课后答案网 您最真诚的朋友

DATABASE MANAGEMENT SYSTEMS SOLUTIONS MANUAL THIRD EDITION

Raghu Ramakrishnan
University of Wisconsin
Madison, WI, USA

Johannes Gehrke
Cornell University
Ithaca, NY, USA

Jeff Derstadt, Scott Selikoff, and Lin Zhu
Cornell University
Ithaca, NY, USA

CONTENTS

PREFACE

It is not every question that deserves an answer.

Publius Syrus, 42 B.C.

I hope that most of the questions in this book deserve an answer. The set of questions is unusually extensive, and is designed to reinforce and deepen students' understanding of the concepts covered in each chapter. There is a strong emphasis on quantitative and problem-solving type exercises.

While I wrote some of the solutions myself, most were written originally by students in the database classes at Wisconsin. I'd like to thank the many students who helped in developing and checking the solutions to the exercises; this manual would not be available without their contributions. In alphabetical order: X. Bao, S. Biao, M. Chakrabarti, C. Chan, W. Chen, N. Cheung, D. Colwell, J. Derstadt, C. Fritz, V. Ganti, J. Gehrke, G. Glass, V. Gopalakrishnan, M. Higgins, T. Jasmin, M. Krishnaprasad, Y. Lin, C. Liu, M. Lusignan, H. Modi, S. Narayanan, D. Randolph, A. Ranganathan, J. Reminga, A. Therber, M. Thomas, Q. Wang, R. Wang, Z. Wang and J. Yuan. In addition, James Harrington and Martin Reames at Wisconsin and Nina Tang at Berkeley provided especially detailed feedback.

Several students contributed to each chapter's solutions, and answers were subsequently checked by me and by other students. This manual has been in use for several semesters. I hope that it is now mostly accurate, but I'm sure **it still contains errors and omissions**. If you are a student and you do not understand a particular solution, contact your instructor; it may be that you are missing something, but it may also be that the solution is incorrect! If you discover a bug, please send me mail (`raghu@cs.wisc.edu`) and I will update the manual promptly.

The latest version of this solutions manual is distributed freely through the Web; go to the home page mentioned below to obtain a copy.

For More Information

The home page for this book is at URL:

Database Management Systems Solutions Manual Third Edition

http://www.cs.wisc.edu/~dbbook

This page is frequently updated and contains information about the book, past and current users, and the software. This page also contains a link to all known errors in the book, the accompanying slides, and the software. *Since the solutions manual is distributed electronically, all known errors are immediately fixed and no list of errors is maintained.* Instructors are advised to visit this site periodically; they can also register at this site to be notified of important changes by email.

1

INTRODUCTION TO DATABASE SYSTEMS

Exercise 1.1 Why would you choose a database system instead of simply storing data in operating system files? When would it make sense *not* to use a database system?

Answer 1.1 A *database* is an integrated collection of data, usually so large that it has to be stored on secondary storage devices such as disks or tapes. This data can be maintained as a collection of operating system files, or stored in a *DBMS* (database management system). The advantages of using a DBMS are:

- *Data independence and efficient access.* Database application programs are independent of the details of data representation and storage. The conceptual and external schemas provide independence from physical storage decisions and logical design decisions respectively. In addition, a DBMS provides efficient storage and retrieval mechanisms, including support for very large files, index structures and query optimization.
- *Reduced application development time.* Since the DBMS provides several important functions required by applications, such as concurrency control and crash recovery, high level query facilities, etc., only application-specific code needs to be written. Even this is facilitated by suites of application development tools available from vendors for many database management systems.
- *Data integrity and security.* The view mechanism and the authorization facilities of a DBMS provide a powerful access control mechanism. Further, updates to the data that violate the semantics of the data can be detected and rejected by the DBMS if users specify the appropriate *integrity constraints*.
- *Data administration.* By providing a common umbrella for a large collection of data that is shared by several users, a DBMS facilitates maintenance and data administration tasks. A good DBA can effectively shield end-users from the chores of fine-tuning the data representation, periodic back-ups etc.

- *Concurrent access and crash recovery.* A DBMS supports the notion of a *transaction*, which is conceptually a single user's sequential program. Users can write transactions as if their programs were running in isolation against the database. The DBMS executes the actions of transactions in an interleaved fashion to obtain good performance, but schedules them in such a way as to ensure that conflicting operations are not permitted to proceed concurrently. Further, the DBMS maintains a continuous log of the changes to the data, and if there is a system crash, it can restore the database to a *transaction-consistent* state. That is, the actions of incomplete transactions are undone, so that the database state reflects only the actions of completed transactions. Thus, if each complete transaction, executing alone, maintains the consistency criteria, then the database state after recovery from a crash is consistent.

If these advantages are not important for the application at hand, using a collection of files may be a better solution because of the increased cost and overhead of purchasing and maintaining a DBMS.

Exercise 1.2 What is logical data independence and why is it important?

Answer 1.2 *Logical data independence* means that users are shielded from changes in the logical structure of the data, i.e., changes in the choice of relations to be stored. For example, if a relation Students(sid, sname, gpa) is replaced by Studentnames(sid, sname) and Studentgpas(sid, gpa) for some reason, application programs that operate on the Students relation can be shielded from this change by defining a view Students(sid, sname, gpa) (as the natural join of Studentnames and Studentgpas). Thus, application programs that refer to Students need not be changed when the relation Students is replaced by the other two relations. The only change is that instead of storing Students tuples, these tuples are computed as needed by using the view definition; this is transparent to the application program.

Exercise 1.3 Explain the difference between logical and physical data independence.

Answer 1.3 Logical data independence means that users are shielded from changes in the logical structure of the data, while physical data independence insulates users from changes in the physical storage of the data. We saw an example of logical data independence in the answer to Exercise 1.2. Consider the Students relation from that example (and now assume that it is not replaced by the two smaller relations). We could choose to store Students tuples in a heap file, with a clustered index on the sname field. Alternatively, we could choose to store it with an index on the gpa field, or to create indexes on both fields, or to store it as a file sorted by gpa. These storage alternatives are not visible to users, except in terms of improved performance, since they simply see a relation as a set of tuples. This is what is meant by physical data independence.

Exercise 1.4 Explain the difference between external, internal, and conceptual schemas. How are these different schema layers related to the concepts of logical and physical data independence?

Answer 1.4 External schemas allows data access to be customized (and authorized) at the level of individual users or groups of users. Conceptual (logical) schemas describes all the data that is actually stored in the database. While there are several views for a given database, there is exactly one conceptual schema to *all* users. Internal (physical) schemas summarize how the relations described in the conceptual schema are actually stored on disk (or other physical media).

External schemas provide logical data independence, while conceptual schemas offer physical data independence.

Exercise 1.5 What are the responsibilities of a DBA? If we assume that the DBA is never interested in running his or her own queries, does the DBA still need to understand query optimization? Why?

Answer 1.5 The DBA is responsible for:

- *Designing the logical and physical schemas, as well as widely-used portions of the external schema.*
- *Security and authorization.*
- *Data availability and recovery from failures.*
- *Database tuning:* The DBA is responsible for evolving the database, in particular the conceptual and physical schemas, to ensure adequate performance as user requirements change.

A DBA needs to understand query optimization even if s/he is not interested in running his or her own queries because some of these responsibilities (database design and tuning) are related to query optimization. Unless the DBA understands the performance needs of widely used queries, and how the DBMS will optimize and execute these queries, good design and tuning decisions cannot be made.

Exercise 1.6 Scrooge McNugget wants to store information (names, addresses, descriptions of embarrassing moments, etc.) about the many ducks on his payroll. Not surprisingly, the volume of data compels him to buy a database system. To save money, he wants to buy one with the fewest possible features, and he plans to run it as a stand-alone application on his PC clone. Of course, Scrooge does not plan to share his list with anyone. Indicate which of the following DBMS features Scrooge should pay for; in each case, also indicate why Scrooge should (or should not) pay for that feature in the system he buys.

1. A security facility.
2. Concurrency control.
3. Crash recovery.
4. A view mechanism.
5. A query language.

Answer 1.6 Let us discuss the individual features in detail.

- A security facility is necessary because Scrooge does not plan to share his list with anyone else. Even though he is running it on his stand-alone PC, a rival duckster could break in and attempt to query his database. The database's security features would foil the intruder.
- Concurrency control is not needed because only he uses the database.
- Crash recovery is essential for any database; Scrooge would not want to lose his data if the power was interrupted while he was using the system.
- A view mechanism is needed. Scrooge could use this to develop "custom screens" that he could conveniently bring up without writing long queries repeatedly.
- A query language is necessary since Scrooge must be able to analyze the dark secrets of his victims. In particular, the query language is also used to define views.

Exercise 1.7 Which of the following plays an important role in *representing* information about the real world in a database? Explain briefly.

1. The data definition language.
2. The data manipulation language.
3. The buffer manager.
4. The data model.

Answer 1.7 Let us discuss the choices in turn.

- The data definition language is important in representing information because it is used to describe external and logical schemas.

- The data manipulation language is used to access and update data; it is not important for representing the data. (Of course, the data manipulation language must be aware of how data is represented, and reflects this in the constructs that it supports.)
- The buffer manager is not very important for representation because it brings arbitrary disk pages into main memory, independent of any data representation.
- The data model is fundamental to representing information. The data model determines what data representation mechanisms are supported by the DBMS. The data definition language is just the specific set of language constructs available to describe an actual application's data in terms of the *data model*.

Exercise 1.8 Describe the structure of a DBMS. If your operating system is upgraded to support some new functions on OS files (e.g., the ability to force some sequence of bytes to disk), which layer(s) of the DBMS would you have to rewrite to take advantage of these new functions?

Answer 1.8 The architecture of a relational DBMS typically consists of a layer that manages space on disk, a layer that manages available main memory and brings disk pages into memory as needed, a layer that supports the abstractions of files and index structures, a layer that implements relational operators, and a layer that parses and optimizes queries and produces an execution plan in terms of relational operators. In addition, there is support for concurrency control and recovery, which interacts with the buffer management and access method layers.

The disk space management layer has to be rewritten to take advantage of the new functions on OS files. It is likely that the buffer management layer will also be affected.

Exercise 1.9 Answer the following questions:

1. What is a transaction?
2. Why does a DBMS interleave the actions of different transactions instead of executing transactions one after the other?
3. What must a user guarantee with respect to a transaction and database consistency? What should a DBMS guarantee with respect to concurrent execution of several transactions and database consistency?
4. Explain the strict two-phase locking protocol.
5. What is the WAL property, and why is it important?

Answer 1.9 Let us answer each question in turn:

1. A transaction is any one execution of a user program in a DBMS. This is the basic unit of change in a DBMS.

2. A DBMS is typically shared among many users. Transactions from these users can be interleaved to improve the execution time of users' queries. By interleaving queries, users do not have to wait for other user's transactions to complete fully before their own transaction begins. Without interleaving, if user A begins a transaction that will take 10 seconds to complete, and user B wants to begin a transaction, user B would have to wait an additional 10 seconds for user A's transaction to complete before the database would begin processing user B's request.

3. A user must guarantee that his or her transaction does not corrupt data or insert nonsense in the database. For example, in a banking database, a user must guarantee that a cash withdraw transaction accurately models the amount a person removes from his or her account. A database application would be worthless if a person removed 20 dollars from an ATM but the transaction set their balance to zero! A DBMS must guarantee that transactions are executed fully and independently of other transactions. An essential property of a DBMS is that a transaction should execute atomically, or as if it is the only transaction running. Also, transactions will either complete fully, or will be aborted and the database returned to it's initial state. This ensures that the database remains consistent.

4. Strict two-phase locking uses shared and exclusive locks to protect data. A transaction must hold all the required locks before executing, and does not release any lock until the transaction has completely finished.

