6.3

b. Manufacturers have tie-ups with distributors to distribute products. Each tie-up has specified for it the set of products which are to be distributed. See Figure 6.4

6.21 Consider the E-R diagram in Figure 6.31, which models an online bookstore.

a. List the entity sets and their primary keys.

b. Suppose the bookstore adds music cassettes and compact disks to its collection. The same music item may be present in cassette or compact disk format, with differing prices. Extend the E-R diagram to model this addition, ignoring the effect on shopping baskets.

c. Now extend the E-R diagram, using generalization, to model the case where a shopping basket may contain any combination of books, music cassettes, or compact disks.

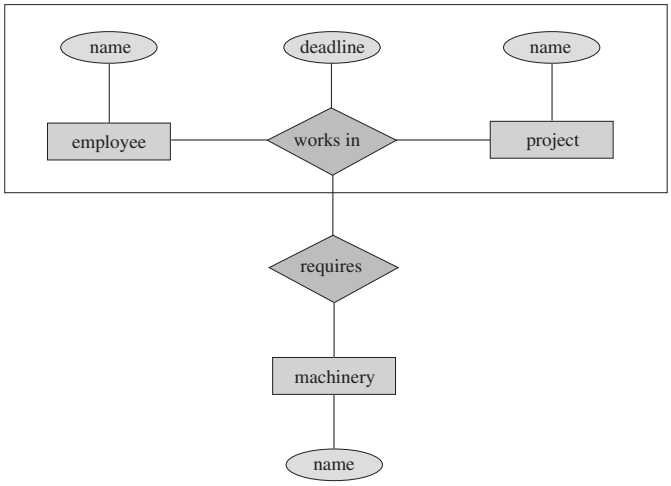


Figure 6.3 E-R diagram: Example 1 of aggregation.

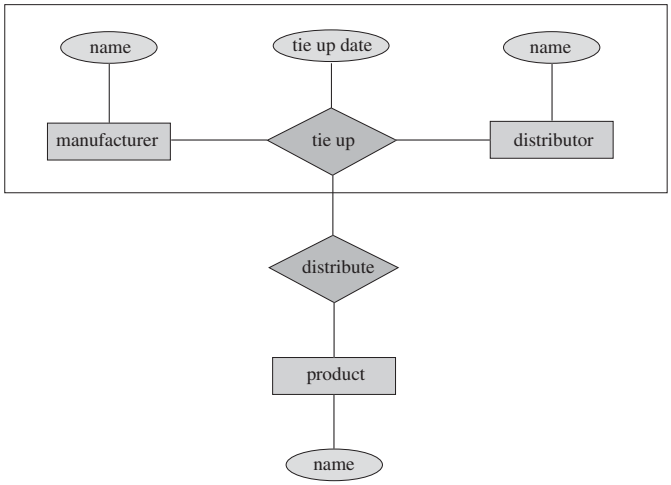


Figure 6.4 E-R diagram: Example 2 of aggregation.

Answer:

6.22 In Section 6.5.3, we represented a ternary relationship (repeated in Figure 6.29a) using binary relationships, as shown in Figure 6.29b. Consider the alternative shown in Figure 6.29c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.
Answer: The model of Figure 6.29c will not be able to represent all ternary relationships. Consider the *ABC* relationship set below.

A	B	C
1	2	3
4	2	7
4	8	3

If *ABC* is broken into three relationships sets *AB*, *BC* and *AC*, the three will imply that the relation (4, 2, 3) is a part of *ABC*.

6.23 Consider the relation schemas shown in Section 6.9.7, which were generated from the E-R diagram in Figure 6.25. For each schema, specify what foreign-key constraints, if any, should be created.
Answer: No Answer.

6.24 Design a generalization–specialization hierarchy for a motor-vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.
Answer: Figure 6.5 gives one possible hierarchy, there could be many different solutions. The generalization–specialization hierarchy for the motor-vehicle company is given in the figure. *model*, *sales-tax-rate* and *sales-volume* are attributes necessary for all types of vehicles. Commercial vehicles attract commercial vehicle tax, and each kind of commercial vehicle has a passenger carrying capacity specified for it. Some kinds of non-commercial vehicles attract luxury vehicle tax. Cars alone can be of several types, such as sports-car, sedan, wagon etc., hence the attribute *type*.

6.25 Explain the distinction between condition-defined and user-defined constraints. Which of these constraints can the system check automatically? Explain your answer.
Answer: In a generalization–specialization hierarchy, it must be possible to decide which entities are members of which lower level entity sets. In a condition-defined design constraint, membership in the lower level entity-sets is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate.User-defined lower-level entity sets are not constrained by a membership condition; rather, entities are assigned to a given entity set by the database user.

Condition-defined constraints alone can be automatically handled by the system. Whenever any tuple is inserted into the database, its membership in the various lower level entity-sets can be automatically decided by evaluating the respective membership predicates. Similarly when a tuple is updated, its membership in the various entity sets can be re-evaluated automatically.

6.26 Explain the distinction between disjoint and overlapping constraints.
Answer: In a disjointness design constraint, an entity can belong to not more than one lower-level entity set. In overlapping generalizations, the same entity may belong to more than one lower-level entity sets. For example, in the employee-workteam example of the book, a manager may participate in more than one work-team.

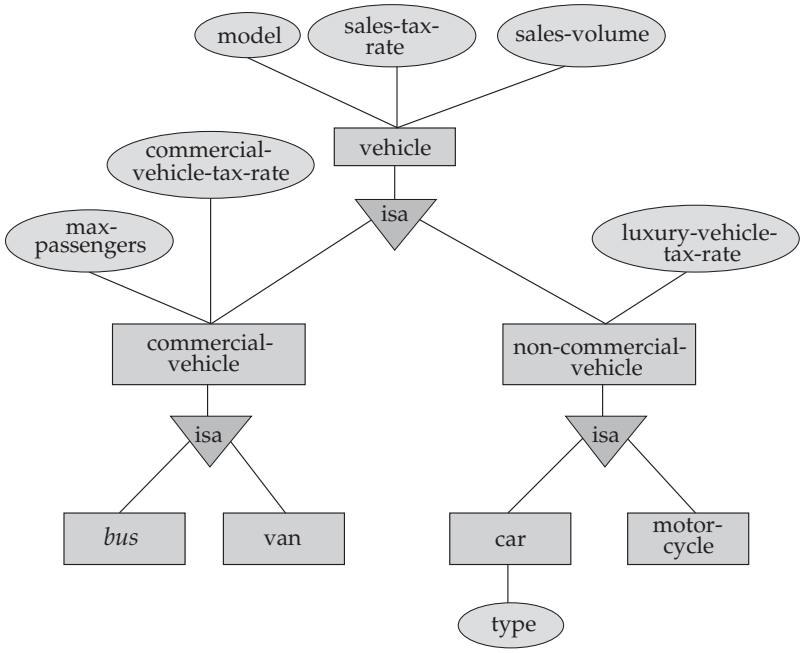


Figure 6.5 E-R diagram of motor-vehicle sales company.

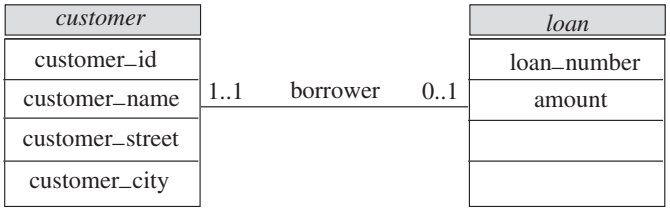


Figure 6.6 UML equivalent of Figure 6.8c.

6.27 Explain the distinction between total and partial constraints.
Answer: In a total design constraint, each higher-level entity must belong to a lower-level entity set. The same need not be true in a partial design constraint. For instance, some employees may belong to no work-team.

6.28 Draw the UML equivalents of the E-R diagrams of text Figures 6.8c, 6.9, 6.11, 6.12 and 6.20.
Answer: See Figures 6.6 to 6.10

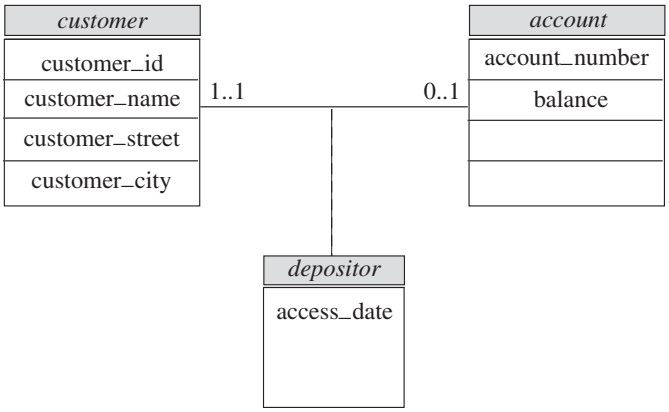


Figure 6.7 UML equivalent of Figure 6.9

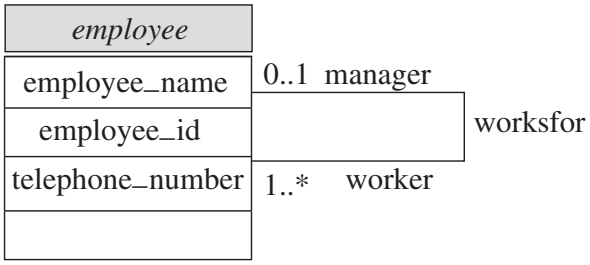


Figure 6.8 UML equivalent of Figure 6.11

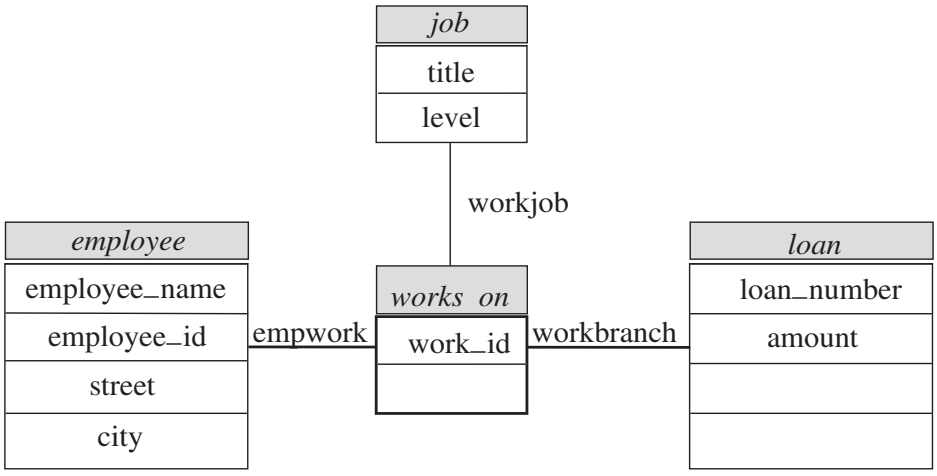


Figure 6.9 UML equivalent of Figure 6.12

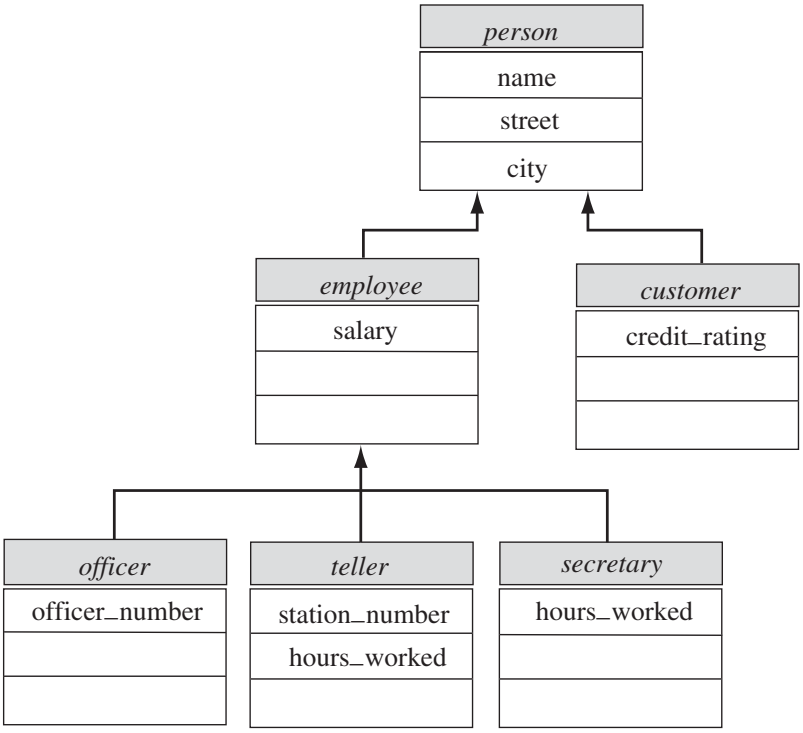


Figure 6.10 UML equivalent of Figure 6.20

CHAPTER 7

Relational Database Design

Exercises

7.17 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

Answer:

- Repetition of information is a condition in a relational database where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult.
- Inability to represent information is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values otherwise a tuple without the unrelated information cannot be inserted into the relation.
- Loss of information is a condition of a relational database which results from the decomposition of one relation into two relations and which cannot be combined to recreate the original relation. It is a bad relational database design because certain queries cannot be answered using the reconstructed relation that could have been answered using the original relation.

7.18 Why are certain functional dependencies called *trivial* functional dependencies?

Answer: Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations.

7.19 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

Answer: The definition of functional dependency is: $\alpha \rightarrow \beta$ holds on R if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Reflexivity rule: if α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.
Assume $\exists\, t_1$ and t_2 such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	since $\beta \subseteq \alpha$
$\alpha \rightarrow \beta$	definition of FD

Augmentation rule: if $\alpha \rightarrow \beta$, and γ is a set of attributes, then $\gamma\,\alpha \rightarrow \gamma\,\beta$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\gamma\,\alpha] = t_2[\gamma\,\alpha]$

$t_1[\gamma] = t_2[\gamma]$	$\gamma \subseteq \gamma\,\alpha$
$t_1[\alpha] = t_2[\alpha]$	$\alpha \subseteq \gamma\,\alpha$
$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma\,\beta] = t_2[\gamma\,\beta]$	$\gamma\,\beta = \gamma \cup \beta$
$\gamma\,\alpha \rightarrow \gamma\,\beta$	definition of FD

Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma] = t_2[\gamma]$	definition of $\beta \rightarrow \gamma$
$\alpha \rightarrow \gamma$	definition of FD

7.20 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.
Answer: Consider the following rule: if $A \rightarrow B$ and $C \rightarrow B$, then $A \rightarrow C$. That is, $\alpha = A$, $\beta = B$, $\gamma = C$. The following relation r is a counterexample to the rule.

r:

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2

Note: $A \rightarrow B$ and $C \rightarrow B$, (since no 2 tuples have the same C value, $C \rightarrow B$ is true trivially). However, it is not the case that $A \rightarrow C$ since the same A value is in two tuples, but the C value in those tuples disagree.