5. The WAL property affects the logging strategy in a DBMS. The WAL, Write-Ahead Log, property states that each write action must be recorded in the log (on disk) before the corresponding change is reflected in the database itself. This protects the database from system crashes that happen during a transaction's execution. By recording the change in a log before the change is truly made, the database knows to undo the changes to recover from a system crash. Otherwise, if the system crashes just after making the change in the database but before the database logs the change, then the database would not be able to detect his change during crash recovery.

2

INTRODUCTION TO DATABASE DESIGN

Exercise 2.1 Explain the following terms briefly: *attribute, domain, entity, relationship, entity set, relationship set, one-to-many relationship, many-to-many relationship, participation constraint, overlap constraint, covering constraint, weak entity set, aggregation*, and *role indicator*.

Answer 2.1 Term explanations:

- *Attribute* - a property or description of an entity. A toy department employee entity could have attributes describing the employee's name, salary, and years of service.
- *Domain* - a set of possible values for an attribute.
- *Entity* - an object in the real world that is distinguishable from other objects such as the green dragon toy.
- *Relationship* - an association among two or more entities.
- *Entity set* - a collection of similar entities such as all of the toys in the toy department.
- *Relationship set* - a collection of similar relationships
- *One-to-many relationship* - a key constraint that indicates that one entity can be associated with many of another entity. An example of a one-to-many relationship is when an employee can work for only one department, and a department can have many employees.
- *Many-to-many relationship* - a key constraint that indicates that many of one entity can be associated with many of another entity. An example of a many-to-many relationship is employees and their hobbies: a person can have many different hobbies, and many people can have the same hobby.

- *Participation constraint* - a participation constraint determines whether relationships must involve certain entities. An example is if every department entity has a manager entity. Participation constraints can either be total or partial. A total participation constraint says that every department has a manager. A partial participation constraint says that every employee does not have to be a manager.
- *Overlap constraint* - within an ISA hierarchy, an overlap constraint determines whether or not two subclasses can contain the same entity.
- *Covering constraint* - within an ISA hierarchy, a covering constraint determines where the entities in the subclasses collectively include all entities in the superclass. For example, with an Employees entity set with subclasses HourlyEmployee and SalaryEmployee, does every Employee entity necessarily have to be within either HourlyEmployee or SalaryEmployee?
- *Weak entity set* - an entity that cannot be identified uniquely without considering some primary key attributes of another identifying owner entity. An example is including Dependent information for employees for insurance purposes.
- *Aggregation* - a feature of the entity relationship model that allows a relationship set to participate in another relationship set. This is indicated on an ER diagram by drawing a dashed box around the aggregation.
- *Role indicator* - If an entity set plays more than one role, role indicators describe the different purpose in the relationship. An example is a single Employee entity set with a relation Reports-To that relates supervisors and subordinates.

Exercise 2.2 A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each situation, draw an ER diagram that describes it (assuming no further constraints hold).

1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume this condition applies in all subsequent questions.)
3. Every professor must teach some course.
4. Every professor teaches exactly one course (no more, no less).
5. Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.

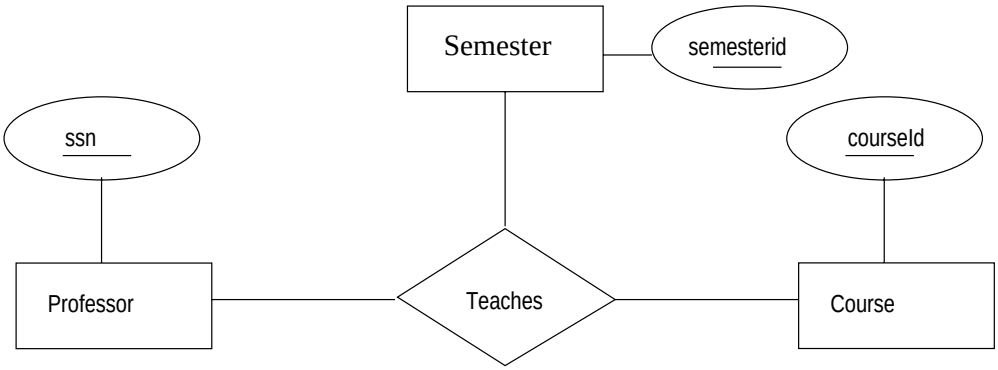


Figure 2.1 ER Diagram for Exercise 2.2, Part 1

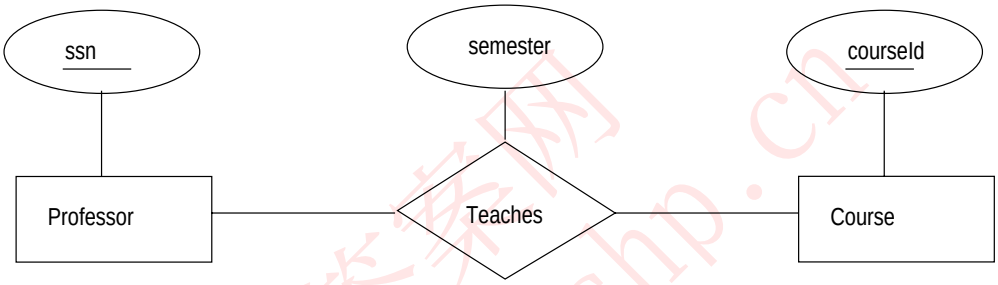


Figure 2.2 ER Diagram for Exercise 2.2, Part 2

6. Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this situation, introducing additional entity sets and relationship sets if necessary.

Answer 2.2 1. The ER diagram is shown in Figure 2.1.

2. The ER diagram is shown in Figure 2.2.
3. The ER diagram is shown in Figure 2.3.
4. The ER diagram is shown in Figure 2.4.
5. The ER diagram is shown in Figure 2.5.
6. The E.R. diagram is shown in Figure 2.6. An additional entity set called Group is introduced to identify the professors who team to teach a course. We assume that only the latest offering of a course needs to be recorded.

Exercise 2.3 Consider the following information about a university database:

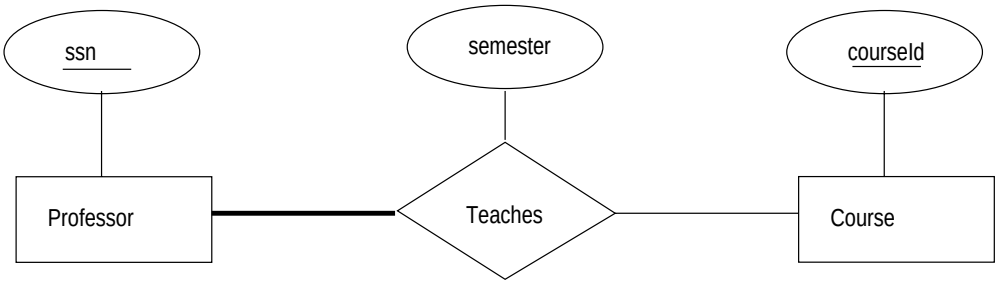


Figure 2.3 ER Diagram for Exercise 2.2, Part 3

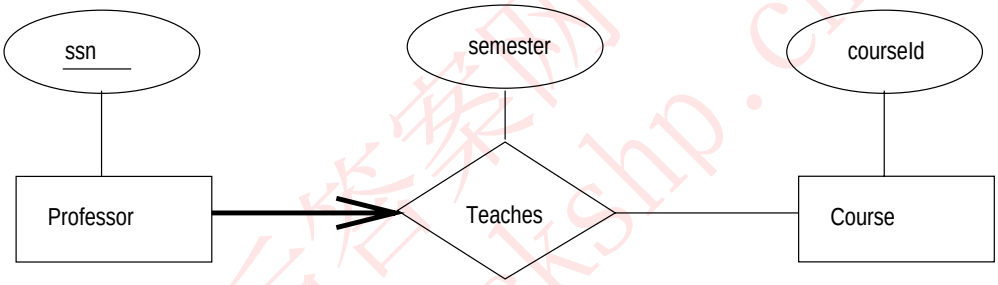


Figure 2.4 ER Diagram for Exercise 2.2, Part 4

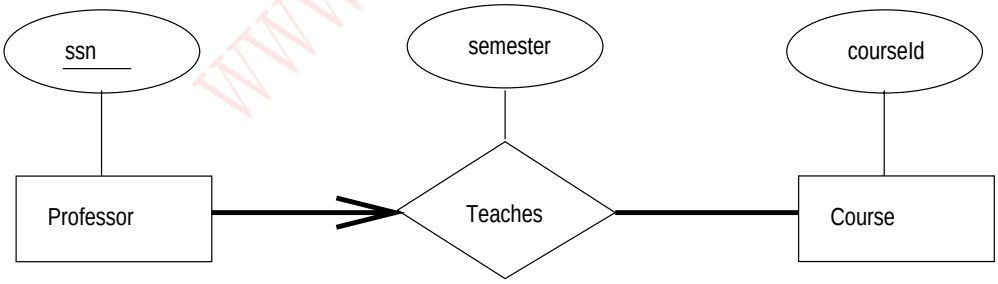


Figure 2.5 ER Diagram for Exercise 2.2, Part 5

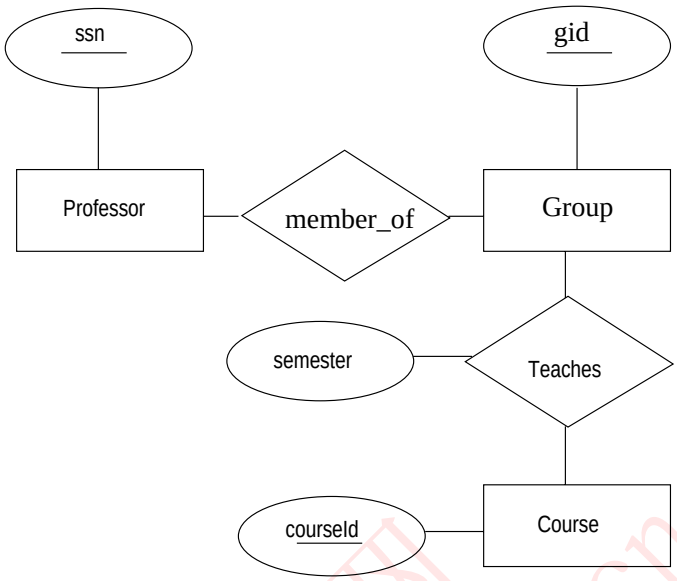


Figure 2.6 ER Diagram for Exercise 2.2, Part 6

- Professors have an SSN, a name, an age, a rank, and a research specialty.
- Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
- Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
- Each project is managed by one professor (known as the project's principal investigator).
- Each project is worked on by one or more professors (known as the project's co-investigators).
- Professors can manage and/or work on multiple projects.
- Each project is worked on by one or more graduate students (known as the project's research assistants).
- When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
- Departments have a department number, a department name, and a main office.
- Departments have a professor (known as the chairman) who runs the department.

- Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
- Graduate students have one major department in which they are working on their degree.
- Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here; that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

Answer 2.3 The ER diagram is shown in Figure 2.7.

Exercise 2.4 A company database needs to store information about employees (identified by *ssn*, with *salary* and *phone* as attributes), departments (identified by *dno*, with *dname* and *budget* as attributes), and children of employees (with *name* and *age* as attributes). Employees *work* in departments; each department is *managed by* an employee; a child must be identified uniquely by *name* when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.

Answer 2.4 The ER diagram is shown in Figure 2.8.

Exercise 2.5 Notown Records has decided to store information about musicians who perform on its albums (as well as other company data) in a database. The company has wisely chosen to hire you as a database designer (at your usual consulting fee of $2500/day).