7.21 Use Armstrong's axioms to prove the soundness of the decomposition rule.
Answer: The decomposition rule, and its derivation from Armstrong's axioms are given below:
if $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

$\alpha \rightarrow \beta\gamma$	given
$\beta\gamma \rightarrow \beta$	reflexivity rule
$\alpha \rightarrow \beta$	transitivity rule
$\beta\gamma \rightarrow \gamma$	reflexive rule
$\alpha \rightarrow \gamma$	transitive rule

7.22 Using the functional dependencies of Practice Exercise 7.6, compute B^+.
Answer: Computing B^+ by the algorithm in Figure 7.9 we start with $result = \{B\}$. Considering FDs of the form $\beta \rightarrow \gamma$ in F, we find that the only dependencies satisfying $\beta \subseteq result$ are $B \rightarrow B$ and $B \rightarrow D$. Therefore $result = \{B, D\}$. No more dependencies in F apply now. Therefore $B^+ = \{B, D\}$

7.23 Show that the following decomposition of the schema *R* of Practice Exercise 7.1 is not a lossless-join decomposition:

$$(A, B, C)$$
$$(C, D, E).$$

Hint: Give an example of a relation *r* on schema *R* such that

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

Answer: Following the hint, use the following example of r:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

With $R_1 = (A, B, C)$, $R_2 = (C, D, E)$:

a. $\Pi_{R_1}(r)$ would be:

A	B	C
a_1	b_1	c_1
a_2	b_2	c_1

b. $\Pi_{R_2}(r)$ would be:

C	D	E
c_1	d_1	e_1
c_1	d_2	e_2

c. $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$ would be:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_2	b_2	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

Clearly, $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \neq r$. Therefore, this is a lossy join.

7.24 List the three design goals for relational databases, and explain why each is desirable.
Answer: The three design goals are lossless-join decompositions, dependency preserving decompositions, and minimization of repetition of information. They

```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
      let β → γ denote the ith FD;
      fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
      appears [A] := NIL;
      for i := 1 to |F| do
         begin
            let β → γ denote the ith FD;
            if A ∈ β then add i to appears [A];
         end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
      if A ∉ result then
         begin
            result := result ∪ {A};
            for each element i of appears[A] do
               begin
                  fdcount [i] := fdcount [i] − 1;
                  if fdcount [i] := 0 then
                     begin
                        let β → γ denote the ith FD;
                        addin (γ);
                     end
               end
         end
   end
```

Figure 7.19. An algorithm to compute α^+.

are desirable so we can maintain an accurate database, check correctness of updates quickly, and use the smallest amount of space possible.

7.25 Give a lossless-join decomposition into BCNF of schema R of Exercise 7.1.
Answer: From Exercise 7.6, we know that $B \rightarrow D$ is nontrivial and the left hand side is not a superkey. By the algorithm of Figure 7.12 we derive the relations $\{(A, B, C, E), (B, D)\}$. This is in BCNF.

7.26 In designing a relational database, why might we choose a non-BCNF design?
Answer: BCNF is not always dependency preserving. Therefore, we may want to choose another normal form (specifically, 3NF) in order to make checking dependencies easier during updates. This would avoid joins to check dependencies and increase system performance.

7.27 Give a lossless-join, dependency-preserving decomposition into 3NF of schema R of Practice Exercise 7.1.
Answer: First we note that the dependencies given in Practice Exercise 7.1 form a canonical cover. Generating the schema from the algorithm of Figure 7.13 we get

$$R' = \{(A, B, C), (C, D, E), (B, D), (E, A)\}.$$

Schema (A, B, C) contains a candidate key. Therefore R' is a third normal form dependency-preserving lossless-join decomposition.

Note that the original schema $R = (A, B, C, D, E)$ is already in 3NF. Thus, it was not necessary to apply the algorithm as we have done above. The single original schema is trivially a lossless join, dependency-preserving decomposition.

7.28 Given the three goals of relational-database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Practice Exercise 7.15 for the definition of 2NF.)
Answer: The three design goals of relational databases are to avoid

- Repetition of information
- Inability to represent information
- Loss of information.

2NF does not prohibit as much repetition of information since the schema (A, B, C) with dependencies $A \rightarrow B$ and $B \rightarrow C$ is allowed under 2NF, although the same (B, C) pair could be associated with many A values, needlessly duplicating C values. To avoid this we must go to 3NF. Repetition of information is allowed in 3NF in some but not all of the cases where it is allowed in 2NF. Thus, in general, 3NF reduces repetition of information. Since we can always achieve a lossless join 3NF decomposition, there is no loss of information needed in going from 2NF to 3NF.

Note that the decomposition $\{(A, B), (B, C)\}$ is a dependency-preserving and lossless-loin 3NF decomposition of the schema (A, B, C). However, in case we choose this decomposition, retrieving information about the relationship be-

tween A, B and C requires a join of two relations, which is avoided in the corresponding 2NF decomposition.

Thus, the decision of which normal form to choose depends upon how the cost of dependency checking compares with the cost of the joins. Usually, the 3NF would be preferred. Dependency checks need to be made with *every* insert or update to the instances of a 2NF schema, whereas, only some queries will require the join of instances of a 3NF schema.

7.29 Given a relational schema $r(A, B, C, D)$, does $AMVDBC$ logically imply $AMVDB$ and $AMVDC$? If yes prove it, else give a counter example.
Answer: No answer

7.30 Explain why 4NF is a normal form more desirable than BCNF.
Answer: 4NF is more desirable than BCNF because it reduces the repetition of information. If we consider a BCNF schema not in 4NF (see Practice Exercise 7.16), we observe that decomposition into 4NF does not lose information provided that a lossless join decomposition is used, yet redundancy is reduced.

CHAPTER 8

Application Design and Development

Exercises

8.8 Write a servlet and associated HTML code for the following very simple application: A user is allowed to submit a form containing a value, say n, and should get a response containing n "*" symbols.
Answer: No answer.

8.9 Write a servlet and associated HTML code for the following simple application: A user is allowed to submit a form containing a number, say n, and should get a response saying how many times the value n has been submitted previously. The number of times each value has been submitted previously should be stored in a database.
Answer: No answer.

8.10 Write a servlet that authenticates a user (based on user names and passwords stored in a database relation), and sets a session variable called *userid* after authentication.
Answer: No answer.

8.11 What is an SQL injection attack? Explain how it works, and what precautions must be taken to prevent SQL injection attacks.
Answer: No answer.

8.12 Write pseudocode to manage a connection pool. Your pseudocode must include a function to create a pool (providing a database connection string, database user name and password as parameters), a function to request a connection from the pool, a connection to release a connection to the pool, and a function to close the connection pool.
Answer: No answer.

8.13 Suppose there are two relations r and s, such that the foreign key B of r references the primary key A of s. Describe how the trigger mechanism can be used to implement the **on delete cascade** option, when a tuple is deleted from s.
Answer: We define triggers for each relation whose primary-key is referred to by the foreign-key of some other relation. The trigger would be activated whenever a tuple is deleted from the referred-to relation. The action performed by the trigger would be to visit all the referring relations, and delete all the tuples in them whose foreign-key attribute value is the same as the primary-key attribute value of the deleted tuple in the referred-to relation. These set of triggers will take care of the **on delete cascade** operation.

8.14 The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.
Answer: No answer.

8.15 Explain why, when a manager, say Mary, grants an authorization, the grant should be done by the manager role, rather than by the user Mary.
Answer: No answer.

8.16 Suppose user A, who has all authorizations on a relation r, grants select on relation r to **public** with grant option. Suppose user B then grants select on r to A. Does this cause a cycle in the authorization graph? Explain why.
Answer: No answer.

8.17 Make a list of security concerns for a bank. For each item on your list, state whether this concern relates to physical security, human security, operating-system security, or database security.
Answer: No answer.

8.18 Database systems that store each relation in a separate operating-system file may use the operating system's security and authorization scheme, instead of defining a special scheme themselves. Discuss an advantage and a disadvantage of such an approach.
Answer: No answer.

8.19 Oracle's VPD mechanism implements row-level security by adding predicates to the where clause of each query. Give an example of a predicate that could be used to implement row-level security, and three queries with the following properties:

- **a.** For the first query, the query with the added predicate gives the same result as the same as the original query.
- **b.** For the second query, the query with the added predicates gives a result that is always a subset of the original query result.
- **c.** For the third query, the query with the added predicate give incorrect answers.

Answer: No answer.

8.20 What are two advantages of encrypting data stored in the database?
Answer: No answer.

8.21 Suppose you wish to create an audit trail of changes to the *account* relation.

a. Define triggers to create an audit trail, logging the information into a relation called, for example, *account_trail*. The logged information should include the user-id (assume a function *user_id*() provides this information) and a time stamp, in addition to old and new values. You must also provide the schema of the *account_trail* relation.

b. Can the above implementation guarantee that updates made by a malicious database administrator (or someone who manages to get the administrator's password) will be in the audit trail? Explain your answer.

Answer: No answer.

8.22 Hackers may be able to fool you into believing that their Web site is actually a Web site (such as a bank or credit card Web site) that you trust. This may be done by misleading email, or even by breaking into the network infrastructure and re-routing network traffic destined for, say mybank.com, to the hackers site. If you enter your user name and password on the hackers site, the site can record it, and use it later to break into your account at the real site. When you use a URL such as https://mybank.com, the the HTTPS protocol is used to prevent such attacks. Explain how the protocol might use digital certificates to verify authenticity of the site.
Answer: No answer.

8.23 Explain what is a challenge-response system for authentication. Why is it more secure than a traditional password-based system?
Answer: No answer.

CHAPTER 9

Object-Based Databases

Exercises

9.7 Redesign the database of Exercise 9.2 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first- and fourth-normal-form schemas.

Answer: To put the schema into first normal form, we flatten all the attributes into a single relation schema.

Employee-details = (*ename, cname, bday, bmonth, byear, stype, xyear, xcity*)

We rename the attributes for the sake of clarity. *cname* is *Children.name*, and *bday, bmonth, byear* are the *Birthday* attributes. *stype* is *Skills.type*, and *xyear* and *xcity* are the *Exams* attributes. The FDs and multivalued dependencies we assume are:-

$$\begin{aligned} ename, cname &\rightarrow bday, bmonth, byear \\ ename &\twoheadrightarrow cname, bday, bmonth, byear \\ ename, stype &\twoheadrightarrow xyear, xcity \end{aligned}$$

The FD captures the fact that a child has a unique birthday, under the assumption that one employee cannot have two children of the same name. The MVDs capture the fact there is no relationship between the children of an employee and his or her skills-information.

The redesigned schema in fourth normal form is:-

Employee = (*ename*)
Child = (*ename, cname, bday, bmonth, byear*)
Skill = (*ename, stype, xyear, xcity*)

ename will be the primary key of *Employee*, and (*ename, cname*) will be the primary key of *Child*. The *ename* attribute is a foreign key in *Child* and in *Skill*, referring to the *Employee* relation.

9.8 Consider the schema from Practice Exercise 9.2.

a. Give SQL:2003 DDL statements to create a relation *EmpA* which has the same information as *Emp*, but where multiset valued attributes *ChildrenSet*, *SkillsSet* and *ExamsSet* are replaced by array valued attributes *ChildrenArray*, *SkillsArray* and *ExamsArray*.

b. Write a query to convert data from the schema of *Emp* to that of *EmpA*, with the array of children sorted by birthday, the array of skills by the skill type and the array of exams by the year.

c. Write an SQL statement to update the *Emp* relation by adding a child Jeb, with a birthdate of February 5, 2001, to the employee named George.

d. Write an SQL statement to perform the same update as above but on the *EmpA* relation. Make sure that the array of children remains sorted by year.

Answer: No answer

9.9 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 9.4. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.
Answer: A corresponding relational schema in third normal form is given below:-

People = (*name, address*)
Students = (*name, degree, student-department*)
Teachers = (*name, salary, teacher-department*)

name is the primary key for all the three relations, and it is also a foreign key referring to *People*, for both *Students* and *Teachers*.

Instead of placing only the *name* attribute of *People* in *Students* and *Teachers*, both its attributes can be included. In that case, there will be a slight change, namely – (*name, address*) will become the foreign key in *Students* and *Teachers*. The primary keys will remain the same in all tables.

9.10 Explain the distinction between a type x and a reference type **ref**(x). Under what circumstances would you choose to use a reference type?
Answer: If the type of an attribute is x, then in each tuple of the table, corresponding to that attribute, there is an actual object of type x . If its type is **ref**(x), then in each tuple, corresponding to that attribute, there is a *reference* to some object of type x. We choose a reference type for an attribute, if that attribute's intended purpose is to refer to an independent object.

9.11 Give an SQL:1999 schema definition of the E-R diagram in Figure 9.7, which contains specializations, using subtypes and subtables.
Answer:

create type *Person*
(*name* **varchar(30)**,
street **varchar(15)**,
city **varchar(15)**)
create type *Employee*
under *Person*
(*salary* **integer**)
create type *Customer*
under *Person*
(*credit-rating* **integer**)
create type *Officer*
under *Employee*
(*office-number* **integer**)
create type *Teller*
under *Employee*
(*station-number* **integer**,
hours-worked **integer**)
create type *Secretary*
under *Employee*
(*hours-worked* **integer**)
create table *person* **of** *Person*
create table *employee* **of** *Employee*
under *person*
create table *customer* **of** *Customer*
n **under** *person*
create table *officer* **of** *Officer*
under *employee*
create table *teller* **of** *Teller*
under *employee*
create table *secretary* **of** *Secretary*
under *employee*

9.12 Suppose a JDO database had an object A, which references object B, which in turn references object C. Assume all objects are on disk initially. Suppose a program first dereferences A, then dereferences B by following the reference from A, and then finally dereferences C. Show the objects that are represented in memory after each dereference, along with their state (hollow or filled, and values in their reference fields).
Answer: No answer.