- Each musician that records at Notown has an SSN, a name, an address, and a phone number. Poorly paid musicians often share the same address, and no address has more than one phone.
- Each instrument used in songs recorded at Notown has a unique identification number, a name (e.g., guitar, synthesizer, flute) and a musical key (e.g., C, B-flat, E-flat).
- Each album recorded on the Notown label has a unique identification number, a title, a copyright date, a format (e.g., CD or MC), and an album identifier.
- Each song recorded at Notown has a title and an author.

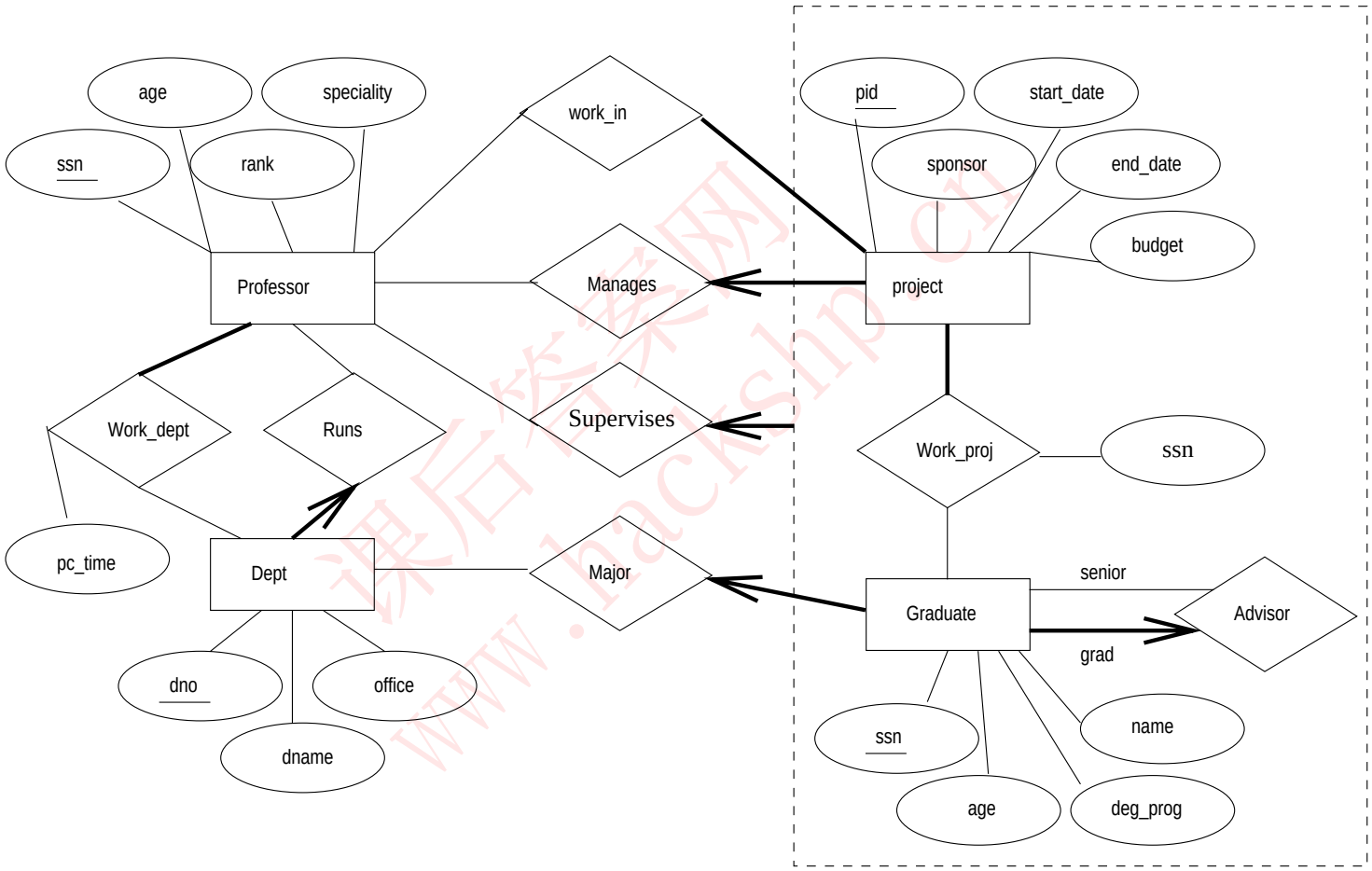


Figure 2.7 ER Diagram for Exercise 2.3

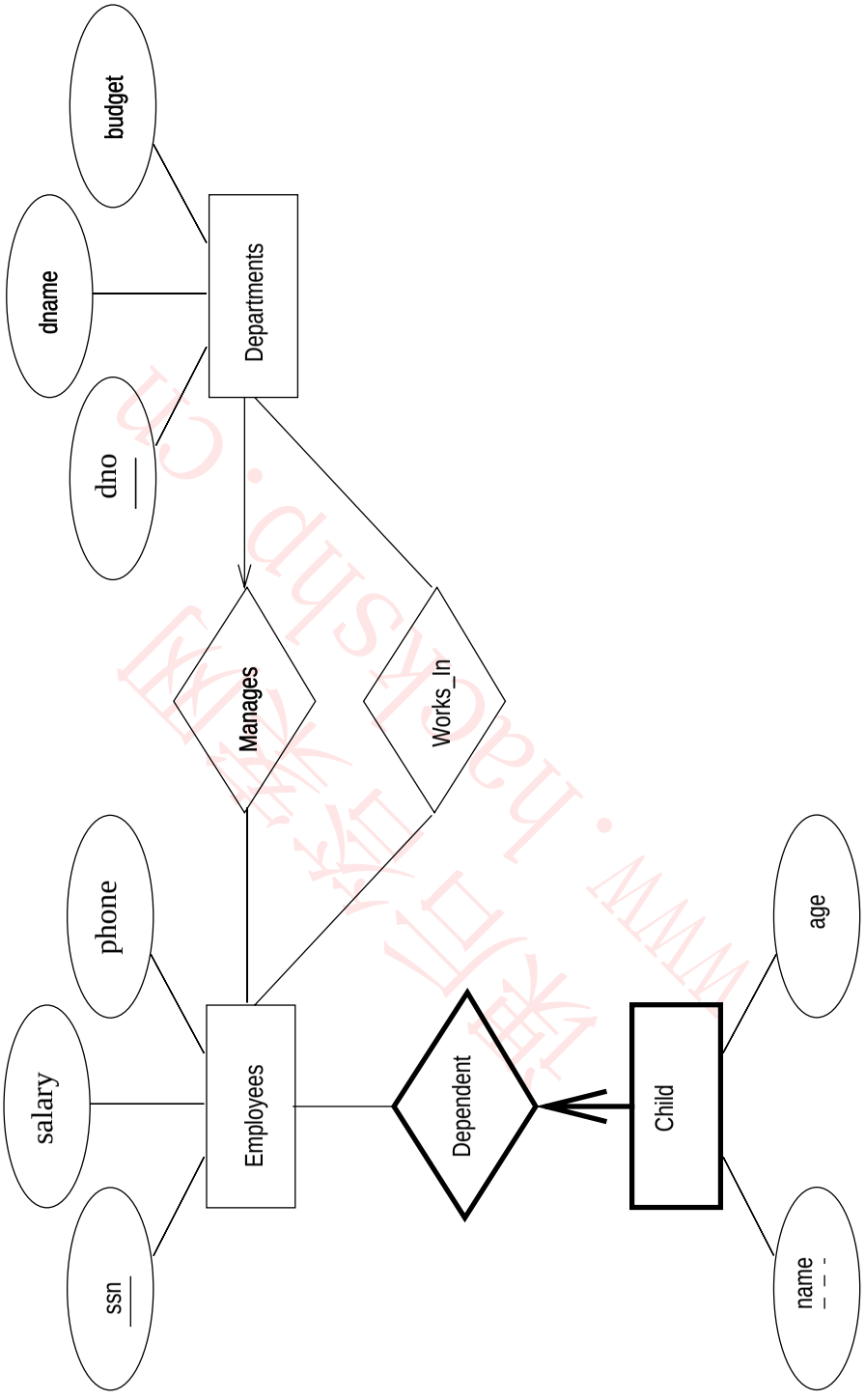


Figure 2.8 ER Diagram for Exercise 2.4

- Each musician may play several instruments, and a given instrument may be played by several musicians.
- Each album has a number of songs on it, but no song may appear on more than one album.
- Each song is performed by one or more musicians, and a musician may perform a number of songs.
- Each album has exactly one musician who acts as its producer. A musician may produce several albums, of course.

Design a conceptual schema for Notown and draw an ER diagram for your schema. The preceding information describes the situation that the Notown database must model. Be sure to indicate all key and cardinality constraints and any assumptions you make. Identify any constraints you are unable to capture in the ER diagram and briefly explain why you could not express them.

Answer 2.5 The ER diagram is shown in Figure 2.9.

Exercise 2.6 Computer Sciences Department frequent fliers have been complaining to Dane County Airport officials about the poor organization at the airport. As a result, the officials decided that all information related to the airport should be organized using a DBMS, and you have been hired to design the database. Your first task is to organize the information about all the airplanes stationed and maintained at the airport. The relevant information is as follows:

- Every airplane has a registration number, and each airplane is of a specific model.
- The airport accommodates a number of airplane models, and each model is identified by a model number (e.g., DC-10) and has a capacity and a weight.
- A number of technicians work at the airport. You need to store the name, SSN, address, phone number, and salary of each technician.
- Each technician is an expert on one or more plane model(s), and his or her expertise may overlap with that of other technicians. This information about technicians must also be recorded.
- Traffic controllers must have an annual medical examination. For each traffic controller, you must store the date of the most recent exam.
- All airport employees (including technicians) belong to a union. You must store the union membership number of each employee. You can assume that each employee is uniquely identified by a social security number.