CHAPTER 10

XML

Exercises

10.10 Show, by giving a DTD, how to represent the Non-1NF *books* relation from Section 9.2, using XML.

Answer:

```
<!DOCTYPE bib [
    <!ELEMENT book (title, author+, publisher, keyword+)>
    <!ELEMENT publisher (pub-name, pub-branch) >
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT author ( #PCDATA )>
    <!ELEMENT keyword ( #PCDATA )>
    <!ELEMENT pub-name( #PCDATA )>
    <!ELEMENT pub-branch( #PCDATA )>
] >
```

10.11 Write the following queries in XQuery, assuming the DTD from Practice Exercise 10.2.

a. Find the names of all employees who have a child who has a birthday in March.

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

c. List all skill types in *Emp*.

Answer:

a. Find the names of all employees who have a child who has a birthday in March.

```
for $e in /db/emp,
    $m in distinct($e/children/birthday/month)
where $m = 'March'
return $e/ename
```

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

```
for $e in /db/emp
    $s in $e/skills[type='typing']
    $exam in $s/exams
where $exam/city= 'Dayton'
return $e/ename
```

c. List all skill types in *Emp*.

```
for $t in distinct (/db/emp/skills/type)
return $t
```

10.12 Consider the XML data shown in Figure 10.2. Suppose we wish to find purchase orders that ordered two or more copies of the part with identifier 123. Consider the following attempt to solve this problem:

```
for $p in purchaseorder
where $p/part/id = 123 and $p/part/quantity >= 2
return $p
```

Explain why the query may return some purchase orders that order less than two copies of part 123. Give a correct version of the above query.
Answer: No answer

10.13 Give a query in XQuery to flip the nesting of data from Exercise 10.10. That is, at the outermost level of nesting the output must have elements corresponding to authors, and each such element must have nested within it items corresponding to all the books written by the author.
Answer: No answer

10.14 Give the DTD for an XML representation of the information in Figure 6.31. Create a separate element type to represent each relationship, but use ID and IDREF to implement primary and foreign keys.
Answer: The answer is given in Figure 10.1.

10.15 Give an XMLSchema representation of the DTD from Exercise 10.14.
Answer: No answer

10.16 Write queries in XQuery on the bibliography DTD fragment in Figure 10.15, to do the following.

a. Find all authors who have authored a book and an article in the same year.
b. Display books and articles sorted by year.
c. Display books with more than one author.
d. Find all books that contain the word "database" in their title and the word "Hank" in an author's name (whether first or last).

```
<!DOCTYPE bookstore [
    <!ELEMENT basket (contains+, basket-of)>
    <!ATTLIST basket
        basketid ID #REQUIRED >
    <!ELEMENT customer (name, address, phone)>
    <!ATTLIST customer
        email ID #REQUIRED >
    <!ELEMENT book (year, title, price, written-by, published-by)>
    <!ATTLIST book
        ISBN ID #REQUIRED >
    <!ELEMENT warehouse (address, phone, stocks)>
    <!ATTLIST warehouse
        code ID #REQUIRED >
    <!ELEMENT author (name, address, URL)>
    <!ATTLIST author
        authid ID #REQUIRED >
    <!ELEMENT publisher (address, phone)>
    <!ATTLIST publisher
        name ID #REQUIRED >
    <!ELEMENT basket-of >
    <!ATTLIST basket-of
        owner IDREF #REQUIRED >
    <!ELEMENT contains >
    <!ATTLIST contains
        book IDREF #REQUIRED
        number CDATA #REQUIRED >
    <!ELEMENT stocks >
    <!ATTLIST stocks
        book IDREF #REQUIRED
        number CDATA #REQUIRED >
    <!ELEMENT written-by >
    <!ATTLIST written-by
        authors IDREFS #REQUIRED >
    <!ELEMENT published-by >
    <!ATTLIST published-by
        publisher IDREF #REQUIRED >
    <!ELEMENT name (#PCDATA )>
    <!ELEMENT address (#PCDATA )>
    <!ELEMENT phone (#PCDATA )>
    <!ELEMENT year (#PCDATA )>
    <!ELEMENT title (#PCDATA )>
    <!ELEMENT price (#PCDATA )>
    <!ELEMENT number (#PCDATA )>
    <!ELEMENT URL (#PCDATA )>
] >
```

Figure 10.1 XML DTD for Bookstore

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last-name, first-name) >
    <!ELEMENT title ( #PCDATA )>
    · · · similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last-name and first-name
] >
```

Figure 10.15. DTD for bibliographical data.

Answer:

a. Find all authors who have authored a book and an article in the same year.

```
for $a in distinct (/bib/book/author),
    $y in /bib/book[author=$a]/year,
    $art in /bib/article[author=$a and year=$y]
return $a
```

b. Display books and articles sorted by year.

```
for $a in ((/bib/book) | (/bib/article))
return $a sortby(year)
```

c. Display books with more than one author.

```
for $b in ((/bib/book[author/count()>1])
return $b
```

d. Find all books that contain the word "database" in their title and the word "Hank" in an author's name (whether first or last).
No Answer.

10.17 Give a relational mapping of the XML purchase order schema illustrated in Figure 10.2, using the approach described in Section 10.6.2.3.

Suggest how to remove redundancy in the relational schema, if item identifiers functionally determine the description and purchase and supplier names functionally determine the purchase and supplier address, respectively.
Answer: No answer

10.18 Write queries in SQL/XML to convert bank data from the relational schema we have used in earlier chapters to the *bank-1* and *bank-2* XML schemas. (For the *bank-2* schema, you may assume that the customer relation has an extra attribute *customer_id*.)
Answer: No answer

10.19 As in Exercise 10.18, write queries to convert bank data to the *bank-1* and *bank-2* XML schemas, but this time by writing XQuery queries on the default SQL/XML

database to XML mapping.
Answer: No answer

10.20 One way to shred an XML document is to use XQuery to convert the schema to an SQL/XML mapping of the corresponding relational schema, and then use the SQL/XML mapping in the backward direction to populate the relation.

As an illustration, give an XQuery query to convert data from the *bank-1* XML schema to the SQL/XML schema shown in Figure 10.14.
Answer: No answer

10.21 Consider the example XML schema from Section 10.3.2, and write XQuery queries to carry out the following tasks.

a. Check if the key constraint shown in Section 10.3.2 holds.
b. Check if the keyref constraint shown in Section 10.3.2 holds.

Answer: No answer

10.22 Consider Practice Exercise 10.7, and suppose that authors could also appear as top-level elements. What change would have to be done to the relational schema?
Answer: No answer

CHAPTER 11

Storage and File Structure

Exercises

11.8 List the physical storage media available on the computers you use routinely. Give the speed with which data can be accessed on each medium.
Answer: Your answer will be based on the computers and storage media that you use. Typical examples would be hard disk, floppy disks and CD-ROM drives.

11.9 How does the remapping of bad sectors by disk controllers affect data-retrieval rates?
Answer: Remapping of bad sectors by disk controllers does reduce data retrieval rates because of the loss of sequentiality amongst the sectors. But that is better than the loss of data in case of no remapping!

11.10 RAID systems typically allow you to replace failed disks without stopping access to the system. Thus, the data in the failed disk must be rebuilt and written to the replacement disk while the system is in operation. With which of the RAID levels is the amount of interference between the rebuild and ongoing disk accesses least? Explain your answer.
Answer: RAID level 1 (mirroring) is the one which facilitates rebuilding of a failed disk with minimum interference with the on-going disk accesses. This is because rebuilding in this case involves copying data from just the failed disk's mirror. In the other RAID levels, rebuilding involves reading the entire contents of all the other disks.

11.11 Explain why the allocation of records to blocks affects database-system performance significantly.
Answer: If we allocate related records to blocks, we can often retrieve most, or all, of the requested records by a query with one disk access. Disk accesses

tend to be the bottlenecks in databases; since this allocation strategy reduces the number of disk accesses for a given operation, it significantly improves performance.

11.12 If possible, determine the buffer-management strategy used by the operating system running on your local computer system, and what mechanisms it provides to control replacement of pages. Discuss how the control on replacement that it provides would be useful for the implementation of database systems.
Answer: The typical OS uses LRU for buffer replacement. This is often a bad strategy for databases. As explained in Section 11.5.2 of the text, MRU is the best strategy for nested loop join. In general no single strategy handles all scenarios well, and ideally the database system should be given its own buffer cache for which the replacement policy takes into account all the performance related issues.

11.13 In the sequential file organization, why is an overflow *block* used even if there is, at the moment, only one overflow record?
Answer: An overflow block is used in sequential file organization because a block is the smallest space which can be read from a disk. Therefore, using any smaller region would not be useful from a performance standpoint. The space saved by allocating disk storage in record units would be overshadowed by the performance cost of allowing blocks to contain records of multiple files.

11.14 List two advantages and two disadvantages of each of the following strategies for storing a relational database:

a. Store each relation in one file.
b. Store multiple relations (perhaps even the entire database) in one file.

Answer:

a. Advantages of storing a relation as a file include using the file system provided by the OS , thus simplifying the DBMS, but incurs the disadvantage of restricting the ability of the DBMS to increase performance by using more sophisticated storage structures.
b. By using one file for the entire database, these complex structures can be implemented through the DBMS, but this increases the size and complexity of the DBMS.

11.15 Give a normalized version of the *Index-metadata* relation, and explain why using the normalized version would result in worse performance.
Answer: The *Index-metadata* relation can be normalized as follows

Index-metadata (index-name, relation-name, index-type, attrib-set)
Attribset-metadata (relation-name, attrib-set, attribute-name)

Though the normalized version will have less space requirements, it will require extra disk accesses to read *Attribset-metadata* everytime an index has to be accessed. Thus, it will lead to worse performance.

11.16 If you have data that should not be lost on disk failure, and the data are write intensive, how would you store the data?
Answer: No answer.

11.17 In earlier generation disks the number of sectors per track was the same across all tracks. Current generation disks have more sectors per track on outer tracks, and fewer sectors per track on inner tracks (since they are shorter in length). What is the effect of such a change on each of the three main indicators of disk speed?
Answer: No answer.

11.18 Standard buffer managers assume each page is of the same size and costs the same to read. Consider a buffer manager that, instead of LRU, uses the rate of reference to objects, that is, how often an object has been accessed in the last n seconds. Suppose we want to store in the buffer objects of varying sizes, and varying read costs (such as Web pages, whose read cost depends on the site from which they are fetched). Suggest how a buffer manager may choose which page to evict from the buffer.
Answer: No answer.

CHAPTER 12

Indexing and Hashing

Exercises

12.13 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

Answer: It is preferable to use a dense index instead of a sparse index when the file is not sorted on the indexed field (such as when the index is a secondary index) or when the index file is small compared to the size of memory.

12.14 What is the difference between a clustering index and a secondary index?

Answer: The clustering index is on the field which specifies the sequential order of the file. There can be only one clustering index while there can be many secondary indices.

12.15 For each B^+-tree of Practice Exercise 12.3, show the steps involved in the following queries:

a. Find records with a search-key value of 11.

b. Find records with a search-key value between 7 and 17, inclusive.

Answer:

With the structure provided by the solution to Practice Exercise 12.3a:

a. Find records with a value of 11
   i. Search the first level index; follow the first pointer.
   ii. Search next level; follow the third pointer.
   iii. Search leaf node; follow first pointer to records with key value 11.

b. Find records with value between 7 and 17 (inclusive)
   i. Search top index; follow first pointer.
   ii. Search next level; follow second pointer.

iii. Search third level; follow second pointer to records with key value 7, and after accessing them, return to leaf node.
iv. Follow fourth pointer to next leaf block in the chain.
v. Follow first pointer to records with key value 11, then return.
vi. Follow second pointer to records with with key value 17.

With the structure provided by the solution to Practice Exercise 12.3b:

a. Find records with a value of 11
i. Search top level; follow second pointer.
ii. Search next level; follow second pointer to records with key value 11.
b. Find records with value between 7 and 17 (inclusive)
i. Search top level; follow second pointer.
ii. Search next level; follow first pointer to records with key value 7, then return.
iii. Follow second pointer to records with key value 11, then return.
iv. Follow third pointer to records with key value 17.

With the structure provided by the solution to Practice Exercise 12.3c:

a. Find records with a value of 11
i. Search top level; follow second pointer.
ii. Search next level; follow first pointer to records with key value 11.
b. Find records with value between 7 and 17 (inclusive)
i. Search top level; follow first pointer.
ii. Search next level; follow fourth pointer to records with key value 7, then return.
iii. Follow eighth pointer to next leaf block in chain.
iv. Follow first pointer to records with key value 11, then return.
v. Follow second pointer to records with key value 17.

12.16 he solution presented in Section 12.5.3 to deal with non-unique search keys added an extra attribute to the search key. What effect does this change have on the height of the B^+-tree?
Answer: No answer.

12.17 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.
Answer: Open hashing may place keys with the same hash function value in different buckets. Closed hashing always places such keys together in the same bucket. Thus in this case, different buckets can be of different sizes, though the implementation may be by linking together fixed size buckets using overflow chains. Deletion is difficult with open hashing as *all* the buckets may have to inspected before we can ascertain that a key value has been deleted, whereas in closed hashing only that bucket whose address is obtained by hashing the key value need be inspected. Deletions are more common in databases and hence closed hashing is more appropriate for them. For a small, static set of data lookups may be more efficient using open hashing. The symbol table of a compiler would be a good example.

12.18 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?
Answer: The causes of bucket overflow are :-

- **a.** Our estimate of the number of records that the relation will have was too low, and hence the number of buckets allotted was not sufficient.
- **b.** Skew in the distribution of records to buckets. This may happen either because there are many records with the same search key value, or because the the hash function chosen did not have the desirable properties of uniformity and randomness.

To reduce the occurrence of overflows, we can :-

- **a.** Choose the hash function more carefully, and make better estimates of the relation size.
- **b.** If the estimated size of the relation is n_r and number of records per block is f_r, allocate $(n_r/f_r) * (1 + d)$ buckets instead of (n_r/f_r) buckets. Here d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that some of the skew is handled and the probability of overflow is reduced.

12.19 Why is a hash structure not the best choice for a search key on which range queries are likely?
Answer: A range query cannot be answered efficiently using a hash index, we will have to read all the buckets. This is because key values in the range do not occupy consecutive locations in the buckets, they are distributed uniformly and randomly throughout all the buckets.

12.20 Suppose there is a relation $R(A, B, C)$, with a B$^+$-tree index with search key (A, B).

- **a.** What is the worst case cost of finding records satisfying $10 < A < 50$ using this index, in terms of the number of records retrieved n_1 and the height h of the tree?
- **b.** What is the worst case cost of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$ using this index, in terms of the number of records n_2 that satisfy this selection, as well as n_1 and h defined above.
- **c.** Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$.

Answer: No answer.

12.21 Suppose you have to create a B$^+$-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 charac ters) and the average name is itself large (say 10 characters). Explain how prefix compression can be used to maximize the average fanout of internal nodes.
Answer: No answer.

12.22 Why might the leaf nodes of a B^+-tree file organization lose sequentiality? Suggest how the file organization may be reorganized to restore sequentiality.
Answer: No answer.

12.23 Suppose a relation is stored in a B^+-tree file organization. Suppose secondary indices stored record identifiers that are pointers to records on disk.

a. What would be the effect on the secondary indices if a page split happens in the file organization?
b. What would be the cost of updating all affected records in a secondary index?
c. How does using the search key of the file organization as a logical record identifier solve this problem?
d. What is the extra cost due to the use of such logical record identifiers?

Answer: No answer.

12.24 Show how to compute existence bitmaps from other bitmaps. Make sure that your technique works even in the presence of null values, by using a bitmap for the value *null*.
Answer: The existence bitmap for a relation can be calculated by taking the union (logical-or) of all the bitmaps on that attribute, including the bitmap for value *null*.

12.25 How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?
Answer: Note that indices must operate on the encrypted data or someone could gain access to the index to interpret the data. Otherwise, the index would have to be restricted so that only certain users could access it. To keep the data in sorted order, the index scheme would have to decrypt the data at each level in a tree. Note that hash systems would not be affected.

12.26 Our description of static hashing assumes that a large contiguous stretch of disk blocks can be allocated to a static hash table. Suppose you can allocate only C contiguous blocks. Suggest how to implement the hash table, if it can be much larger than C blocks. Access to a block should still be efficient.
Answer: No answer.

CHAPTER 13

Query Processing

Exercises

13.10 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

Answer: In general it is not desirable to force users to choose a query processing strategy because naive users might select an inefficient strategy. The reason users would make poor choices about processing queries is that they would not know how a relation is stored, nor about its indices. It is unreasonable to force users to be aware of these details since ease of use is a major object of database query languages. If users are aware of the costs of different strategies they could write queries efficiently, thus helping performance. This could happen if experts were using the system.

13.11 Design a variant of the hybrid merge–join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

Answer: We merge the leaf entries of the first sorted secondary index with the leaf entries of the second sorted secondary index. The result file contains pairs of addresses, the first address in each pair pointing to a tuple in the first relation, and the second address pointing to a tuple in the second relation.

This result file is first sorted on the first relation's addresses. The relation is then scanned in physical storage order, and addresses in the result file are replaced by the actual tuple values. Then the result file is sorted on the second relation's addresses, allowing a scan of the second relation in physical storage order to complete the join.

13.12 Estimate the number of block accesses required by your solution to Practice Exercise 13.11 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Exercise 13.3.

Answer: r_1 occupies 800 blocks, and r_2 occupies 1500 blocks. Let there be n pointers per index leaf block (we assume that both the indices have leaf blocks and pointers of equal sizes). Let us assume M pages of memory, $M < 800$. r_1's index will need $B_1 = \lceil \frac{20000}{n} \rceil$ leaf blocks, and r_2's index will need $B_2 = \lceil \frac{45000}{n} \rceil$ leaf blocks. Therefore the merge join will need $B_3 = B_1 + B_2$ accesses, without output. The number of output tuples is estimated as $n_o = \lceil \frac{20000 * 45000}{max(V(C,r_1),V(C,r_2))} \rceil$. Each output tuple will need two pointers, so the number of blocks of join output will be $B_{o1} = \lceil \frac{n_o}{n/2} \rceil$. Hence the join needs $B_j = B_3 + B_{o1}$ disk block accesses.

Now we have to replace the pointers by actual tuples. For the first sorting, $B_{s1} = B_{o1}(2\lceil log_{M-1}(B_{o1}/M) \rceil + 2)$ disk accesses are needed, including the writing of output to disk. The number of blocks of r_1 which have to be accessed in order to replace the pointers with tuple values is $min(800, n_o)$. Let n_1 pairs of the form (r_1 *tuple, pointer to* r_2) fit in one disk block. Therefore the intermediate result after replacing the r_1 pointers will occupy $B_{o2} = \lceil (n_o/n_1) \rceil$ blocks. Hence the first pass of replacing the r_1-pointers will cost $B_f = B_{s1} + B_{o1} + min(800, n_o) + B_{o2}$ disk accesses.

The second pass for replacing the r_2-pointers has a similar analysis. Let n_2 tuples of the final join fit in one block. Then the second pass of replacing the r_2-pointers will cost $B_s = B_{s2} + B_{o2} + min(1500, n_o)$ disk accesses, where $B_{s2} = B_{o2}(2\lceil log_{M-1}(B_{o2}/M) \rceil + 2)$.

Hence the total number of disk accesses for the join is $B_j + B_f + B_s$, and the number of pages of output is $\lceil n_o/n_2 \rceil$.

13.13 The hash join algorithm as described in Section 13.5.5 computes the natural join of two relations. Describe how to extend the hash join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *customer* and *depositor* relations.

Answer: For the probe relation tuple t_r under consideration, if no matching tuple is found in the build relation's hash partition, it is padded with nulls and included in the result. This will give us the natural left outer join $t_r \ \bowtie\!\!\!\!\supset \ t_s$. To get the natural right outer join $t_r \ \subset\!\!\!\!\bowtie \ t_s$, we can keep a boolean flag with each tuple in the current build relation partition H_{s_i} residing in memory, and set it whenever any probe relation tuple matches with it. When we are finished with H_{s_i}, all the tuples in it which do not have their flag set, are padded with nulls and included in the result. To get the natural full outer join, we do both the above operations together.

To try out our algorithm, we use the sample *customer* and *depositor* relations of Figures 13.17 and 13.18. Let us assume that there is enough memory to hold three tuples of the build relation plus a hash index for those three tuples. We use *depositor* as the build relation. We use the simple hashing function which returns the first letter of *customer_name*. Taking the first partitions, we get

customer_name	*customer_street*	*customer_city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Curry	North	Rye
Smith	North	Rye
Turner	Putnam	Stamford
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Williams	Nassau	Princeton

Figure 13.17 Sample *customer* relation

customer_name	*account_number*
Johnson	A-101
Johnson	A-201
Jones	A-217
Smith	A-215
Hayes	A-102
Turner	A-305
David	A-306
Lindsay	A-222

Figure 13.18 Sample *depositor* relation

$H_{r_1} = \{$("Adams", "Spring", "Pittsfield")$\}$, and $H_{s_1} = \phi$. The tuple in the probe relation partition will have no matching tuple, so ("Adams", "Spring", "Pittsfield", $null$) is outputted. In the partition for "D", the lone build relation tuple is unmatched, thus giving an output tuple ("David", $null$, $null$, A-306). In the partition for "H", we find a match for the first time, producing the output tuple ("Hayes", "Main", "Harrison", A-102). Proceeding in a similar way, we process all the partitions and complete the join.

13.14 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Use the standard iterator functions in your pseudocode. Show what state information the iterator must maintain between calls.

Answer: Let *outer* be the iterator which returns successive tuples from the pipelined outer relation. Let *inner* be the iterator which returns successive tuples of the inner relation having a given value at the join attributes. The *inner* iterator returns these tuples by performing an index lookup. The functions **In-**

dexedNLJoin::open, **IndexedNLJoin::close** and **IndexedNLJoin::next** to implement the indexed nested-loop join iterator are given below. The two iterators *outer* and *inner*, the value of the last read outer relation tuple t_r and a flag $done_r$ indicating whether the end of the outer relation scan has been reached are the state information which need to be remembered by **IndexedNLJoin** between calls.