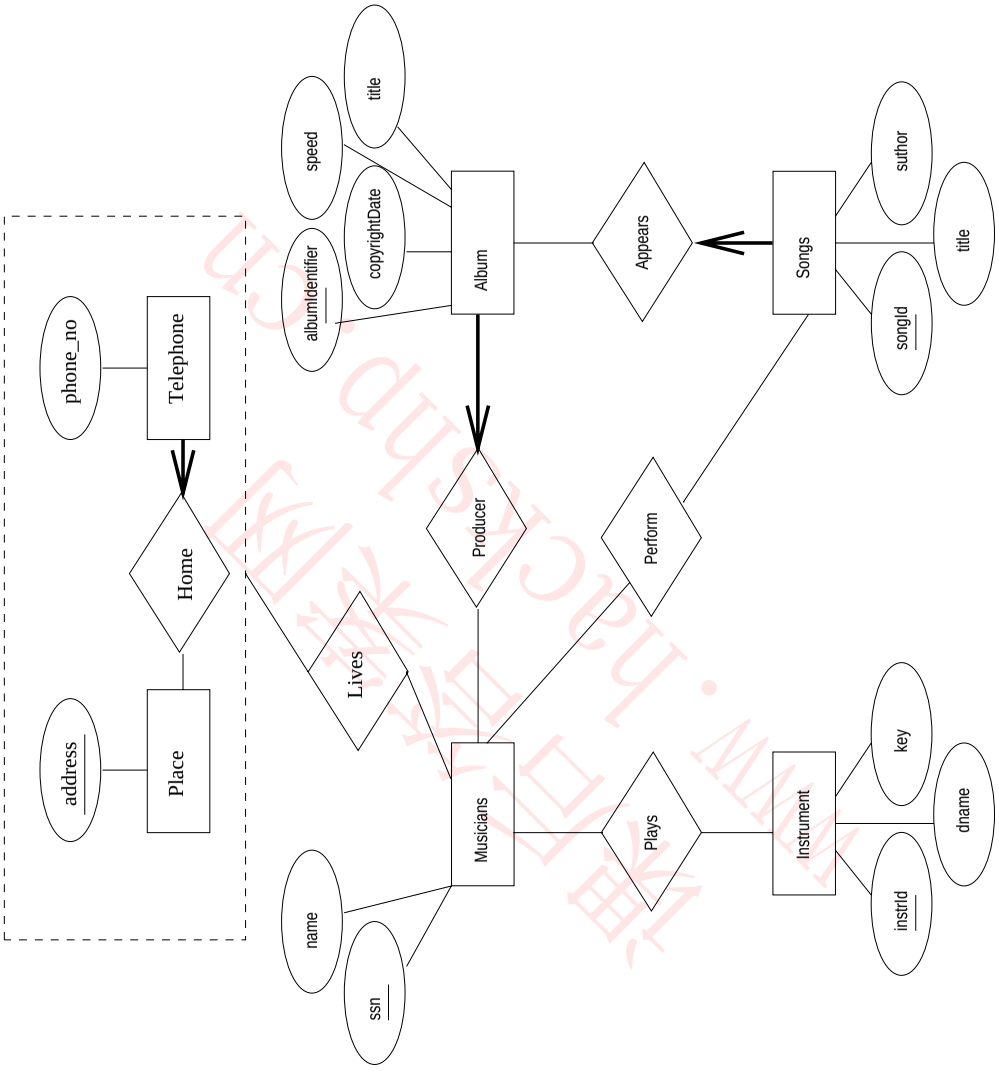


Figure 2.9 ER Diagram for Exercise 2.5

- The airport has a number of tests that are used periodically to ensure that airplanes are still airworthy. Each test has a Federal Aviation Administration (FAA) test number, a name, and a maximum possible score.
- The FAA requires the airport to keep track of each time a given airplane is tested by a given technician using a given test. For each testing event, the information needed is the date, the number of hours the technician spent doing the test, and the score the airplane received on the test.

1. Draw an ER diagram for the airport database. Be sure to indicate the various attributes of each entity and relationship set; also specify the key and participation constraints for each relationship set. Specify any necessary overlap and covering constraints as well (in English).
2. The FAA passes a regulation that tests on a plane must be conducted by a technician who is an expert on that model. How would you express this constraint in the ER diagram? If you cannot express it, explain briefly.

Answer 2.6 The ER diagram is shown in Figure 2.10.

1. Since all airline employees belong to a union, there is a covering constraint on the Employees ISA hierarchy.
2. You cannot note the expert technician constraint the FAA requires in an ER diagram. There is no notation for equivalence in an ER diagram and this is what is needed: the Expert relation must be equivalent to the Type relation.

Exercise 2.7 The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicine if you design its database. Given the rising cost of health care, you agree. Here's the information that you gather:

- Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
- Each pharmaceutical company is identified by name and has a phone number.
- For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.
- Each pharmacy has a name, address, and phone number.

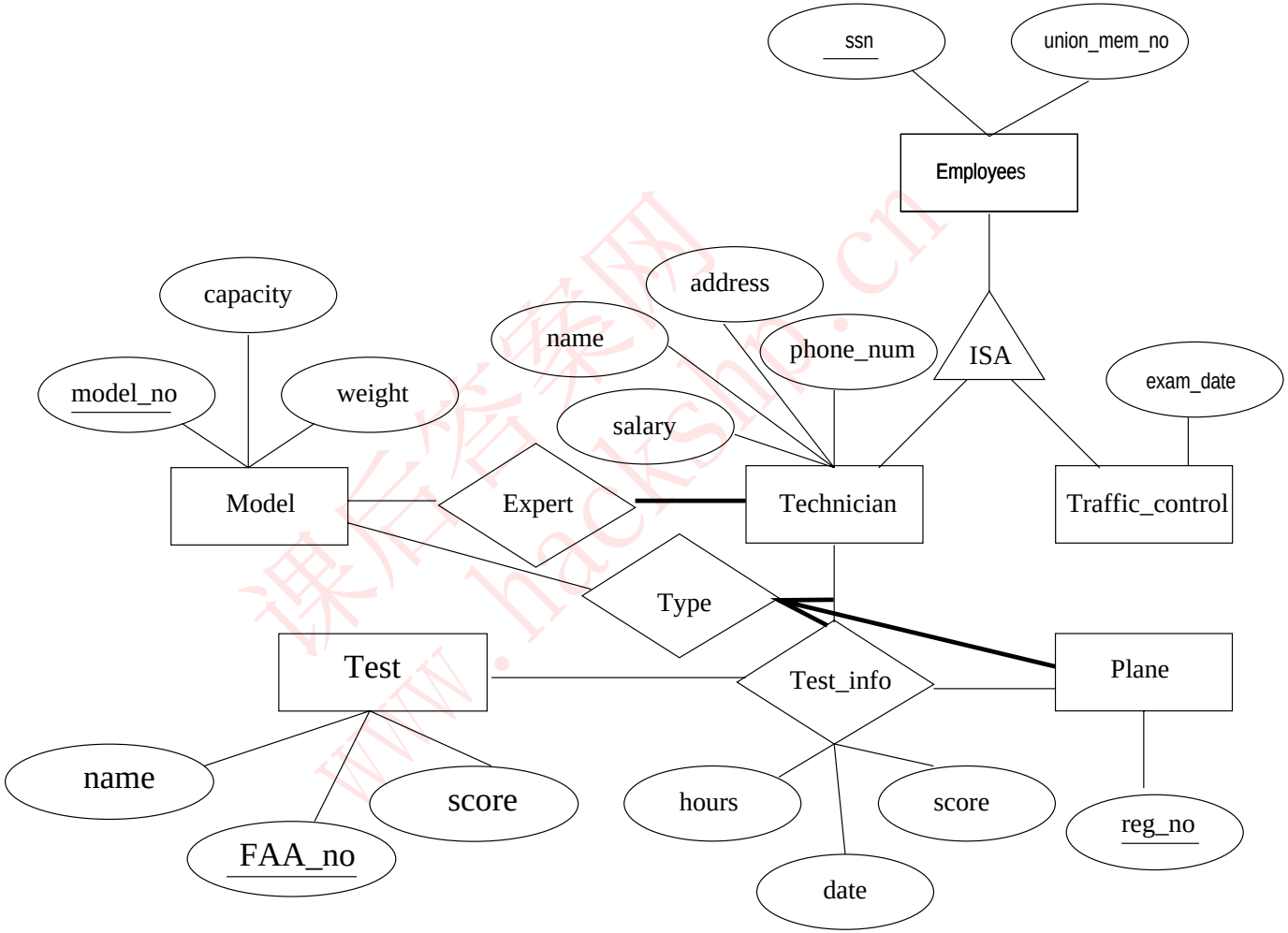


Figure 2.10 ER Diagram for Exercise 2.6

- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.
- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and a quantity associated with it. You can assume that, if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.
- Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.
- Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

1. Draw an ER diagram that captures the preceding information. Identify any constraints not captured by the ER diagram.
2. How would your design change if each drug must be sold at a fixed price by all pharmacies?
3. How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored.

Answer 2.7 1. The ER diagram is shown in Figure 2.11.

2. If the drug is to be sold at a fixed price we can add the price attribute to the Drug entity set and eliminate the price from the Sell relationship set.
3. The date information can no longer be modeled as an attribute of Prescription. We have to create a new entity set called Prescription_date and make Prescription a 4-way relationship set that involves this additional entity set.

Exercise 2.8 Although you always wanted to be an artist, you ended up being an expert on databases because you love to cook data and you somehow confused *database* with *data baste*. Your old love is still there, however, so you set up a database company, ArtBase, that builds a product for art galleries. The core of this product is a database with a schema that captures all the information that galleries need to maintain. Galleries keep information about artists, their names (which are unique), birthplaces, age, and style of art. For each piece of artwork, the artist, the year it was made, its unique

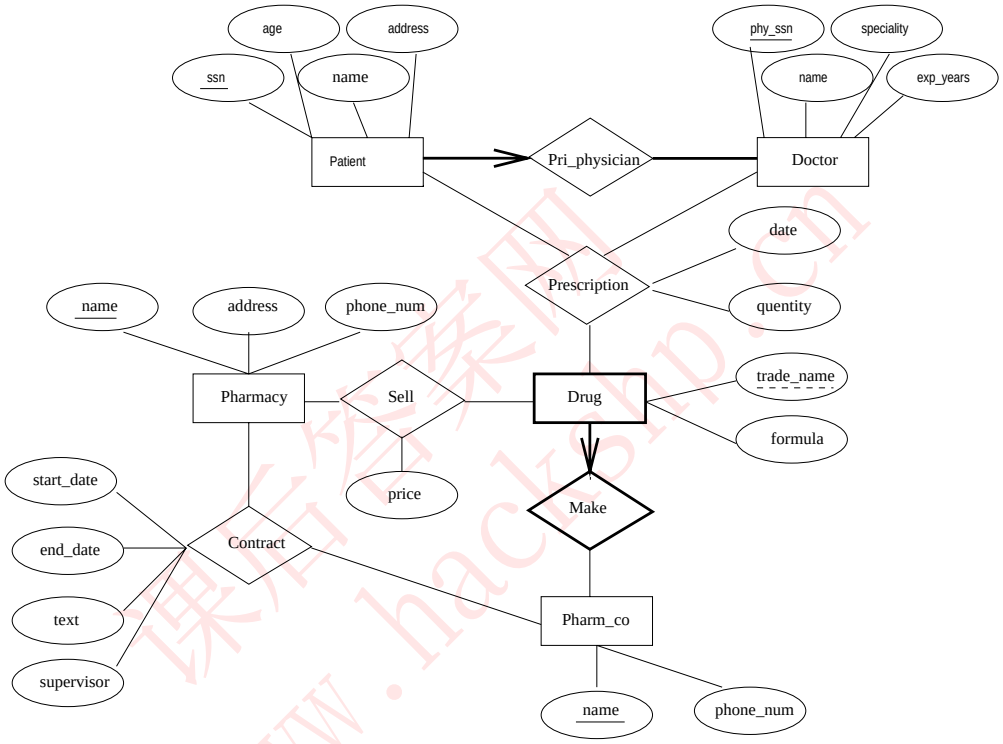


Figure 2.11 ER Diagram for Exercise 2.7

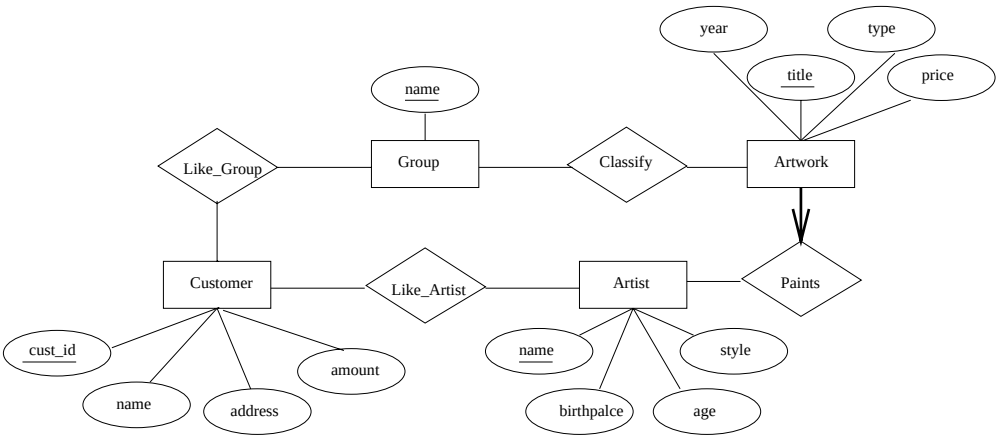


Figure 2.12 ER Diagram for Exercise 2.8

title, its type of art (e.g., painting, lithograph, sculpture, photograph), and its price must be stored. Pieces of artwork are also classified into groups of various kinds, for example, portraits, still lifes, works by Picasso, or works of the 19th century; a given piece may belong to more than one group. Each group is identified by a name (like those just given) that describes the group. Finally, galleries keep information about customers. For each customer, galleries keep that person's unique name, address, total amount of dollars spent in the gallery (very important!), and the artists and groups of art that the customer tends to like.