```
IndexedNLJoin::open()
begin
    outer.open();
    inner.open();
    done_r := false;
    if(outer.next() ≠ false)
        move tuple from outer's output buffer to t_r;
    else
        done_r := true;
end
```

```
IndexedNLJoin::close()
begin
    outer.close();
    inner.close();
end
```

```
boolean IndexedNLJoin::next()
begin
    while(¬done_r)
    begin
        if(inner.next(t_r[JoinAttrs]) ≠ false)
        begin
            move tuple from inner's output buffer to t_s;
            compute t_r ⋈ t_s and place it in output buffer;
            return true;
        end
        else
            if(outer.next() ≠ false)
            begin
                move tuple from outer's output buffer to t_r;
                rewind inner to first tuple of s;
            end
            else
                done_r := true;
    end
    return false;
end
```

13.15 Pipelining is used to avoid writing intermediate results to disk. Suppose you need to sort relation r using sort–merge and merge-join the result with an already sorted relation s.

a. Describe how the output of the sort of r can be pipelined to the merge join without being written back to disk.

b. The same idea is applicable even if both inputs to the merge-join are the outputs of sort–merge oeprations. However, the available memory has to be shared between the two merge operations (the merge-join algorithm itself needs very little memory). What is the effect of having to share memory on the cost of each sort–merge operation.

Answer: No answer.

13.16 Suppose you have to compute ${}_{A}\mathcal{G}_{sum(C)}(r)$ as well as ${}_{A,B}\mathcal{G}_{sum(C)}(r)$. Describe how to compute these together using a single sorting of r.

Answer: No answer.

CHAPTER 14

Query Optimization

Exercises

14.11 Suppose that a B^+-tree index on (*branch-name, branch-city*) is available on relation *branch*. What would be the best way to handle the following selection?

$$\sigma_{(branch\text{-}city < \text{“Brooklyn”}) \wedge (assets < 5000) \wedge (branch\text{-}name = \text{“Downtown”})}(branch)$$

Answer: Using the index, we locate the first tuple having *branch-name* "Downtown". We then follow the pointers retrieving successive tuples as long as *branch-city* is less than "Brooklyn". From the tuples retrieved, the ones not satisfying the condition $(assets < 5000)$ are rejected.

14.12 Show how to derive the following equivalences by a sequence of transformations using the equivalence rules in Section 14.2.1.

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

Answer:

a. Using rule 1, $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E)$ becomes $\sigma_{\theta_1}(\sigma_{\theta_2 \wedge \theta_3}(E))$. On applying rule 1 again, we get $\sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$.

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2)$ on applying rule 1 becomes $\sigma_{\theta_1}(\sigma_{\theta_2}(E_1 \bowtie_{\theta_3} E_2))$. This on applying rule 7.a becomes $\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$.

14.13 A set of equivalence rules is said to be *complete* if, whenever two expressions are equivalent, one can be derived from the other by a sequence of uses of the equivalence rules. Is the set of equivalence rules that we considered in Section 14.2.1 complete? Hint: Consider the equivalence $\sigma_{3=5}(r) = \{\}$.

Answer: Two relational expressions are defined to be *equivalent* when on all input relations, they give the same output. The set of equivalence rules considered in Section 14.2.1 is not complete. The expressions $\sigma_{3=5}(r)$ and $\{\ \}$ are equivalent, but this cannot be shown by using just these rules.

14.14 Explain how to use a histogram to estimate the size of a selection of the form $\sigma_{A \leq v}(r)$.
Answer: No answer.

14.15 Suppose two relations r and s have histograms on attributes $r.A$ and $s.A$, respectively, but with different ranges. Suggest how to use the histograms to estimate the size of $r \bowtie s$. Hint: Split the ranges of each histogram further.
Answer: No answer.

14.16 Describe how to incrementally maintain the results of the following operations, on both insertions and deletions.

a. Union and set difference
b. Left outer join

Answer:

a. Given materialized view $v = r \cup s$, when a tuple is inserted in r, we check if it is present in v, and if not we add it to v. When a tuple is deleted from r, we check if it is there in s, and if not, we delete it from v. Inserts and deletes in s are handled in symmetric fashion.

For set difference, given view $v = r - s$, when a tuple is inserted in r, we check if it is present in s, and if not we add it to v. When a tuple is deleted from r, we delete it from v if present. When a tuple is inserted in s, we delete it from v if present. When a tuple is deleted from s, we check if it is present in r, and if so we add it to v.

b. Given materialized view $v = r \ \mathbin{⟕} \ s$, when a set of tuples i_r is inserted in r, we add the tuples $i_r \ \mathbin{⟕} \ s$ to the view. When i_r is deleted from r, we delete $i_r \ \mathbin{⟕} \ s$ from the view. When a set of tuples i_s is inserted in s, we compute $r \bowtie i_s$. We find all the tuples of r which previously did not match any tuple from s(i.e. those padded with *null* in $r \ \mathbin{⟕} \ s$) but which match i_s. We remove all those *null* padded entries from the view and add the tuples $r \bowtie s$ to the view. When i_s is deleted from s, we delete the tuples $r \bowtie i_s$ from the view. Also we find all the tuples in r which match i_s but which do not match any other tuples in s. We add all those to the view, after padding them with *null* values.

14.17 Give an example of an expression defining a materialized view and two situations (sets of statistics for the input relations and the differentials) such that incremental view maintenance is better than recomputation in one situation, and recomputation is better in the other situation.
Answer: Let r and s be two relations. Consider a materialized view on these defined by $(r \bowtie s)$. Suppose 70% of the tuples in r are deleted. Then recompution is better that incremental view maintenance. This is because in incre-

mental view maintenance, the 70% of the deleted tuples have to be joined with s while in recompution, just the remaining 30% of the tuples in r have to be joined with s.

However, if the number of tuples in r is increased by a small percentage, for example 2%, then incremental view maintenance is likely to be better than recomputation.

14.18 Suppose you want to get answers to $r \bowtie s$ sorted on an attribute of r, and want only the top K answers for some relatively small K. Give a good way of evaluating the query

a. When the join is on a foreign key of r referencing s.
b. When the join is not on a foreign key.

Answer: No answer.

14.19 Consider a relation $r(A, B, C)$, with an index on attribute A. Give an example of a query that can be answered by using the index only, without looking at the tuples in the relation. (Query plans that use only the index, without accessing the actual relation, are called *index-only* plans.)
Answer: No answer.

CHAPTER 15

Transactions

Exercises

15.8 List the ACID properties. Explain the usefulness of each.
Answer: The ACID properties, and the need for each of them are:

- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are. Clearly lack of atomicity will lead to inconsistency in the database.
- **Isolation:** When multiple transactions execute concurrently, it should be the case that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently with it. The user view of a transaction system requires the isolation property, and the property that concurrent schedules take the system from one consistent state to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency preserving transactions are allowed.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

15.9 During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.
Answer: The possible sequences of states are:-

a. *active → partially committed → committed*. This is the normal sequence a successful transaction will follow. After executing all its statements it enters the *partially committed* state. After enough recovery information has been written to disk, the transaction finally enters the *committed* state.

b. *active → partially committed → aborted*. After executing the last statement of the transaction, it enters the *partially committed* state. But before enough recovery information is written to disk, a hardware failure may occur destroying the memory contents. In this case the changes which it made to the database are undone, and the transaction enters the *aborted* state.

c. *active → failed → aborted*. After the transaction starts, if it is discovered at some point that normal execution cannot continue (either due to internal program errors or external errors), it enters the failed state. It is then rolled back, after which it enters the *aborted* state.

15.10 Explain the distinction between the terms *serial schedule* and *serializable schedule*.

Answer: A schedule in which all the instructions belonging to one single transaction appear together is called a *serial schedule*. A *serializable schedule* has a weaker restriction that it should be *equivalent* to some serial schedule. There are two definitions of schedule equivalence – conflict equivalence and view equivalence. Both of these are described in the chapter.

15.11 Consider the following two transactions:

T_1: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_2: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

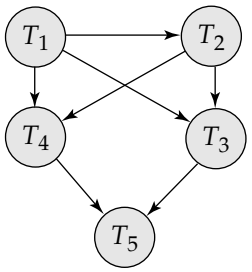


Figure 15.18. Precedence graph.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.
b. Show a concurrent execution of T_1 and T_2 that produces a nonserializable schedule.
c. Is there a concurrent execution of T_1 and T_2 that produces a serializable schedule?

Answer:

a. There are two possible executions: T_1 T_2 and T_2 T_1.

Case 1:

	A	B
initially	0	0
after T_1	0	1
after T_2	0	1

Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$

Case 2:

	A	B
initially	0	0
after T_2	1	0
after T_1	1	0

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$

b. Any interleaving of T_1 and T_2 results in a non-serializable schedule.

T_1	T_2
read(A)	
	read(B)
	read(A)
read(B)	
if $A = 0$ **then** $B = B + 1$	
	if $B = 0$ **then** $A = A + 1$
	write(A)
write(B)	

c. There is no parallel execution resulting in a serializable schedule. From part a. we know that a serializable schedule results in $A = 0 \vee B = 0$. Suppose we start with T_1 **read**(A). Then when the schedule ends, no matter when we run the steps of T_2, $B = 1$. Now suppose we start executing T_2 prior to completion of T_1. Then T_2 **read**(B) will give B a value of 0. So when T_2 completes, $A = 1$. Thus $B = 1 \wedge A = 1 \rightarrow \neg (A = 0 \vee B = 0)$. Similarly for starting with T_2 **read**(B).

15.12 What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.
Answer: A recoverable schedule is one where, for each pair of transactions

T_i and T_j such that T_j reads data items previously written by T_i, the commit operation of T_i appears before the commit operation of T_j. Recoverable schedules are desirable because failure of a transaction might otherwise bring the system into an irreversibly inconsistent state. Nonrecoverable schedules may sometimes be needed when updates must be made visible early due to time constraints, even if they have not yet been committed, which may be required for very long duration transactions.

15.13 Why do database systems support concurrent execution of transactions, in spite of the extra programming effort needed to ensure that concurrent execution does not cause any problems?
Answer: No answer.

CHAPTER 16

Concurrency Control

Exercises

16.19 What benefit does strict two-phase locking provide? What disadvantages result?

Answer: Because it produces only cascadeless schedules, recovery is very easy. But the set of schedules obtainable is a subset of those obtainable from plain two phase locking, thus concurrency is reduced.

16.20 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

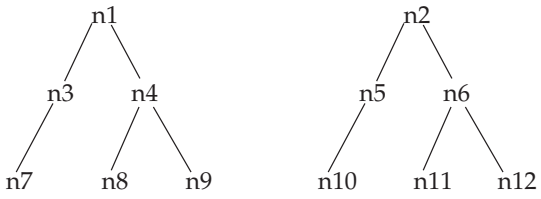
Answer: It is relatively simple to implement, imposes low rollback overhead because of cascadeless schedules, and usually allows an acceptable level of concurrency.

16.21 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

Answer: Take a system with 2 trees:



We have 2 transactions, T_1 and T_2. Consider the following legal schedule:

T_1	T_2
lock(n1)	
lock(n3)	
write(n3)	
unlock(n3)	
	lock(n2)
	lock(n5)
	write(n5)
	unlock(n5)
lock(n5)	
read(n5)	
unlock(n5)	
unlock(n1)	
	lock(n3)
	read(n3)
	unlock(n3)
	unlock(n2)

This schedule is not serializable.

16.22 When a transaction is rolled back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?
Answer: A transaction is rolled back because a newer transaction has read or written the data which it was supposed to write. If the rolled back transaction is re-introduced with the same timestamp, the same reason for rollback is still valid, and the transaction will have be rolled back again. This will continue indefinitely.

16.23 In multiple-granularity locking, what is the difference between implicit and explicit locking?
Answer: When a transaction *explicitly* locks a node in shared or exclusive mode, it *implicitly* locks all the descendents of that node in the same mode. The transaction need not explicitly lock the descendent nodes. There is no difference in the functionalities of these locks, the only difference is in the way they are acquired, and their presence tested.

16.24 Although SIX mode is useful in multiple-granularity locking, an exclusive and intend-shared (XIS) mode is of no use. Why is it useless?
Answer: An exclusive lock is incompatible with any other lock kind. Once a node is locked in exclusive mode, none of the descendents can be simultaneously accessed by any other transaction in any mode. Therefore an exclusive and intend-shared declaration has no meaning.

16.25 Show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.
Answer: A schedule which is allowed in the two-phase locking protocol but not in the timestamp protocol is:

step	T_0	T_1	*Precedence marks*
1	**lock-S**(A)		
2	**read**(A)		
3		**lock-X**(B)	
4		**write**(B)	
5		**unlock**(B)	
6	**lock-S**(B)		
7	**read**(B)		$T_1 \rightarrow T_0$
8	**unlock**(A)		
9	**unlock**(B)		

This schedule is not allowed in the timestamp protocol because at step 7, the W-timestamp of B is 1.

A schedule which is allowed in the timestamp protocol but not in the two-phase locking protocol is:

step	T_0	T_1	T_2
1	**write**(A)		
2		**write**(A)	
3			**write**(A)
4	**write**(B)		
5		**write**(B)	

This schedule cannot have lock instructions added to make it legal under two-phase locking protocol because T_1 must unlock (A) between steps 2 and 3, and must lock (B) between steps 4 and 5.

16.26 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a read request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for write requests?

Answer: Using the commit bit, a read request is made to wait if the transaction which wrote the data item has not yet committed. Therefore, if the writing transaction fails before commit, we can abort that transaction alone. The waiting read will then access the earlier version in case of a multiversion system, or the restored value of the data item after abort in case of a single-version system. For writes, this commit bit checking is unnecessary. That is because either the write is a "blind" write and thus independent of the old value of the data item or there was a prior read, in which case the test was already applied.

16.27 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?
Answer: Deadlock avoidance is preferable if the consequences of abort are serious (as in interactive transactions), and if there is high contention and a resulting high probability of deadlock.

16.28 If deadlock is avoided by deadlock avoidance schemes, is starvation still possible? Explain your answer.
Answer: A transaction may become the victim of deadlock-prevention rollback arbitrarily many times, thus creating a potential starvation situation.

16.29 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?
Answer: The phantom phenomenon arises when, due to an insertion or deletion, two transactions logically conflict despite not locking any data items in common. The insertion case is described in the book. Deletion can also lead to this phenomenon. Suppose T_i deletes a tuple from a relation while T_j scans the relation. If T_i deletes the tuple and then T_j reads the relation, T_i should be serialized before T_j. Yet there is no tuple that both T_i and T_j conflict on.

An interpretation of 2PL as just locking the accessed tuples in a relation is incorrect. There is also an index or a relation data that has information about the tuples in the relation. This information is read by any transaction that scans the relation, and modified by transactions that update, or insert into, or delete from the relation. Hence locking must also be performed on the index or relation data, and this will avoid the phantom phenomenon.

16.30 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?
Answer: The degree-two consistency avoids cascading aborts and offers increased concurrency but the disadvantage is that it does not guarantee serializability and the programmer needs to ensure it.

16.31 Give example schedules to show that with key-value locking, if any of lookup, insert, or delete do not lock the next-key value, the phantom phenomemon could go undetected.
Answer: No answer.

16.32 If many transactions update a common item (e.g., the cash balance at a br anch), and private items (e.g., individual account balances). Explain how you can in-

crease concurrency (and throughput) by ordering the operations of the transaction.

Answer: No answer.

16.33 Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher numbered items may be locked. Locks may be released at any time. Only X-locks are used.

Show by an example that this protocol does not guarantee serializability.

Answer: No answer.

CHAPTER 17

Recovery System

Exercises

17.7 Explain the difference between the three storage types—volatile, nonvolatile, and stable—in terms of I/O cost.

Answer: Volatile storage is storage which fails when there is a power failure. Cache, main memory, and registers are examples of volatile storage. Nonvolatile storage is storage which retains its content despite power failures. An example is magnetic disk. Stable storage is storage which theoretically survives any kind of failure (short of a complete disaster!). This type of storage can only be approximated by replicating data.

In terms of I/O cost, volatile memory is the fastest and non-volatile storage is typically several times slower. Stable storage is slower than non-volatile storage because of the cost of data replication.

17.8 Stable storage cannot be implemented.

a. Explain why it cannot be.

b. Explain how database systems deal with this problem.

Answer:

a. Stable storage cannot really be implemented because all storage devices are made of hardware, and all hardware is vulnerable to mechanical or electronic device failures.

b. Database systems approximate stable storage by writing data to multiple storage devices simultaneously. Even if one of the devices crashes, the data will still be available on a different device. Thus data loss becomes extremely unlikely.

17.9 Assume that immediate modification is used in a system. Show, by an example, how an inconsistent database state could result if log records for a transaction

are not output to stable storage prior to data updated by the transaction being written to disk.

Answer: Consider a banking scheme and a transaction which transfers \$50 from account A to account B. The transaction has the following steps:

a. **read**(A,a_1)
b. $a_1 := a_1 - 50$
c. **write**(A,a_1)
d. **read**(B,b_1)
e. $b_1 := b_1 + 50$
f. **write**(B,b_1)

Suppose the system crashes after the transaction commits, but before its log records are flushed to stable storage. Further assume that at the time of the crash the update of A in the third step alone had actually been propagated to disk whereas the buffer page containing B was not yet written to disk. When the system comes up it is in an inconsistent state, but recovery is not possible because there are no log records corresponding to this trabsaction in stable storage.

17.10 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect

- System performance when no failure occurs
- The time it takes to recover from a system crash
- The time it takes to recover from a disk crash

Answer: Checkpointing is done with log-based recovery schemes to reduce the time required for recovery after a crash. If there is no checkpointing, then the entire log must be searched after a crash, and all transactions undone/redone from the log. If checkpointing had been performed, then most of the log-records prior to the checkpoint can be ignored at the time of recovery.

Another reason to perform checkpoints is to clear log-records from stable storage as it gets full.

Since checkpoints cause some loss in performance while they are being taken, their frequency should be reduced if fast recovery is not critical. If we need fast recovery checkpointing frequency should be increased. If the amount of stable storage available is less, frequent checkpointing is unavoidable. Checkpoints have no effect on recovery from a disk crash; archival dumps are the equivalent of checkpoints for recovery from disk crashes.

17.11 Explain how the buffer manager may cause the database to become inconsistent if some log records pertaining to a block are not output to stable storage before the block is output to disk.

Answer: If a data item x is modified on disk by a transaction before the corresponding log record is written to stable storage, then the only record of the old value of x is in main memory where it would be lost in a crash. If the transaction had not yet finished at the time of the crash, an unrecoverable inconsistency will result.

17.12 Explain the benefits of logical logging. Give examples of one situation where logical logging is preferable to physical logging and one situation where physical logging is preferable to logical logging.
Answer: Logical logging has less log space requirement, and with logical undo logging it allows early release of locks. This is desirable in situations like concurrency control for index structures, where a very high degree of concurrency is required. An advantage of employing physical redo logging is that fuzzy checkpoints are possible. Thus in a system which needs to perform frequent checkpoints, this reduces checkpointing overhead.

17.13 Explain the difference between a system crash and a "disaster."
Answer: In a system crash, the CPU goes down, and disk may also crash. But stable-storage at the site is assumed to survive system crashes. In a "disaster", *everything* at a site is destroyed. Stable storage needs to be distributed to survive disasters.

17.14 For each of the following requirements, identify the best choice of degree of durability in a remote backup system:

a. Data loss must be avoided but some loss of availability may be tolerated.
b. Transaction commit must be accomplished quickly, even at the cost of loss of some committed transactions in a disaster.
c. A high degree of availability and durability is required, but a longer running time for the transaction commit protocol is acceptable.

Answer:

a. Two very safe is suitable here because it guarantees durability of updates by committed transactions, though it can proceed only if both primary and backup sites are up. Availability is low, but it is mentioned that this is acceptable.
b. One safe committing is fast as it does not have to wait for the logs to reach the backup site. Since data loss can be tolerated, this is the best option.
c. With two safe committing, the probability of data loss is quite low, and also commits can proceed as long as at least the primary site is up. Thus availability is high. Commits take more time than in the one safe protocol, but that is mentioned as acceptable.

17.15 The Oracle database system uses undo log records to provide a snapshot view of the database to read-only transactions. The snapshot view reflects updates of all transactions that had committed when the read-only transaction started; updates of all other transactions are not visible to the read-only transactions.

Describe a scheme for buffer handling whereby read-only transactions are given a snapshot view of pages in the buffer. Include details of how to use the log to generate the snapshot view, assuming that the advanced recovery algorithm is used. Assume for simplicity that a logical operation and its undo affect only a single page.
Answer: No answer.

CHAPTER 18

Data Analysis and Mining

Exercises

18.8 For each of the SQL aggregate functions **sum, count, min** and **max**, show how to compute the aggregate value on a multiset $S_1 \cup S_2$, given the aggregate values on multisets S_1 and S_2.

Based on the above, give expressions to compute aggregate values with grouping on a subset S of the attributes of a relation $r(A, B, C, D, E)$, given aggregate values for grouping on attributes $T \supseteq S$, for the following aggregate functions:

a. **sum, count, min** and **max**
b. **avg**
c. standard deviation

Answer: Given aggregate values on multisets S_1 and S_2, e can calulate the corresponding aggregate values on multiset $S_1 \cup S_2$ as follows:

a. **sum**$(S_1 \cup S_2) =$ **sum**$(S_1) +$ **sum**(S_2)
b. **count**$(S_1 \cup S_2) =$ **count**$(S_1) +$ **count**(S_2)
c. **min**$(S_1 \cup S_2) =$ **min**(**min**(S_1), **min**(S_2))
d. **max**$(S_1 \cup S_2) =$ **max**(**max**(S_1), **max**(S_2))

Let the attribute set *T = (A, B, C, D)* and the attribute set *S = (A, B)*. Let the aggregation on the attribute set T be stored in table *aggregation-on-t* with aggregation columns *sum-t*, *count-t*, *min-t*, and *max-t* storing **sum, count, min** and **max** resp.

a. The aggregations *sum-s*, *count-s*, *min-s*, and *max-s* on the attribute set *S* are computed by the query:

select *A*, *B*, **sum**(*sum-t*) **as** *sum-s*, **sum**(*count-t*) **as** *count-s*,
min(*min-t*) **as** *min-s*, **max**(*max-t*) **as** *max-s*
from *aggregation-on-t*
groupby *A*, *B*

b. The aggregation *avg* on the attribute set *S* is computed by the query:

select *A*, *B*, **sum**(*sum-t*)/**sum**(*count-t*) **as** *avg-s*
from *aggregation-on-t*
groupby *A*, *B*

c. For calculating standard deviation we use an alternative formula:

$$stddev(S) = \frac{\sum_{s \in S} s^2}{|S|} - avg(S)^2$$

which we get by expanding the formula

$$stddev(S) = \frac{\sum_{s \in S} (s^2 - avg(S))^2}{|S|}$$

If S is partitioned into n sets $S_1, S_2, \ldots S_n$ then the following relation holds:

$$stddev(S) = \frac{\sum_{S_i} |S_i|(stddev(S_i)^2 + avg(S_i)^2)}{|S|} - avg(S)^2$$

Using this formula, the aggregation **stddev** is computed by the query:

select *A*, *B*,
[**sum**(*count-t* $*$ (*stddev-t*^{2}+ *avg-t*^{2}))/**sum**(*count-t*)] -
[**sum**(*sum-t*)/**sum**(*count-t*)]
from *aggregation-on-t*
groupby *A*, *B*

18.9 Give an example of a pair of groupings that cannot be expressed by using a single **group by** clause with **cube** and **rollup**.
Answer: Consider an example of hierarchies on dimensions from Figure 18.4. We can not express a query to seek aggregation on groups (*City*, *Hour of day*) and (*City*, *Date*) using a single **group by** clause with **cube** and **rollup**.

Any single **groupby** clause with **cube** and **rollup** that computes these two groups would also compute other groups also.

18.10 Given relation $r(a, b, c)$, Show how to use the extended SQL features to generate a histogram of c versus a, dividing a into 20 equal-sized partitions (that is, where each partition contains 5 percent of the tuples in r, sorted by a).
Answer:

select *tile20*, **sum**(*c*)
from (**select** *c*, **ntile**(20) **over** (**order by** (*a*)) **as** *tile20*
from *r*) **as** *s*
groupby *tile20*

18.11 Consider the *balance* attribute of the *account* relation. Write an SQL query to compute a histogram of *balance* values, dividing the range 0 to the maximum account balance present, into three equal ranges.
Answer:

```
(select 1, count(*)
 from account
 where 3* balance <= (select max(balance)
                      from account)
)
union
(select 2, count(*)
 from account
 where 3* balance > (select max(balance)
                     from account)
       and 1.5* balance <= (select max(balance)
                            from account)
)
union
(select 3, count(*)
 from account
 where 1.5* balance > (select max(balance)
                       from account)
)
```

18.12 Construct a decision tree classifier with binary splits at each node, using tuples in relation $r(A, B, C)$ shown below as training data; attribute C denotes the class. Show the final tree, and with each node show the best split for each attribute along with its information gain value.

$$(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b), (3, 6, b), (4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)$$

Answer: No answer.

18.13 Suppose half of all the transactions in a clothes shop purchase jeans, and one third of all transactions in the shop purchase T-shirts. Suppose also that half of the transactions that purchase jeans also purchase T-shirts. Write down all the (nontrivial) association rules you can deduce from the above information, giving support and confidence of each rule.

Answer: The rules are as follows. The last rule can be deduced from the previous ones.

Rule	Support	Conf.
$\forall\, transactions\ T,\ \mathbf{true} \Rightarrow buys(T, jeans)$	50%	50%
$\forall\, transactions\ T,\ \mathbf{true} \Rightarrow buys(T, t\text{-}shirts)$	33%	33%
$\forall\, transactions\ T,\ buys(T, jeans) \Rightarrow buys(T, t\text{-}shirts)$	25%	50%
$\forall\, transactions\ T,\ buys(T, t\text{-}shirts) \Rightarrow buys(T, jeans)$	25%	75%

18.14 Consider the problem of finding large itemsets.

a. Describe how to find the support for a given collection of itemsets by using a single scan of the data. Assume that the itemsets and associated information, such as counts, will fit in memory.

b. Suppose an itemset has support less than j. Show that no superset of this itemset can have support greater than or equal to j.

Answer:

a. Let $\{S_1, S_2, \ldots, S_n\}$ be the collection of item-sets for which we want to find the support. Associate a counter $count(S_i)$ with each item-set S_i.

Initialize each counter to zero. Now examine the transactions one-by-one. Let $S(T)$ be the item-set for a transaction T. For each item-set S_i that is a subset of $S(T)$, increment the corresponding counter $count(S_i)$.

When all the transactions have been scanned, the values of $count(S_i)$ for each i will give the support for item-set S_i.

b. Let A be an item-set. Consider any item-set B which is a superset of A. Let τ_A and τ_B be the sets of transactions that purchase all items in A and all items in B, respectively. For example, suppose A is $\{a, b, c\}$, and B is $\{a, b, c, d\}$.

A transaction that purchases all items from B must also have purchased all items from A (since $A \subseteq B$). Thus, every transaction in τ_B is also in τ_A. This implies that the number of transactions in τ_B is at most the number of transactions in τ_A. In other words, the support for B is at most the support for A.

Thus, if any item-set has support less than j, all supersets of this item-set have support less than j.

18.15 Create a small example of a set of transactions showing that that although many transactions contain two items, that is the itemset containing the two items has a high support, purchase of one of the items may have a negative correlation with purchase of the other.
Answer: No answer.

18.16 The organization of parts, chapters, sections and subsections in a book is related to clustering. Explain why, and to what form of clustering.
Answer: No answer.

18.17 Suggest how predictive mining techniques can be used by a sports team, using your favorite sport as an example.
Answer: No answer.

CHAPTER 19

Information Retrieval

Exercises

19.6 Using a simple definition of term frequency as the number of occurrences of the term in a document, give the TF-IDF scores of each term in the set of documents consisting of this and the next exercise.
Answer: No answer.

19.7 Create a small example of a 4 small documents each with a PageRank, and create inverted lists for the documents sorted by the PageRank. You do not need to compute PageRank, just assume some values for each page.
Answer: No answer.

19.8 Suppose you wish to perform keyword querying on a set of tuples in a database, where each tuple has only a few attributes, each containing only a few words. Does the concept of term frequency make sense in this context? And that of inverse document frequency? Explain your answer. Also suggest how you can define the similarity of two tuples using TF-IDF concepts.
Answer: No answer.

19.9 Web sites that want to get some publicity can join a Web ring, where they create links to other sites in the ring, in exchange for other sites in the ring creating links to their site. What is the effect of such rings on popularity ranking techniques such as PageRank?
Answer: No answer.

19.10 The Google search engine provide a feature whereby Web sites can display advertisements supplied by Google. The advertisements supplied are based on the contents of the page. Suggest how Google might choose which advertisements to supply for a page, given the page contents.
Answer: No answer.

19.11 One way to create a keyword-specific version of PageRank is to modify the random jump such that a jump is only possible to pages containing the keyword. Thus pages that do not contain the keyword but are close (in terms of links) to pages that contain the keyword also get a non-zero rank for that keyword.

a. Give equations defining such a keyword-specific version of PageRank.
b. Give a formula for computing the relevance of a page to a query containing multiple keywords.

Answer: No answer.

19.12 The idea of popularity ranking using hyperlinks can be extended to relational and XML data, using foreign key and IDREF edges in place of hyperlinks. Suggest how such a ranking scheme may be of value in the following applications.

a. A bibliographic database, which has links from articles to authors of the articles and links from each article to every article that it references.
b. A sales database which has links from each sales record to the items that were sold.

Also suggest why prestige ranking can give less than meaningful results in a movie database that records which actor has acted in which movies.
Answer: No answer.

19.13 What is the difference between a false positive and a false drop? If it is essential that no relevant information be missed by an information retrieval query, is it acceptable to have either false positives or false drops? Why?
Answer: No answer.

CHAPTER 20

Database-System Architectures

Exercises

20.6 Why is it relatively easy to port a database from a single processor machine to a multiprocessor machine if individual queries need not be parallelized?
Answer: Porting is relatively easy to a shared memory multiprocessor machine. Databases designed for single-processor machines already provide multitasking, allowing multiple processes to run on the same processor in a time-shared manner, giving a view to the user of multiple processes running in parallel. Thus, coarse-granularity parallel machines logically appear to be identical to single-processor machines, making the porting relatively easy.

Porting a database to a shared disk or shared nothing multiprocessor architecture is a little harder.

20.7 Transaction server architectures are popular for client-server relational databases, where transactions are short. On the other hand, data server architectures are popular for client-server object-oriented database systems, where transactions are expected to be relatively long. Give two reasons why data servers may be popular for object-oriented databases but not for relational databases.
Answer: Data servers are good if data transfer is small with respect to computation, which is often the case in applications of OODBs such as computer aided design. In contrast, in typical relational database applications such as transaction processing, a transaction performs little computation but may touch several pages, which will result in a lot of data transfer with little benefit in a data server architecture. Another reason is that structures such as indices are heavily used in relational databases, and will become spots of contention in a data server architecture, requiring frequent data transfer. There are no such points of frequent contention in typical current-day OODB applications such as computer aided design.

20.8 What is lock de-escalation, and under what conditions is it required? Why is it not required if the unit of data shipping is an item?

Answer: In a client-server system with page shipping, when a client requests an item, the server typically grants a lock not on the requested item, but on the *page* having the item, thus implicitly granting locks on all the items in the page. The other items in the page are said to be *prefetched*. If some other client subsequently requests one of the prefetched items, the server may ask the owner of the page lock to transfer back the lock on this item. If the page lock owner doesn't need this item, it de-escalates the page lock that it holds, to item locks on all the items that it is actually accessing, and then returns the locks on the unwanted items. The server can then grant the latter lock request.

If the unit of data shipping is an item, there are no coarser granularity locks; even if prefetching is used, it is typically implemented by granting individual locks on each of the prefetched items. Thus when the server asks for a return of a lock, there is no question of de-escalation, the requested lock is just returned if the client has no use for it.

20.9 Suppose you were in charge of the database operations of a company whose main job is to process transactions. Suppose the company is growing rapidly each year, and has outgrown its current computer system. When you are choosing a new parallel computer, what measure is most relevant—speedup, batch scaleup, or transaction scaleup? Why?

Answer: With increasing scale of operations, we expect that the number of transactions submitted per unit time increases. On the other hand, we wouldn't expect most of the individual transactions to grow longer, nor would we require that a given transaction should execute more quickly now than it did before. Hence transaction scale-up is the most relevant measure in this scenario.

20.10 Database systems are typically implemented as a set of processes (or threads) sharing a shared memory area.

a. How is access to the shared memory area controlled?

b. Is two-phase locking appropriate for serializing access to the data structures in shared memory? Explain your answer.

Answer: No answer.

20.11 What are the factors that can work against linear scaleup in a transaction processing system? Which of the factors are likely to be the most important in each of the following architectures: shared memory, shared disk, and shared nothing?

Answer: Increasing contention for shared resources prevents linear scale-up with increasing parallelism. In a shared memory system, contention for memory (which implies bus contention) will result in falling scale-up with increasing parallelism. In a shared disk system, it is contention for disk and bus access which affects scale-up. In a shared-nothing system, inter-process communication overheads will be the main impeding factor. Since there is no shared mem-

ory, acquiring locks, and other activities requiring message passing between processes will take more time with increased parallelism.

20.12 Processor speeds have been increasing much faster than memory access speeds. What impact does this have on the number of processors that can effectively share a common memory?
Answer: No answer.

20.13 Consider a bank that has a collection of sites, each running a database system. Suppose the only way the databases interact is by electronic transfer of money between one another. Would such a system qualify as a distributed database? Why?
Answer: In a distributed system, all the sites typically run the same database management software, and they share a global schema. Each site provides an environment for execution of both global transactions initiated at remote sites and local transactions. The system described in the question does not have these properties, and hence it cannot qualify as a distributed database.

CHAPTER 21

Parallel Databases

Exercises

21.7 For each of the three partitioning techniques, namely round-robin, hash partitioning, and range partitioning, give an example of a query for which that partitioning technique would provide the fastest response.

Answer:

Round robin partitioning:

When relations are large and queries read entire relations, round-robin gives good speed-up and fast response time.

Hash partitioning

For point queries, this gives the fastest response, as each disk can process a query simultaneously. If the hash partitioning is uniform, even entire relation scans can be performed efficiently.

Range partitioning

For range queries which access a few tuples, this gives fast response.

21.8 What factors could result in skew when a relation is partitioned on one of its attributes by:

a. Hash partitioning
b. Range partitioning

In each case, what can be done to reduce the skew?

Answer:

a. Hash-partitioning:

Too many records with the same value for the hashing attribute, or a poorly chosen hash function without the properties of randomness and uniformity, can result in a skewed partition. To improve the situation, we should experiment with better hashing functions for that relation.

b. Range-partitioning:

Non-uniform distribution of values for the partitioning attribute (including duplicate values for the partitioning attribute) which are not taken into account by a bad partitioning vector is the main reason for skewed partitions. Sorting the relation on the partitioning attribute and then dividing it into n ranges with equal number of tuples per range will give a good partitioning vector with very low skew.

21.9 Give an example of a join that is not a simple equi-join for which partitioned parallelism can be used. What attributes should be used for partitioning?
Answer: We give two examples of such joins.

a. $r \bowtie_{(r.A=s.B) \wedge (r.A<s.C)} s$

Here we have extra conditions which can be checked after the join. Hence partitioned parallelism is useful.

b. $r \bowtie_{(r.A \geq (\lfloor s.B/20 \rfloor)*20) \wedge (r.A<((\lfloor s.B/20 \rfloor)+1)*20)} s$

This is a query in which an r tuple and an s tuple join with each other if they fall into the same range of values. Hence partitioned parallelism applies naturally to this scenario.

For both the queries, r should be partitioned on attribute A and s on attribute B.

21.10 Describe a good way to parallelize each of the following.

a. The difference operation
b. Aggregation by the **count** operation
c. Aggregation by the **count distinct** operation
d. Aggregation by the **avg** operation
e. Left outer join, if the join condition involves only equality
f. Left outer join, if the join condition involves comparisons other than equality
g. Full outer join, if the join condition involves comparisons other than equality

Answer:

a. We can parallelize the difference operation by partitioning the relations on all the attributes, and then computing differences locally at each processor. As in aggregation, the cost of transferring tuples during partitioning can be reduced by partially computing differences at each processor, before partitioning.

b. Let us refer to the group-by attribute as attribute A, and the attribute on which the aggregation function operates, as attribute B. **count** is performed just like **sum** (mentioned in the book) except that, a count of the number of values of attribute B for each value of attribute A) is transferred to the correct destination processor, instead of a sum. After partitioning, the partial counts from all the processors are added up locally at each processor to get the final result.

c. For this, partial counts cannot be computed locally before partitioning. Each processor instead transfers all unique B values for each A value to the correct destination processor. After partitioning, each processor locally counts the number of unique tuples for each value of A, and then outputs the final result.

d. This can again be implemented like **sum**, except that for each value of A, a **sum** of the B values as well as a **count** of the number of tuples in the group, is transferred during partitioning. Then each processor outputs its local result, by dividing the total sum by total number of tuples for each A value assigned to its partition.