Draw the ER diagram for the database.

Answer 2.8 The ER diagram is shown in Figure 2.12.

Exercise 2.9 Answer the following questions.

- Explain the following terms briefly: *UML, use case diagrams, statechart diagrams, class diagrams, database diagrams, component diagrams*, and *deployment diagrams.*
- Explain the relationship between ER diagrams and UML.

Answer 2.9 Not yet done.

3

THE RELATIONAL MODEL

Exercise 3.1 Define the following terms: *relation schema, relational database schema, domain, attribute, attribute domain, relation instance, relation cardinality*, and *relation degree.*

Answer 3.1 A *relation schema* can be thought of as the basic information describing a table or *relation*. This includes a set of column names, the data types associated with each column, and the name associated with the entire table. For example, a relation schema for the relation called Students could be expressed using the following representation:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)

There are five fields or columns, with names and types as shown above.

A *relational database schema* is a collection of relation schemas, describing one or more relations.

Domain is synonymous with *data type*. *Attributes* can be thought of as columns in a table. Therefore, an *attribute domain* refers to the data type associated with a column.

A *relation instance* is a set of tuples (also known as *rows* or *records*) that each conform to the schema of the relation.

The *relation cardinality* is the number of tuples in the relation.

The *relation degree* is the number of fields (or columns) in the relation.

Exercise 3.2 How many distinct tuples are in a relation instance with cardinality 22?

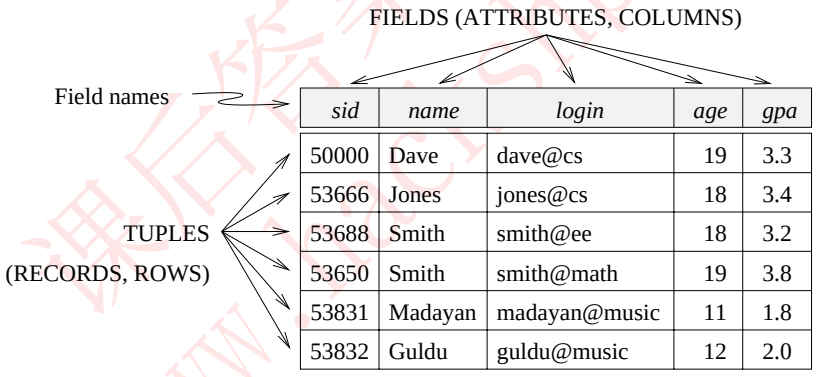
Answer 3.2 Since a relation is formally defined as a *set* of tuples, if the cardinality is 22 (i.e., there are 22 tuples), there must be 22 distinct tuples.

Exercise 3.3 Does the relational model, as seen by an SQL query writer, provide physical and logical data independence? Explain.

Answer 3.3 The user of SQL has no idea how the data is physically represented in the machine. He or she relies entirely on the relation abstraction for querying. Physical data independence is therefore assured. Since a user can define views, logical data independence can also be achieved by using view definitions to hide changes in the conceptual schema.

Exercise 3.4 What is the difference between a candidate key and the primary key for a given relation? What is a superkey?

Answer 3.4 The *primary key* is the key selected by the DBA from among the group of *candidate keys*, all of which uniquely identify a tuple. A *superkey* is a set of attributes that contains a key.



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance $S1$ of the Students Relation

Exercise 3.5 Consider the instance of the Students relation shown in Figure 3.1.

1. Give an example of an attribute (or set of attributes) that you can deduce is *not* a candidate key, based on this instance being legal.
2. Is there any example of an attribute (or set of attributes) that you can deduce *is* a candidate key, based on this instance being legal?

Answer 3.5 Examples of non-candidate keys include the following: {name}, {age}. (Note that {gpa} can *not* be declared as a non-candidate key from this evidence alone even though common sense tells us that clearly more than one student could have the same grade point average.)

You cannot determine a key of a relation given only one instance of the relation. The fact that the instance is "legal" is immaterial. A candidate key, as defined here, *is a key*, not something that only *might* be a key. The instance shown is just one possible "snapshot" of the relation. At other times, the same relation may have an instance (or snapshot) that contains a totally different set of tuples, and we cannot make predictions about those instances based only upon the instance that we are given.

Exercise 3.6 What is a foreign key constraint? Why are such constraints important? What is referential integrity?

Answer 3.6 A *foreign key* constraint is a statement of the form that one or more fields of a relation, say R, together *refer* to a second relation, say S. That is, the values in these fields of a tuple in R are either *null*, or uniquely identify some tuple in S. Thus, these fields of R should be a (candidate or primary) key. For example, a student, uniquely identified by an *sid*, enrolled in a class must also be registered in the school's student database (say, in a relation called Students). Therefore, the *sid* of a legal entry in the Class_Enrollment relation must match an existing *sid* in the Students relation.

Foreign key constraints are important because they provide safeguards for insuring the integrity of data. Users are alerted/thwarted when they try to do something that does not make sense. This can help minimize errors in application programs or in data-entry.

Referential integrity means all foreign key constraints are enforced.

Exercise 3.7 Consider the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In defined in Section 1.5.2.

1. List all the foreign key constraints among these relations.
2. Give an example of a (plausible) constraint involving one or more of these relations that is not a primary key or foreign key constraint.

Answer 3.7 There is no reason for a foreign key constraint (FKC) on the Students, Faculty, Courses, or Rooms relations. These are the most basic relations and must be free-standing. Special care must be given to entering data into these base relations.

In the Enrolled relation, *sid* and *cid* should both have FKCs placed on them. (Real students must be enrolled in real courses.) Also, since real teachers must teach real courses, both the *fid* and the *cid* fields in the Teaches relation should have FKCs. Finally, Meets_In should place FKCs on both the *cid* and *rno* fields.

It would probably be wise to enforce a few other constraints on this DBMS: the length of *sid*, *cid*, and *fid* could be standardized; checksums could be added to these identification numbers; limits could be placed on the size of the numbers entered into the credits, capacity, and salary fields; an enumerated type should be assigned to the grade field (preventing a student from receiving a grade of *G*, among other things); etc.

Exercise 3.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pcttime:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *budget:* `real`, *managerid:* `integer`)

1. Give an example of a foreign key constraint that involves the Dept relation. What are the options for enforcing this constraint when a user attempts to delete a Dept tuple?
2. Write the SQL statements required to create the preceding relations, including appropriate versions of all primary and foreign key integrity constraints.
3. Define the Dept relation in SQL so that every department is guaranteed to have a manager.
4. Write an SQL statement to add John Doe as an employee with $eid = 101$, $age = 32$ and $salary = 15,000$.
5. Write an SQL statement to give every employee a 10 percent raise.
6. Write an SQL statement to delete the Toy department. Given the referential integrity constraints you chose for this schema, explain what happens when this statement is executed.

Answer 3.8 The answers are given below:

1. Consider the following example. It is natural to require that the *did* field of Works should be a foreign key, and refer to Dept.

```
CREATE TABLE Works ( eid     INTEGER NOT NULL ,
                     did     INTEGER NOT NULL ,
```

```
                                pcttime INTEGER,
                                PRIMARY KEY (eid, did),
                                UNIQUE (eid),
                                FOREIGN KEY (did) REFERENCES Dept )
```

When a user attempts to delete a Dept tuple, There are four options:

- Also delete all Works tuples that refer to it.
- Disallow the deletion of the Dept tuple if some Works tuple refers to it.
- For every Works tuple that refers to it, set the *did* field to the *did* of some (existing) 'default' department.
- For every Works tuple that refers to it, set the *did* field to *null*.

2.

```
CREATE TABLE Emp (    eid         INTEGER,
                      ename       CHAR(10),
                      age         INTEGER,
                      salary      REAL,
                      PRIMARY KEY (eid) )
CREATE TABLE Works (  eid         INTEGER,
                      did         INTEGER,
                      pcttime     INTEGER,
                      PRIMARY KEY (eid, did),
                      FOREIGN KEY (did) REFERENCES Dept,
                      FOREIGN KEY (eid) REFERENCES Emp,
                      ON DELETE CASCADE)
CREATE TABLE Dept (   did         INTEGER,
                      budget      REAL,
                      managerid   INTEGER ,
                      PRIMARY KEY (did),
                      FOREIGN KEY (managerid) REFERENCES Emp,
                      ON DELETE SET NULL)
```

3.

```
CREATE TABLE Dept (  did         INTEGER,
                     budget      REAL,
                     managerid INTEGER  NOT NULL ,
                     PRIMARY KEY (did),
                     FOREIGN KEY (managerid) REFERENCES Emp)
```

4.

```
INSERT
INTO    Emp     (eid, ename, age, salary)
VALUES          (101, 'John Doe', 32, 15000)
```

5.

```
UPDATE Emp E
```

```
SET     E.salary = E.salary * 1.10
```

6.
```
DELETE
FROM    Dept D
        WHERE D.dname = 'Toy'
```

The did field in the Works relation is a foreign key and references the Dept relation. This is the referential integrity constraint chosen. By adding the action `ON DELETE CASCADE` to this, when a department record is deleted, the Works record associated with that Dept is also deleted.

The query works as follows: The Dept relation is searched for a record with name = 'Toy' and that record is deleted. The did field of that record is then used to look in the Works relation for records with a matching did value. All such records are then deleted from the Works relation.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.2 Students with $age < 18$ on Instance S

Exercise 3.9 Consider the SQL query whose answer is shown in Figure 3.2.

1. Modify this query so that only the *login* column is included in the answer.
2. If the clause `WHERE` $S.gpa >= 2$ is added to the original query, what is the set of tuples in the answer?

Answer 3.9 The answers are as follows:

1. Only *login* is included in the answer:
```
SELECT S.login
FROM   Students S
WHERE  S.age < 18
```
2. The answer tuple for Madayan is omitted then.

Exercise 3.10 Explain why the addition of `NOT NULL` constraints to the SQL definition of the Manages relation (in Section 3.5.3) does not enforce the constraint that each department must have a manager. What, if anything, is achieved by requiring that the *ssn* field of Manages be non-*null*?

Answer 3.10 The constraint that the *ssn* field of Manages should not be *null* implies that for every tuple present in Manages, there should be a Manager. This does not however ensure that each department has an entry (or a tuple corresponding to that dept) in the Manages relation.

Exercise 3.11 Suppose that we have a ternary relationship R between entity sets A, B, and C such that A has a key constraint and total participation and B has a key constraint; these are the only constraints. A has attributes $a1$ and $a2$, with $a1$ being the key; B and C are similar. R has no descriptive attributes. Write SQL statements that create tables corresponding to this information so as to capture as many of the constraints as possible. If you cannot capture some constraint, explain why.

Answer 3.11 The following SQL statements create the corresponding relations.

```
CREATE TABLE A ( a1      CHAR(10),
                 a2      CHAR(10),
                 b1      CHAR(10),
                 c1      CHAR(10),
                 PRIMARY KEY (a1),
                 UNIQUE (b1),
                 FOREIGN KEY (b1) REFERENCES B,
                 FOREIGN KEY (c1) REFERENCES C )

CREATE TABLE B ( b1      CHAR(10),
                 b2      CHAR(10),
                 PRIMARY KEY (b1) )

CREATE TABLE C ( b1      CHAR(10),
                 c2      CHAR(10),
                 PRIMARY KEY (c1) )
```

The first SQL statement folds the relationship R into table A and thereby guarantees the participation constraint.

Exercise 3.12 Consider the scenario from Exercise 2.2, where you designed an ER diagram for a university database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Answer 3.12 The following SQL statements create the corresponding relations.

1. `CREATE TABLE` Teaches (ssn `CHAR(10)`,
   courseId `INTEGER`,
   semester `CHAR(10)`,
   `PRIMARY KEY` (ssn, courseId, semester),
   `FOREIGN KEY` (ssn) `REFERENCES` Professor,
   `FOREIGN KEY` (courseId) `REFERENCES` Course)
   `FOREIGN KEY` (semester) `REFERENCES` Semester)

   The table corresponding to Course is created using:

   `CREATE TABLE` Course (courseId `INTEGER`,
   `PRIMARY KEY` (courseId))

   The tables for Professor and Semester can be created similarly.

2. `CREATE TABLE` Teaches (ssn `CHAR(10)`,
   courseId `INTEGER`,
   semester `CHAR(10)`,
   `PRIMARY KEY` (ssn, courseId),
   `FOREIGN KEY` (ssn) `REFERENCES` Professor,
   `FOREIGN KEY` (courseId) `REFERENCES` Course)

   Tables for Professor and Course can be created as before.

3. The tables created for the previous part to this question are the best we can do without using check constraints or assertions in SQL. The participation constraint cannot be captured using only primary and foreign key constraints because we cannot ensure that every entry in $Professor$ has an entry in $Teaches$.

4. `CREATE TABLE` Professor_teaches (ssn `CHAR(10)`,
   courseId `INTEGER`,
   semester `CHAR(10)`,
   `PRIMARY KEY` (ssn),
   `FOREIGN KEY` (courseId)
   `REFERENCES` Course)

   `CREATE TABLE` Course (courseId `INTEGER`,
   `PRIMARY KEY` (courseId))

   Observe that we do not need a separate table for Professor.

5. CREATE TABLE Professor_teaches (ssn CHAR(10),
   courseId INTEGER,
   semester CHAR(10),
   PRIMARY KEY (ssn),
   FOREIGN KEY (courseId)
   REFERENCES Course)

   Observe that the table for Professor can be omitted as before. Interestingly, we do not need a table for Course either, because (i) every course must be taught, and (ii) the only attribute for Course is courseId, which is included in the Professor_teaches table. If Course had other attributes, we could need a separate table for Course, and would not be able to enforce the constraint that every course should be taught by some professor (without including all attributes of Course in Professor_teaches and dropping the Course table; a solution that leads to *redundancy* if several professors teach the same course.)

6. CREATE TABLE Teaches (gid INTEGER,
   courseId INTEGER,
   semester CHAR(10),
   PRIMARY KEY (gid, courseId),
   FOREIGN KEY (gid) REFERENCES Group,
   FOREIGN KEY (courseId) REFERENCES Course)

   CREATE TABLE MemberOf (ssn CHAR(10),
   gid INTEGER,
   PRIMARY KEY (ssn, gid),
   FOREIGN KEY (ssn) REFERENCES Professor,
   FOREIGN KEY (gid) REFERENCES Group)

   CREATE TABLE Course (courseId INTEGER,
   PRIMARY KEY (courseId))

   CREATE TABLE Group (gid INTEGER,
   PRIMARY KEY (gid))

   CREATE TABLE Professor (ssn CHAR(10),
   PRIMARY KEY (ssn))

Exercise 3.13 Consider the university database from Exercise 2.3 and the ER diagram you designed. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Answer 3.13 The following SQL statements create the corresponding relations.

1. `CREATE TABLE` Professors (
   prof_ssn `CHAR(10)`,
   name `CHAR(64)`,
   age `INTEGER`,
   rank `INTEGER`,
   speciality `CHAR(64)`,
   `PRIMARY KEY` (prof_ssn))

2. `CREATE TABLE` Depts (
   dno `INTEGER`,
   dname `CHAR(64)`,
   office `CHAR(10)`,
   `PRIMARY KEY` (dno))

3. `CREATE TABLE` Runs (
   dno `INTEGER`,
   prof_ssn `CHAR(10)`,
   `PRIMARY KEY` (dno, prof_ssn),
   `FOREIGN KEY` (prof_ssn) `REFERENCES` Professors,
   `FOREIGN KEY` (dno) `REFERENCES` Depts)

4. `CREATE TABLE` Work_Dept (
   dno `INTEGER`,
   prof_ssn `CHAR(10)`,
   pc_time `INTEGER`,
   `PRIMARY KEY` (dno, prof_ssn),
   `FOREIGN KEY` (prof_ssn) `REFERENCES` Professors,
   `FOREIGN KEY` (dno) `REFERENCES` Depts)

   Observe that we would need check constraints or assertions in SQL to enforce the rule that Professors work in at least one department.

5. `CREATE TABLE` Project (
   pid `INTEGER`,
   sponsor `CHAR(32)`,
   start_date `DATE`,
   end_date `DATE`,
   budget `FLOAT`,
   `PRIMARY KEY` (pid))

6. CREATE TABLE Graduates (grad_ssn CHAR(10),
   age INTEGER,
   name CHAR(64),
   deg_prog CHAR(32),
   major INTEGER,
   PRIMARY KEY (grad_ssn),
   FOREIGN KEY (major) REFERENCES Depts)

   Note that the Major table is not necessary since each Graduate has only one major and so this can be an attribute in the Graduates table.

7. CREATE TABLE Advisor (senior_ssn CHAR(10),
   grad_ssn CHAR(10),
   PRIMARY KEY (senior_ssn, grad_ssn),
   FOREIGN KEY (senior_ssn)
   REFERENCES Graduates (grad_ssn),
   FOREIGN KEY (grad_ssn) REFERENCES Graduates)

8. CREATE TABLE Manages (pid INTEGER,
   prof_ssn CHAR(10),
   PRIMARY KEY (pid, prof_ssn),
   FOREIGN KEY (prof_ssn) REFERENCES Professors,
   FOREIGN KEY (pid) REFERENCES Projects)

9. CREATE TABLE Work_In (pid INTEGER,
   prof_ssn CHAR(10),
   PRIMARY KEY (pid, prof_ssn),
   FOREIGN KEY (prof_ssn) REFERENCES Professors,
   FOREIGN KEY (pid) REFERENCES Projects)

   Observe that we cannot enforce the participation constraint for Projects in the Work_In table without check constraints or assertions in SQL.

10. CREATE TABLE Supervises (prof_ssn CHAR(10),
    grad_ssn CHAR(10),
    pid INTEGER,
    PRIMARY KEY (prof_ssn, grad_ssn, pid),
    FOREIGN KEY (prof_ssn) REFERENCES Professors,
    FOREIGN KEY (grad_ssn) REFERENCES Graduates,
    FOREIGN KEY (pid) REFERENCES Projects)

    Note that we do not need an explicit table for the Work_Proj relation since every time a Graduate works on a Project, he or she must have a Supervisor.

Exercise 3.14 Consider the scenario from Exercise 2.4, where you designed an ER diagram for a company database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Answer 3.14 The following SQL statements create the corresponding relations.

```
CREATE TABLE Employees ( ssn      CHAR(10),
                         sal      INTEGER,
                         phone    CHAR(13),
                         PRIMARY KEY (ssn) )


CREATE TABLE Departments ( dno      INTEGER,
                           budget   INTEGER,
                           dname    CHAR(20),
                           PRIMARY KEY (dno) )


CREATE TABLE Works_in (  ssn      CHAR(10),
                         dno      INTEGER,
                         PRIMARY KEY (ssn, dno),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (dno) REFERENCES Departments)


CREATE TABLE Manages (   ssn      CHAR(10),
                         dno      INTEGER,
                         PRIMARY KEY (dno),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (dno) REFERENCES Departments)


CREATE TABLE Dependents (ssn      CHAR(10),
                         name     CHAR(10),
                         age      INTEGER,
                         PRIMARY KEY (ssn, name),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                                 ON DELETE CASCADE )
```

Exercise 3.15 Consider the Notown database from Exercise 2.5. You have decided to recommend that Notown use a relational database system to store company data. Show the SQL statements for creating relations corresponding to the entity sets and relationship sets in your design. Identify any constraints in the ER diagram that you are unable to capture in the SQL statements and briefly explain why you could not express them.

Answer 3.15 The following SQL statements create the corresponding relations.

1. `CREATE TABLE` Musicians (ssn `CHAR(10)`,
   name `CHAR(30)`,
   `PRIMARY KEY` (ssn))

2. `CREATE TABLE` Instruments (instrId `CHAR(10)`,
   dname `CHAR(30)`,
   key `CHAR(5)`,
   `PRIMARY KEY` (instrId))

3. `CREATE TABLE` Plays (ssn `CHAR(10)`,
   instrId `INTEGER`,
   `PRIMARY KEY` (ssn, instrId),
   `FOREIGN KEY` (ssn) `REFERENCES` Musicians,
   `FOREIGN KEY` (instrId) `REFERENCES` Instruments)

4. `CREATE TABLE` Songs_Appears (songId `INTEGER`,
   author `CHAR(30)`,
   title `CHAR(30)`,
   albumIdentifier `INTEGER NOT NULL`,
   `PRIMARY KEY` (songId),
   `FOREIGN KEY` (albumIdentifier)
   References Album_Producer,

5. `CREATE TABLE` Telephone_Home (phone `CHAR(11)`,
   address `CHAR(30)`,
   `PRIMARY KEY` (phone),
   `FOREIGN KEY` (address) `REFERENCES` Place,

6. `CREATE TABLE` Lives (ssn `CHAR(10)`,
   phone `CHAR(11)`,
   address `CHAR(30)`,

```
                                PRIMARY KEY (ssn, address),
                                FOREIGN KEY (phone, address)
                                        References Telephone_Home,
                                        FOREIGN KEY (ssn) REFERENCES Musicians )
```

7. `CREATE TABLE` Place (address `CHAR(30)`)

8. `CREATE TABLE` Perform (songId `INTEGER`,
   ssn `CHAR(10)`,
   `PRIMARY KEY` (ssn, songId),
   `FOREIGN KEY` (songId) `REFERENCES` Songs,
   `FOREIGN KEY` (ssn) `REFERENCES` Musicians)

9. `CREATE TABLE` Album_Producer (albumIdentifier `INTEGER`,
   ssn `CHAR(10)`,
   copyrightDate `DATE`,
   speed `INTEGER`,
   title `CHAR(30)`,
   `PRIMARY KEY` (albumIdentifier),
   `FOREIGN KEY` (ssn) `REFERENCES` Musicians)

Exercise 3.16 Translate your ER diagram from Exercise 2.6 into a relational schema, and show the SQL statements needed to create the relations, using only key and null constraints. If your translation cannot capture any constraints in the ER diagram, explain why.

In Exercise 2.6, you also modified the ER diagram to include the constraint that tests on a plane must be conducted by a technician who is an expert on that model. Can you modify the SQL statements defining the relations obtained by mapping the ER diagram to check this constraint?

Answer 3.16 The following SQL statements create the corresponding relations.

1. `CREATE TABLE` Expert (ssn `CHAR(11)`,
   model_no `INTEGER`,
   `PRIMARY KEY` (ssn, model_no),
   `FOREIGN KEY` (ssn) `REFERENCES` Technician,
   `FOREIGN KEY` (model_no) `REFERENCES` Models)

The participation constraint cannot be captured in the table.