e. This can be performed just like partitioned natural join. After partitioning, each processor computes the left outer join locally using any of the strategies of Chapter 13.

f. The left outer join can be computed using an extension of the Fragment-and-Replicate scheme to compute non equi-joins. Consider $r \supset\!\bowtie\ s$. The relations are partitioned, and $r \bowtie s$ is computed at each site. We also collect tuples from r that did not match any tuples from s; call the set of these dangling tuples at site i as d_i. After the above step is done at each site, for each fragment of r, we take the intersection of the d_i's from every processor in which the fragment of r was replicated. The intersections give the real set of dangling tuples; these tuples are padded with nulls and added to the result. The intersections themselves, followed by addition of padded tuples to the result, can be done in parallel by partitioning.

g. The algorithm is basically the same as above, except that when combining results, the processing of dangling tuples must done for both relations.

21.11 Describe the benefits and drawbacks of pipelined parallelism.

Answer:

- **Benefits:** No need to write intermediate relations to disk only to read them back immediately.
- **Drawbacks:**
  - **a.** Cannot take advantage of high degrees of parallelism, as typical queries do not have large number of operations.
  - **b.** Not possible to pipeline operators which need to look at all the input before producing any output.
  - **c.** Since each operation executes on a single processor, the most expensive ones take a long time to finish. Thus speed-up will be low inspite of parallelism.

21.12 Suppose you wished to handle a workload consisting of a large number of small transactions by using shared nothing parallelism.

a. Is intraquery parallelism required in such a situation? If not, why, and what form of parallism is appropriate?

b. What form of skew would be of signifance with such a workload?

c. Suppose most transactions accessed one *account* record, which includes an account type attribute, and an associated *account_type_master* record, which

provides information about the account type. How would you partition and/or replicate data to speed up transactions? You may assume that the *account_type_master* relation is rarely updated.

Answer: No answer.

CHAPTER 22

Distributed Databases

Exercises

22.13 Discuss the relative advantages of centralized and distributed databases.

Answer:

- A distributed database allows a user convenient and transparent access to data which is not stored at the site, while allowing each site control over its own local data. A distributed database can be made more reliable than a centralized system because if one site fails, the database can continue functioning, but if the centralized system fails, the database can no longer continue with its normal operation. Also, a distributed database allows parallel execution of queries and possibly splitting one query into many parts to increase throughput.
- A centralized system is easier to design and implement. A centralized system is cheaper to operate because messages do not have to be sent.

22.14 Explain how the following differ: fragmentation transparency, replication transparency, and location transparency.

Answer:

a. With fragmentation transparency, the user of the system is unaware of any fragmentation the system has implemented. A user may formulate queries against global relations and the system will perform the necessary transformation to generate correct output.

b. With replication transparency, the user is unaware of any replicated data. The system must prevent inconsistent operations on the data. This requires more complex concurrency control algorithms.

c. Location transparency means the user is unaware of where data are stored. The system must route data requests to the appropriate sites.

22.15 When is it useful to have replication or fragmentation of data? Explain your answer.
Answer: Replication is useful when there are many read-only transactions at different sites wanting access to the same data. They can all execute quickly in parallel, accessing local data. But updates become difficult with replication. Fragmentation is useful if transactions on different sites tend to access different parts of the database.

22.16 Explain the notions of transparency and autonomy. Why are these notions desirable from a human-factors standpoint?
Answer: Autonomy is the amount of control a single site has over the local database. It is important because users at that site want quick and correct access to local data items. This is especially true when one considers that local data will be most frequently accessed in a database. Transparency hides the distributed nature of the database. This is important because users should not be required to know about location, replication, fragmentation or other implementation aspects of the database.

22.17 If we apply a distributed version of the multiple-granularity protocol of Chapter 16 to a distributed database, the site responsible for the root of the DAG may become a bottleneck. Suppose we modify that protocol as follows:

- Only intention-mode locks are allowed on the root.
- All transactions are given all possible intention-mode locks on the root automatically.

Show that these modifications alleviate this problem without allowing any nonserializable schedules.
Answer: Serializability is assured since we have not changed the rules for the multiple granularity protocol. Since transactions are automatically granted all intention locks on the root node, and are not given other kinds of locks on it, there is no need to send any lock requests to the root. Thus the bottleneck is relieved.

22.18 Study and summarize the facilities that the database system you are using provides for dealing with inconsistent states that can be reached with lazy propagation of updates.
Answer: No Answer.

22.19 Discuss the advantages and disadvantages of the two methods that we presented in Section 22.5.2 for generating globally unique timestamps.
Answer: The centralized approach has the problem of a possible bottleneck at the central site and the problem of electing a new central site if it goes down. The distributed approach has the problem that many messages must be exchanges to keep the system fair, or one site can get ahead of all other sites and dominate the database.

22.20 Consider the relations

employee (*name*, *address*, *salary*, *plant_number*)
machine (*machine_number*, *type*, *plant_number*)

Assume that the *employee* relation is fragmented horizontally by *plant_number*, and that each fragment is stored locally at its corresponding plant site. Assume that the *machine* relation is stored in its entirety at the Armonk site. Describe a good strategy for processing each of the following queries.

a. Find all employees at the plant that contains machine number 1130.
b. Find all employees at plants that contain machines whose type is "milling machine."
c. Find all machines at the Almaden plant.
d. Find employee ⋈ machine.

Answer:

a. i. Perform $\Pi_{plant\_number}\ (\sigma_{machine\_number=1130}\ (machine))$ at Armonk.

ii. Send the query $\Pi_{name}\ (employee)$ to all site(s) which are in the result of the previous query.

iii. Those sites compute the answers.

iv. Union the answers at the destination site.

b. This strategy is the same as 0.a, except the first step should be to perform

$$\Pi_{plant\_number}\ (\sigma_{type=\text{“milling machine”}}\ (machine))$$ at Armonk.

c. i. Perform $\sigma_{plant\_number\ =\ x}\ (machine)$ at Armonk, where x is the plant _number for Almaden.

ii. Send the answers to the destination site.

d. Strategy 1:

i. Group *machine* at Armonk by plant number.

ii. Send the groups to the sites with the corresponding plant_number.

iii. Perform a local join between the local data and the received data.

iv. Union the results at the destination site.

Strategy 2:

Send the *machine* relation at Armonk, and all the fragments of the *employee* relation to the destination site. Then perform the join at the destination site.

There is parallelism in the join computation according to the first strategy but not in the second. Nevertheless, in a WAN the amount of data to be shipped is the main cost factor. We expect that each plant will have more than one machine, hence the result of the local join at each site will be a cross-product of the employee tuples and machines at that plant. This

cross-product's size is greater than the size of the *emplyee* fragment at that site. As a result the second strategy will result in less data shipping, and will be more efficient.

22.21 For each of the strategies of Exercise 22.20, state how your choice of a strategy depends on:

a. The site at which the query was entered
b. The site at which the result is desired

Answer:

a. Assuming that the cost of shipping the query itself is minimal, the site at which the query was submitted does not affect our strategy for query evaluation.
b. For the first query, we find out the plant numbers where the machine number 1130 is present, at Armonk. Then the employee tuples at all those plants are shipped to the destination site. We can see that this strategy is more or less independent of the destination site. The same can be said of the second query. For the third query, the selection is performed at Armonk and results shipped to the destination site. This stratgy is obviously independent of the destination site.

For the fourth query, we have two strategies. The first one performs local joins at all the plant sites and their results are unioned at the destination site. In the second strategy, the *machine* relation at Armonk as well as all the fragments of the *employee* relation are first shipped to the destination, where the join operation is performed. There is no obvious way to optimize these two strategies based on the destination site. In the answer to Exercise 22.20 we saw the reason why the second strategy is expected to result in less data shipping than the first. That reason is independent of destination site, and hence we can in general prefer strategy two to strategy one, regardless of the destination site.

22.22 Is the expression $r_i \ltimes r_j$ necessarily equal to $r_j \ltimes r_i$? Under what conditions does $r_i \ltimes r_j = r_j \ltimes r_i$ hold?
Answer: In general, $r_i \ltimes r_j \neq r_j \ltimes r_i$. This can be easily seen from Exercise 22.11, in which $r \ltimes s \neq s \ltimes r$. $r \ltimes s$ was given in 22.11, while

$s \ltimes r =$

C	D	E
3	4	5
3	6	8
2	3	2

By definition, $r_i \ltimes r_j = \Pi_{R_i}(r_i \bowtie r_j)$ and $r_j \ltimes r_i = \Pi_{R_j}(r_i \bowtie r_j)$, where R_i and R_j are the schemas of r_i and r_j respectively. For $\Pi_{R_i}(r_i \bowtie r_j)$ to be always equal to $\Pi_{R_j}(r_i \bowtie r_j)$, the schemas R_i and R_j must be the same.

22.23 Describe how LDAP can be used to provide multiple hierarchical views of data, without replicating the base level data.

Answer: This can be done using referrals. For example an organization may maintain its information about departments either by geography (i.e. all departments in a site of the the organization) or by structure (i.e. information about a department from all sites). These two hierarchies can be maintained by defining two different schemas with department information at a site as the base information. The entries in the two hierarchies will refer to the base information entry using referrals.

CHAPTER 23

Advanced Application Development

Exercises

23.6 Find out what all performance information your favorite database system provides. Look for at least the following: what queries are currently executing or executed recently, what resources each of them consumed (CPU and I/O), what fraction of page requests resulted in buffer misses (for each query, if available), and what locks have a high degree of contention. You may also be able to get information about CPU and I/O utilization from the operating system.
Answer: No answer.

23.7 **a.** What are the three broad levels at which a database system can be tuned to improve performance?
b. Give two examples of how tuning can be done, for each of the levels.

Answer:

a. We refer to performance tuning of a database system as the modification of some system components in order to improve transaction response times, or overall transaction throughput. Database systems can be tuned at various levels to enhance performance. viz.
i. Schema and transaction design
ii. Buffer manager and transaction manager
iii. Access and storage structures
iv. Hardware - disks, CPU, busses etc.

b. We describe some examples for performance tuning of some of the major components of the database system.
i. Tuning the schema -
In this chapter we have seen two examples of schema tuning, viz. vertical partition of a relation (or conversely - join of two relations),

and denormalization (or conversely - normalization). These examples reflect the general scenario, and ideas therein can be applied to tune other schemas.

ii. Tuning the transactions -

One approach used to speed-up query execution is to improve the its plan. Suppose that we need the natural join of two relations - say *account* and *depositor* from our sample bank database. A *sort–merge-join*(Section 13.5.4) on the attribute *account-number* may be quicker than a simple nested-loop join on the relations.

Other ways of tuning transactions are - breaking up long update transactions and combining related sets of queries into a single query. Generic examples for these approaches are given in this chapter.

For client-server systems, wherein the query has to be transmitted from client to server, the query transmission time itself may form a large fraction of the total query cost. Using *stored procedures* can significantly reduce the queries response time.

iii. Tuning the buffer manager -

The buffer manager can be made to increase or decrease the number of pages in the buffer according to changing page-fault rates. However, it must be noted that a larger number of pages may mean higher costs for latch management and maintenance of other data-structures like free-lists and page map tables.

iv. Tuning the transaction manager -

The transaction schedule affects system performance. A query that computes statistics for customers at each branch of the bank will need to scan the relations *account* and *depositor*. During these scans, no updates to any customer's balance will be allowed. Thus, the response time for the update transactions is high. Large queries are best executed when there are few updates, such as at night.

Checkpointing also incurs some cost. If recovery time is not critical, it is preferable to examine a long log (during recovery) rather than spend a lot of (checkpointing) time during normal operation. Hence it may be worthwhile to tune the checkpointing interval according to the expected rate of crashes and the required recovery time.

v. Tuning the access and storage structures -

A query's response time can be improved by creating an appropriate index on the relation. For example, consider a query in which a depositor enquires about her balance in a particular account. This query would result in the scan of the relation *account* if it has is no index on *account-number*. Similar indexing considerations also apply to computing joins. i.e an index on *account-number* in the *account* relation saves scanning *account* when a natural join of *account* is taken with *depositor*.

In contrast, performance of update transactions may suffer due to indexing. Let us assume that frequent updates to the balance are required. Also suppose that there is an index on *balance* (presumably for range queries) in *account*. Now, for each update to the value of the bal-

ance, the index too will have to be updated. In addition, concurrent updates to the index structure will require additional locking overheads. Note that the response time for each update would not be more if there were no index on *balance*.

The type of index chosen also affects performance. For a range query, an order preserving index (like B-trees) is better than a hashed index.

Clustering of data affects the response time for some queries. For example, assume that the tuples of the *account* relation are clustered on *branch-name*. Then the average execution time for a query that finds the total balance amount deposited at a particular branch can be improved. Even more benefit accrues from having a clustered index on *branch-name*.

If the database system has more than one disk, *declustering* of data will enable parallel access. Suppose that we have five disks and that in a hypothetical situation where each customer has five accounts and each account has a lot of historical information that needs to be accessed. Storing one account per customer per disk will enable parallel access to all accounts of a particular customer. Thus, the speed of a scan on *depositor* will increase about five-fold.

vi. Tuning the hardware -

The hardware for the database system typically consists of disks, the processor, and the interconnecting architecture (busses etc.). Each of these components may be a bottleneck and by increasing the number of disks or their block-sizes, or using a faster processor, or by improving the bus architecture, one may obtain an improvement in system performance.

23.8 When carrying out performance tuning, should you try to tune your hardware (by adding disks or memory) first, or should you try to tune your transactions (by adding indices or materialized views) first. Explain your answer.
Answer: No answer.

23.9 Suppose that your application has transactions that each access and update a single tuple in a very large relation stored in a B^+-tree file organization. Assume that all internal nodes of the B^+-tree are in memory, but only a very small fraction of the leaf pages can fit in memory. Explain how to calculate the minimum number of disks required to support a workload of 1000 transactions per second. Also calculate the required number of disks, using values for disk parameters given in Section 11.2.
Answer: No answer.

23.10 What is the motivation for splitting a long transaction into a series of small ones? What problems could arise as a result, and how can these problems be averted?
Answer: Long update transactions cause a lot of log information to be written, and hence extend the checkpointing interval and also the recovery time after a

crash. A transaction that performs many updates may even cause the system log to overflow before the transaction commits.

To avoid these problems with a long update transaction it may be advisable to break it up into smaller transactions. This can be seen as a *group* transaction being split into many small *mini-batch* transactions. The same effect is obtained by executing both the group transaction and the mini-batch transactions, which are scheduled in the order that their operations appear in the group transaction.

However, executing the mini-batch transactions in place of the group transaction has some costs, such as extra effort when recovering from system failures. Also, even if the group transaction satisfies the *isolation* requirement, the mini-batch may not. Thus the transaction manager can release the locks held by the mini-batch only when the last transaction in the mini-batch completes execution.

23.11 Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5 minute and 1 minute rule?
Answer: There will be no effect of these changes on the 5 minute or the 1 minute rule. The value of n, i.e. the frequency of page access at the break-even point, is proportional to the product of memory price and speed of disk access, other factors remaining constant. So when memory price falls by half and access speed doubles, n remains the same.

23.12 List at least 4 features of the TPC benchmarks that help make them realistic and dependable measures.
Answer: Some features that make the TPC benchmarks realistic and dependable are -

a. Ensuring full support for ACID properties of transactions,
b. Calculating the throughput by observing the *end-to-end* performance,
c. Making sizes of relations proportional to the expected rate of transaction arrival, and
d. Measuring the dollar cost per unit of throughput.

23.13 Why was the TPCD benchmark replaced by the TPCH and TPCR benchmarks?
Answer: Various TPCD queries can be significantly speeded up by using materialized views and other redundant information, but the overheads of using them should be properly accounted. Hence TPCR and TPCH were introduced as refinements of TPCD, both of which use same schema and workload. TPCR models periodic reporting queries, and the database running it is permited to use materialized views. TPCH, on the other hand, models ad hoc querying, and prohibits materialized views and other redundant information.

23.14 Explain what application characteristics would help you decide which of TPCC, TPCH, or TPCR best models the application.
Answer: No answer.

CHAPTER 24

Advanced Data Types and New Applications

Exercises

24.9 Will functional dependencies be preserved if a relation is converted to a temporal relation by adding a time attribute? How is the problem handled in a temporal database?

Answer: Functional dependencies may be violated when a relation is augmented to include a time attribute. For example, suppose we add a time attribute to the relation *account* in our sample bank database. The dependency *account-number*→*balance* may be violated since a customer's balance would keep changing with time.

To remedy this problem temporal database systems have a slightly different notion of functional dependency, called *temporal functional dependency*. For example, the temporal functional dependency *account-number* $\xrightarrow{\tau}$ *balance* over *Account-schema* means that for each instance *account* of *Account-schema*, all snapshots of *account* satisfy the functional dependency *account-number*→*balance*; i.e at any time instance, each account will have a unique bank balance corresponding to it.

24.10 Consider two-dimensional vector data where the data items do not overlap. Is it possible to convert such vector data to raster data? If so, what are the drawbacks of storing raster data obtained by such conversion, instead of the original vector data?

Answer: To convert non-overlapping vector data to raster data, we set the values for exactly those pixels that lie on any one of the data items (regions); the other pixels have a default value.

The disadvantages to this approach are: loss of precision in location information (since raster data loses resolution), a much higher storage requirement, and loss of abstract information (like the shape of a region).

24.11 Study the support for spatial data offered by the database system that you use, and implement the following:

a. A schema to represent the geographic location of restaurants along with features such as the cuisine served at the restaurant and the level of expensiveness.

b. A query to find moderately priced restaurants that serve Indian food and are within 5 miles of your house (assume any location for your house).

c. A query to find for each restaurant the distance from the nearest restaurant serving the same cuisine and with the same level of expensiveness.

Answer: No answer.

24.12 What problems can occur in a continuous-media system if data is delivered either too slowly or too fast?

Answer: Continuous media systems typically handle a large amount of data, which have to be delivered at a steady rate. Suppose the system provides the picture frames for a television set. The delivery rate of data from the system should be matched with the frame display rate of the TV set. If the delivery rate is too low, the display would periodically freeze or blank out, since there will be no new data to be displayed for some time. On the other hand, if the delivery rate is too high, the data buffer at the destination TV set will overflow causing loss of data; the lost data will never get displayed.

24.13 List three main features of mobile computing over wireless networks that are distinct from traditional distributed systems.

Answer: Some of the main distinguishing features are as follows.

- In distributed systems, disconnection of a host from the network is considered to be a *failure*, whereas allowing such disconnection is a *feature* of mobile systems.
- Distributed systems are usually centrally administered, whereas in mobile computing, each personal computer that participates in the system is administered by the user (owner) of the machine and there is little central administration, if any.
- In conventional distributed systems, each machine has a fixed location and network address(es). This is not true for mobile computers, and in fact, is antithetical to the very purpose of mobile computing.
- Queries made on a mobile computing system may involve the location and velocity of a host computer.
- Each computer in a distributed system is allowed to be arbitrarily large and may consume a lot of (almost) uninterrupted electrical power. Mobile systems typically have small computers that run on low wattage, short-lived batteries.

24.14 List three factors that need to be considered in query optimization for mobile computing that are not considered in traditional query optimizers.

Answer: The most important factor influencing the cost of query processing in

traditional database systems is that of disk I/O. However, in mobile computing, minimizing the amount of energy required to execute a query is an important task of a query optimizer. To reduce the consumption of energy (battery power), the query optimizer on a mobile computer must minimize the size and number of queries to be transmitted to remote computers as well as the time for which the disk is spinning.

In traditional database systems, the cost model typically does not include connection time and the amount of data transferred. However, mobile computer users are usually charged according to these parameters. Thus, these parameters should also be minimized by a mobile computer's query optimizer.

24.15 Give an example to show that the version-vector scheme does not ensure serializability. (Hint: Use the example from Exercise 24.8, with the assumption that documents 1 and 2 are available on both mobile computers A and B, and take into account the possibility that a document may be read without being updated.)

Answer: Consider the example given in the previous exercise. Suppose that both host A and host B are not connected to each other. Further, assume that identical copies of document 1 and document 2 are stored at host A and host B.

Let $\{X = 5\}$ be the initial contents of document 1, and $\{X = 10\}$ be the initial contents of document 2. Without loss of generality, let us assume that all version-vectors are initially zero.

Suppose host A updates the number its copy of document 1 with that in its copy of document 2. Thus, the contents of both the documents (at host A) are now $\{X = 10\}$. The version number $V_{1,A,A}$ is incremented to 1.

While host B is disconnected from host A, it updates the number in its copy of document 2 with that in its copy of document 1. Thus, the contents of both the documents (at host B) are now $\{X = 5\}$. The version number $V_{2,B,B}$ is incremented to 1.

Later, when host A and host B connect, they exchange version-vectors. The version-vector scheme updates the copy of document 1 at host B to $\{X = 10\}$, and the copy of document 2 at host A to $\{X = 5\}$. Thus, both copies of each document are identical, viz. document 1 contains $\{X = 10\}$ and document 2 contains $\{X = 5\}$.

However, note that a serial schedule for the two updates (one at host A and another at host B) would result in both documents having the *same* contents. Hence this example shows that the version-vector scheme does not ensure serializability.

CHAPTER 25

Advanced Transaction Processing

Exercises

25.9 Explain how a TP monitor manages memory and processor resources more effectively than a typical operating system.

Answer: In a typical OS, each client is represented by a process, which occupies a lot of memory. Also process multi-tasking over-head is high.

A TP monitor is more of a service provider, rather than an environment for executing client processes. The client processes run at their own sites, and they send requests to the TP monitor whenever they wish to avail of some service. The message is routed to the right server by the TP monitor, and the results of the service are sent back to the client.

The advantage of this scheme is that the same server process can be serving several clients simultaneously, by using multithreading. This saves memory space, and reduces CPU overheads on preserving ACID properties and on scheduling entire processes. Even without multi-threading, the TP monitor can dynamically change the number of servers running, depending on whatever factors affect good performance. All this is not possible with a typical OS setup.

25.10 Compare TP monitor features with those provided by Web servers supporting servlets (such servers have been nicknamed *TP-lite*).

Answer: No answer.

25.11 Consider the process of admitting new students at your university (or new employees at your organization).

a. Give a high-level picture of the workflow starting from the student application procedure.

b. Indicate acceptable termination states, and which steps involve human intervention.

c. Indicate possible errors (including deadline expiry) and how they are dealt with.
d. Study how much of the workflow has been automated at your university.

Answer: No answer.

25.12 Answer the following questions regarding electronic payment systems.

a. Explain why electronic transactions carried out using credit card numbers are insecure.
b. An alternative is to have an electronic payment gateway maintained by the credit card company, and the site receiving payment redirects customers to the gateway site to make the payment.
   i. Explain what benefits such a system offers if the gateway does not authenticate the user
   ii. Explain what further benefits are offered if the gateway has a mechanism to authenticate the user.
c. Some credit card companies offer a one-time-use credit card number as a more secure method of electronic payment. Customers connect to the credit card companys Web site to get the one-time-use number. Explain what benefit such a system offers, as compared to using regular credit card numbers. Also explain its benefits and drawbacks as compared to electronic payment gateways with authentication.
d. Does either of the above systems guarantee the same privacy that is available when payments are made in cash? Explain your answer.

Answer: yyyy

25.13 If the entire database fits in main memory, do we still need a database system to manage the data? Explain your answer.
Answer: Even if the entire database fits in main memory, a DBMS is needed to perform tasks like concurrency control, recovery, logging etc, in order to preserve ACID properties of transactions.

25.14 In the group-commit technique, how many transactions should be part of a group? Explain your answer.
Answer: As log-records are written to stable storage in multiples of a block, we should group transaction commits in such a way that the last block containing log-records for the current group is almost full.

25.15 In a database system using write-ahead logging, what is the worst-case number of disk accesses required to read a data item? Explain why this presents a problem to designers of real-time database systems.
Answer: In the worst case, a read can cause a buffer page to be written to disk (preceded by the corresponding log records), followed by the reading from disk of the page containing the data to be accessed. This takes two or more disk accesses, and the time taken is several orders of magnitude more than the main-memory reference required in the best case. Hence transaction execution-

time variance is very high and can be estimated only poorly. It is therefore difficult to plan schedules which need to finish within a deadline.

25.16 What is the purpose of compensating transactions? Present two examples of their use.

Answer: A compensating transaction is used to perform a semantic undo of changes made previously by committed transactions. For example, a person might deposit a check in their savings account. Then the database would be updated to reflect the new balance. Since it takes a few days for the check to clear, it might be discovered later that the check bounced, in which case a compensating transaction would be run to subtract the amount of the bounced check from the depositor's account. Another example of when a compensating transaction would be used is in a grading program. If a student's grade on an assignment is to be changed after it is recorded, a compensating program (usually an option of the grading program itself) is run to change the grade and redo averages, etc.

25.17 Explain the connections between a workflow and a long duration trans action.

Answer: No answer.