2. CREATE TABLE Models (model_no INTEGER,
   capacity INTEGER,
   weight INTEGER,
   PRIMARY KEY (model_no))

3. CREATE TABLE Employees (ssn CHAR(11),
   union_mem_no INTEGER,
   PRIMARY KEY (ssn))

4. CREATE TABLE Technician_emp (ssn CHAR(11),
   name CHAR(20),
   address CHAR(20),
   phone_no CHAR(14),
   PRIMARY KEY (ssn),
   FOREIGN KEY (ssn)
   REFERENCES Employees
   ON DELETE CASCADE)

5. CREATE TABLE Traffic_control_emp (ssn CHAR(11),
   exam_date DATE,
   PRIMARY KEY (ssn),
   FOREIGN KEY (ssn)
   REFERENCES Employees
   ON DELETE CASCADE)

6. CREATE TABLE Plane_Type (reg_no INTEGER,
   model_no INTEGER,
   PRIMARY KEY (reg_no),
   FOREIGN KEY (model_no) REFERENCES Models)

7. CREATE TABLE Test_info (FFA_no INTEGER,
   ssn CHAR(11),
   reg_no INTEGER,
   hours INTEGER,
   date DATE,
   score INTEGER,
   PRIMARY KEY (ssn, reg_no, FFA_no),
   FOREIGN KEY (reg_no) REFERENCES Plane_Type,
   FOREIGN KEY (FAA_no) REFERENCES Test,
   FOREIGN KEY (ssn) REFERENCES Employees)

8. The constraint that tests on a plane must be conducted by a technician who is an expert on that model can be expressed in SQL as follows.

```
CREATE TABLE Test_info (    FFA_no  INTEGER,
                            ssn     CHAR(11),
                            reg_no  INTEGER,
                            hours   INTEGER,
                            date    DATE,
                            score   INTEGER,
                            PRIMARY KEY (ssn, reg_no, FFA_no),
                            FOREIGN KEY (reg_no) REFERENCES Plane_Type,
                            FOREIGN KEY (FAA_no) REFERENCES Test,
                            FOREIGN KEY (ssn) REFERENCES Technician_emp )
                            CONSTRAINT MODEL
                            CHECK ( SELECT * FROM Expert, Type
                                    WHERE Expert.ssn = ssn AND
                                    Expert.model_no = Type.model_no AND
                                    Type.reg_no = reg_no )
```

Exercise 3.17 Consider the ER diagram that you designed for the Prescriptions-R-X chain of pharmacies in Exercise 2.7. Define relations corresponding to the entity sets and relationship sets in your design using SQL.

Answer 3.17 The statements to create tables corresponding to entity sets Doctor, Pharmacy, and Pharm_co are straightforward and omitted. The other required tables can be created as follows:

1.
```
CREATE TABLE Pri_Phy_Patient ( ssn       CHAR(11),
                               name      CHAR(20),
                               age       INTEGER,
                               address   CHAR(20),
                               phy_ssn   CHAR(11),
                               PRIMARY KEY (ssn),
                               FOREIGN KEY (phy_ssn) REFERENCES Doctor )
```

2.
```
CREATE TABLE Prescription ( ssn         CHAR(11),
                            phy_ssn     CHAR(11),
                            date        CHAR(11),
                            quantity    INTEGER,
                            trade_name  CHAR(20),
                            pharm_id    CHAR(11),
```

```
                    PRIMARY KEY (ssn, phy_ssn),
                    FOREIGN KEY (ssn) REFERENCES Patient,
                    FOREIGN KEY (phy_ssn) REFERENCES Doctor,
                    FOREIGN KEY (trade_name, pharm_id)
                              References Make_Drug)
```

3. `CREATE TABLE` Make_Drug (trade_name `CHAR(20)`,
   pharm_id `CHAR(11)`,
   `PRIMARY KEY` (trade_name, pharm_id),
   `FOREIGN KEY` (trade_name) `REFERENCES` Drug,
   `FOREIGN KEY` (pharm_id) `REFERENCES` Pharm_co)

4. `CREATE TABLE` Sell (price `INTEGER`,
   name `CHAR(10)`,
   trade_name `CHAR(10)`,
   `PRIMARY KEY` (name, trade_name),
   `FOREIGN KEY` (name) `REFERENCES` Pharmacy,
   `FOREIGN KEY` (trade_name) `REFERENCES` Drug)

5. `CREATE TABLE` Contract (name `CHAR(20)`,
   pharm_id `CHAR(11)`,
   start_date `CHAR(11)`,
   end_date `CHAR(11)`,
   text `CHAR(10000)`,
   supervisor `CHAR(20)`,
   `PRIMARY KEY` (name, pharm_id),
   `FOREIGN KEY` (name) `REFERENCES` Pharmacy,
   `FOREIGN KEY` (pharm_id) `REFERENCES` Pharm_co)

Exercise 3.18 Write SQL statements to create the corresponding relations to the ER diagram you designed for Exercise 2.8. If your translation cannot capture any constraints in the ER diagram, explain why.

Answer 3.18 The statements to create tables corresponding to entity sets Customer, Group, and Artist are straightforward and omitted. The other required tables can be created as follows:

1. `CREATE TABLE` Classify (title `CHAR(20)`,
   name `CHAR(20)`,
   `PRIMARY KEY` (title, name),

```
                    FOREIGN KEY (title) REFERENCES Artwork_Paints,
                    FOREIGN KEY (name) REFERENCES Group )
```

2. `CREATE TABLE` Like_Group (name `CHAR(20)`,
   cust_name `CHAR(20)`,
   `PRIMARY KEY` (name, cust_name),
   `FOREIGN KEY` (name) `REFERENCES` Group,
   `FOREIGN KEY` (cust_name) `REFERENCES` Customer)

3. `CREATE TABLE` Like_Artist (name `CHAR(20)`,
   cust_name `CHAR(20)`,
   `PRIMARY KEY` (name, cust_name),
   `FOREIGN KEY` (name) `REFERENCES` Artist,
   `FOREIGN KEY` (cust_name) `REFERENCES` Customer)

4. `CREATE TABLE` Artwork_Paints (title `CHAR(20)`,
   artist_name `CHAR(20)`,
   type `CHAR(20)`,
   price `INTEGER`,
   year `INTEGER`,
   `PRIMARY KEY` (title),
   `FOREIGN KEY` (artist_name)
   References Artist)

Exercise 3.19 Briefly answer the following questions based on this schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *budget:* `real`, *managerid:* `integer`)

1. Suppose you have a view SeniorEmp defined as follows:

```
CREATE VIEW SeniorEmp (sname, sage, salary)
       AS SELECT E.ename, E.age, E.salary
          FROM   Emp E
          WHERE  E.age > 50
```

   Explain what the system will do to process the following query:

```
SELECT S.sname
FROM   SeniorEmp S
WHERE  S.salary > 100,000
```

2. Give an example of a view on Emp that could be automatically updated by updating Emp.

3. Give an example of a view on Emp that would be impossible to update (automatically) and explain why your example presents the update problem that it does.

Answer 3.19 The answer to each question is given below.

1. The system will do the following:

```
SELECT   S.name
FROM     ( SELECT E.ename AS name, E.age, E.salary
           FROM     Emp E
           WHERE    E.age > 50 ) AS S
WHERE    S.salary > 100000
```

2. The following view on Emp can be updated automatically by updating Emp:

```
CREATE VIEW SeniorEmp (eid, name, age, salary)
       AS SELECT E.eid, E.ename, E.age, E.salary
          FROM   Emp E
          WHERE  E.age > 50
```

3. The following view cannot be updated automatically because it is not clear which employee records will be affected by a given update:

```
CREATE VIEW AvgSalaryByAge (age, avgSalary)
       AS SELECT     E.eid, AVG (E.salary)
          FROM       Emp E
          GROUP BY   E.age
```

Exercise 3.20 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers. Answer the following questions:

- Give an example of an updatable view involving one relation.
- Give an example of an updatable view involving two relations.
- Give an example of an insertable-into view that is updatable.
- Give an example of an insertable-into view that is not updatable.

Answer 3.20 Not yet available.

4

RELATIONAL ALGEBRA AND CALCULUS

Exercise 4.1 Explain the statement that relational algebra operators can be *composed*. Why is the ability to compose operators important?

Answer 4.1 Every operator in relational algebra accepts one or more relation instances as arguments and the result is always an relation instance. So the argument of one operator could be the result of another operator. This is important because, this makes it easy to write complex queries by simply composing the relational algebra operators.

Exercise 4.2 Given two relations $R1$ and $R2$, where $R1$ contains N1 tuples, $R2$ contains N2 tuples, and N2 > N1 > 0, give the minimum and maximum possible sizes (in tuples) for the resulting relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for $R1$ and $R2$ needed to make the expression meaningful:

(1) $R1 \cup R2$, (2) $R1 \cap R2$, (3) $R1 - R2$, (4) $R1 \times R2$, (5) $\sigma_{a=5}(R1)$, (6) $\pi_a(R1)$, and (7) $R1/R2$

Answer 4.2 See Figure 4.1.

Exercise 4.3 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

Expression	Assumption	Min	Max
$R1 \cup R2$	$R1$ and $R2$ are union-compatible	N2	N1 + N2
$R1 \cap R2$	$R1$ and $R2$ are union-compatible	0	N1
$R1 - R2$	$R1$ and $R2$ are union-compatible	0	N1
$R1 \times R2$		N1 * N2	N1 * N2
$\sigma_{a=5}(R1)$	$R1$ has an attribute named a	0	N1
$\pi_a(R1)$	$R1$ has attribute a, N1>0	1	N1
$R1/R2$	The set of attributes of $R2$ is a subset of the set of attributes of $R1$	0	0
$R2/R1$	The set of attributes of $R1$ is a subset of the set of attributes of $R2$	0	$\lfloor$N2 / N1$\rfloor$

Figure 4.1 Answer to Exercise 4.2.

1. Find the *name*s of suppliers who supply some red part.
2. Find the *sid*s of suppliers who supply some red or green part.
3. Find the *sid*s of suppliers who supply some red part or are at 221 Packer Street.
4. Find the *sid*s of suppliers who supply some red part and some green part.
5. Find the *sid*s of suppliers who supply every part.
6. Find the *sid*s of suppliers who supply every red part.
7. Find the *sid*s of suppliers who supply every red or green part.
8. Find the *sid*s of suppliers who supply every red part or supply every green part.
9. Find pairs of *sid*s such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
10. Find the *pid*s of parts supplied by at least two different suppliers.
11. Find the *pid*s of the most expensive parts supplied by suppliers named Yosemite Sham.
12. Find the *pid*s of parts supplied by every supplier at less than $200. (If any supplier either does not supply the part or charges more than $200 for it, the part is not selected.)

Answer 4.3 In the answers below RA refers to Relational Algebra, TRC refers to Tuple Relational Calculus and DRC refers to Domain Relational Calculus.

1. ■ RA

$$\pi_{sname}(\pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog) \bowtie Suppliers)$$

■ TRC

$$\{T \mid \exists T1 \in Suppliers(\exists X \in Parts(X.color ='red' \wedge \exists Y \in Catalog (Y.pid = X.pid \wedge Y.sid = T1.sid)) \wedge T.sname = T1.sname)\}$$

■ DRC

$$\{\langle Y \rangle \mid \langle X, Y, Z \rangle \in Suppliers \wedge \exists P, Q, R(\langle P, Q, R \rangle \in Parts \wedge R ='red' \wedge \exists I, J, K(\langle I, J, K \rangle \in Catalog \wedge J = P \wedge I = X))\}$$

■ SQL

```
SELECT S.sname
FROM   Suppliers S, Parts P, Catalog C
WHERE  P.color='red' AND C.pid=P.pid AND C.sid=S.sid
```

2. ■ RA

$$\pi_{sid}(\pi_{pid}(\sigma_{color='red' \vee color='green'}Parts) \bowtie catalog)$$

■ TRC

$$\{T \mid \exists T1 \in Catalog(\exists X \in Parts((X.color = `red' \vee X.color = `green') \wedge X.pid = T1.pid) \wedge T.sid = T1.sid)\}$$

■ DRC

$$\{\langle X \rangle \mid \langle X, Y, Z \rangle \in Catalog \wedge \exists A, B, C(\langle A, B, C \rangle \in Parts \wedge (C ='red' \vee C ='green') \wedge A = Y)\}$$

■ SQL

```
SELECT C.sid
FROM   Catalog C, Parts P
WHERE  (P.color = 'red' OR P.color = 'green')
       AND P.pid = C.pid
```

3. ■ RA

$$\rho(R1, \pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog))$$
$$\rho(R2, \pi_{sid}\sigma_{address='221PackerStreet'}Suppliers)$$
$$R1 \cup R2$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\exists X \in Parts(X.color = `red' \wedge X.pid = T1.pid) \wedge T.sid = T1.sid) \vee \exists T2 \in Suppliers(T2.address =' 221 PackerStreet' \wedge T.sid = T2.sid)\}$$

- DRC

$$\{\langle X\rangle \mid \langle X,Y,Z\rangle \in Catalog \wedge \exists A,B,C(\langle A,B,C\rangle \in Parts \wedge C =' red' \wedge A = Y) \vee \exists P,Q(\langle X,P,Q\rangle \in Suppliers \wedge Q =' 221 PackerStreet')\}$$

- SQL

```
SELECT S.sid
FROM   Suppliers S
WHERE  S.address = '221 Packer street'
       OR S.sid IN ( SELECT C.sid
                     FROM   Parts P, Catalog C
                     WHERE  P.color='red' AND P.pid = C.pid )
```

4. - RA

$$\rho(R1, \pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog))$$
$$\rho(R2, \pi_{sid}((\pi_{pid}\sigma_{color='green'}Parts) \bowtie Catalog))$$
$$R1 \cap R2$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\exists X \in Parts(X.color = `red' \wedge X.pid = T1.pid) \wedge \exists T2 \in Catalog(\exists Y \in Parts(Y.color =' green' \wedge Y.pid = T2.pid) \wedge T2.sid = T1.sid) \wedge T.sid = T1.sid)\}$$

- DRC

$$\{\langle X\rangle \mid \langle X,Y,Z\rangle \in Catalog \wedge \exists A,B,C(\langle A,B,C\rangle \in Parts \wedge C =' red' \wedge A = Y) \wedge \exists P,Q,R(\langle P,Q,R\rangle \in Catalog \wedge \exists E,F,G(\langle E,F,G\rangle \in Parts \wedge G =' green' \wedge E = Q) \wedge P = X)\}$$

- SQL

```
SELECT C.sid
FROM   Parts P, Catalog C
WHERE  P.color = 'red' AND P.pid = C.pid
       AND EXISTS ( SELECT P2.pid
                    FROM   Parts P2, Catalog C2
                    WHERE  P2.color = 'green' AND C2.sid = C.sid
                           AND P2.pid = C2.pid )
```

5. ■ RA

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}Parts)$$

■ TRC

$$\{T \mid \exists T1 \in Catalog(\forall X \in Parts(\exists T2 \in Catalog (T2.pid = X.pid \wedge T2.sid = T1.sid)) \wedge T.sid = T1.sid)\}$$

■ DRC

$$\{\langle X\rangle \mid \langle X,Y,Z\rangle \in Catalog \wedge \forall\langle A,B,C\rangle \in Parts (\exists\langle P,Q,R\rangle \in Catalog(Q = A \wedge P = X))\}$$

■ SQL

```
SELECT C.sid
FROM   Catalog C
WHERE  NOT EXISTS (SELECT P.pid
                   FROM   Parts P
                   WHERE  NOT EXISTS (SELECT C1.sid
                                      FROM   Catalog C1
                                      WHERE  C1.sid = C.sid
                                             AND C1.pid = P.pid))
```

6. ■ RA

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red'}Parts)$$

■ TRC

$$\{T \mid \exists T1 \in Catalog(\forall X \in Parts(X.color \neq 'red' \vee \exists T2 \in Catalog(T2.pid = X.pid \wedge T2.sid = T1.sid)) \wedge T.sid = T1.sid)\}$$

■ DRC

$$\{\langle X\rangle \mid \langle X,Y,Z\rangle \in Catalog \wedge \forall\langle A,B,C\rangle \in Parts (C \neq 'red' \vee \exists\langle P,Q,R\rangle \in Catalog(Q = A \wedge P = X))\}$$

- SQL

```
SELECT C.sid
FROM   Catalog C
WHERE  NOT EXISTS (SELECT P.pid
                   FROM   Parts P
                   WHERE  P.color = 'red'
                   AND (NOT EXISTS (SELECT C1.sid
                                    FROM   Catalog C1
                                    WHERE  C1.sid = C.sid AND
                                           C1.pid = P.pid)))
```

7. - RA

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red' \vee color='green'}Parts)$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\forall X \in Parts((X.color \neq \text{`}red' \wedge X.color \neq \text{`}green') \vee \exists T2 \in Catalog (T2.pid = X.pid \wedge T2.sid = T1.sid)) \wedge T.sid = T1.sid)\}$$

- DRC

$$\{\langle X\rangle \mid \langle X,Y,Z\rangle \in Catalog \wedge \forall\langle A,B,C\rangle \in Parts ((C \neq \text{`}red' \wedge C \neq \text{`}green') \vee \exists\langle P,Q,R\rangle \in Catalog (Q = A \wedge P = X))\}$$

- SQL

```
SELECT C.sid
FROM   Catalog C
WHERE  NOT EXISTS (SELECT P.pid
                   FROM   Parts P
                   WHERE  (P.color = 'red' OR P.color = 'green')
                   AND (NOT EXISTS (SELECT C1.sid
                                    FROM   Catalog C1
                                    WHERE  C1.sid = C.sid AND
                                           C1.pid = P.pid)))
```

8. - RA

$$\rho(R1, ((\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red'}Parts)))$$
$$\rho(R2, ((\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='green'}Parts)))$$
$$R1 \cup R2$$

- TRC

$$\{T \mid \exists T1 \in Catalog((\forall X \in Parts (X.color \neq \text{`}red' \vee \exists Y \in Catalog(Y.pid = X.pid \wedge Y.sid = T1.sid)) \vee \forall Z \in Parts(Z.color \neq \text{`}green' \vee \exists P \in Catalog (P.pid = Z.pid \wedge P.sid = T1.sid))) \wedge T.sid = T1.sid)\}$$

- DRC

$$\{\langle X \rangle \mid \langle X, Y, Z \rangle \in Catalog \wedge (\forall \langle A, B, C \rangle \in Parts (C \neq \text{`}red' \vee \exists \langle P, Q, R \rangle \in Catalog(Q = A \wedge P = X)) \vee \forall \langle U, V, W \rangle \in Parts(W \neq \text{`}green' \vee \langle M, N, L \rangle \in Catalog (N = U \wedge M = X)))\}$$

- SQL

```
SELECT C.sid
FROM   Catalog C
WHERE  (NOT EXISTS (SELECT P.pid
                    FROM   Parts P
                    WHERE  P.color = 'red' AND
                    (NOT EXISTS (SELECT C1.sid
                                 FROM   Catalog C1
                                 WHERE  C1.sid = C.sid AND
                                        C1.pid = P.pid))))
       OR ( NOT EXISTS (SELECT P1.pid
                        FROM   Parts P1
                        WHERE  P1.color = 'green' AND
                        (NOT EXISTS (SELECT C2.sid
                                     FROM   Catalog C2
                                     WHERE  C2.sid = C.sid AND
                                            C2.pid = P1.pid))))
```

9. - RA

$$\rho(R1, Catalog)$$
$$\rho(R2, Catalog)$$
$$\pi_{R1.sid, R2.sid}(\sigma_{R1.pid=R2.pid \wedge R1.sid \neq R2.sid \wedge R1.cost > R2.cost}(R1 \times R2))$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\exists T2 \in Catalog (T2.pid = T1.pid \wedge T2.sid \neq T1.sid \wedge T2.cost < T1.cost \wedge T.sid2 = T2.sid) \wedge T.sid1 = T1.sid)\}$$

- DRC

$$\{\langle X, P\rangle \mid \langle X, Y, Z\rangle \in Catalog \wedge \exists P, Q, R \\ (\langle P, Q, R\rangle \in Catalog \wedge Q = Y \wedge P \neq X \wedge R < Z)\}$$

- SQL

```
SELECT C1.sid, C2.sid
FROM   Catalog C1, Catalog C2
WHERE  C1.pid = C2.pid AND C1.sid ≠ C2.sid
       AND C1.cost > C2.cost
```

10. - RA

$$\rho(R1, Catalog) \\ \rho(R2, Catalog) \\ \pi_{R1.pid}\sigma_{R1.pid=R2.pid \wedge R1.sid \neq R2.sid}(R1 \times R2)$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\exists T2 \in Catalog \\ (T2.pid = T1.pid \wedge T2.sid \neq T1.sid) \\ \wedge T.pid = T1.pid)\}$$

- DRC

$$\{\langle X\rangle \mid \langle X, Y, Z\rangle \in Catalog \wedge \exists A, B, C \\ (\langle A, B, C\rangle \in Catalog \wedge B = Y \wedge A \neq X)\}$$

- SQL

```
SELECT C.pid
FROM   Catalog C
WHERE  EXISTS (SELECT C1.sid
               FROM   Catalog C1
               WHERE  C1.pid = C.pid AND C1.sid ≠ C.sid )
```

11. - RA

$$\rho(R1, \pi_{sid}\sigma_{sname='YosemiteSham'} Suppliers) \\ \rho(R2, R1 \bowtie Catalog) \\ \rho(R3, R2) \\ \rho(R4(1 \to sid, 2 \to pid, 3 \to cost), \sigma_{R3.cost<R2.cost}(R3 \times R2)) \\ \pi_{pid}(R2 - \pi_{sid,pid,cost}R4)$$

- TRC

$$\{T \mid \exists T1 \in Catalog(\exists X \in Suppliers$$
$$(X.sname =' YosemiteSham' \wedge X.sid = T1.sid) \wedge \neg(\exists S \in Suppliers$$
$$(S.sname =' YosemiteSham' \wedge \exists Z \in Catalog$$
$$(Z.sid = S.sid \wedge Z.cost > T1.cost))) \wedge T.pid = T1.pid)$$

- DRC

$$\{\langle Y\rangle \mid \langle X, Y, Z\rangle \in Catalog \wedge \exists A, B, C$$
$$(\langle A, B, C\rangle \in Suppliers \wedge C =' YosemiteSham' \wedge A = X)$$
$$\wedge\neg(\exists P, Q, R(\langle P, Q, R\rangle \in Suppliers \wedge R =' YosemiteSham'$$
$$\wedge\exists I, J, K(\langle I, J, K\rangle \in Catalog(I = P \wedge K > Z))))\}$$

- SQL

```
SELECT C.pid
FROM   Catalog C, Suppliers S
WHERE  S.sname = 'Yosemite Sham' AND C.sid = S.sid
       AND C.cost ≥ ALL (Select C2.cost
                         FROM  Catalog C2, Suppliers S2
                         WHERE S2.sname = 'Yosemite Sham'
                               AND C2.sid = S2.sid)
```

Exercise 4.4 Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

1. $\pi_{sname}(\pi_{sid}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog)) \bowtie Suppliers)$
2. $\pi_{sname}(\pi_{sid}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$
3. $(\pi_{sname}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$(\pi_{sname}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$
4. $(\pi_{sid}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$(\pi_{sid}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$
5. $\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$(\pi_{sid,sname}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)))$

Answer 4.4 The statements can be interpreted as:

1. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars.
2. This Relational Algebra statement does not return anything because of the sequence of projection operators. Once the sid is projected, it is the only field in the set. Therefore, projecting on sname will not return anything.
3. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.
4. Find the Supplier ids of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.
5. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

Exercise 4.5 Consider the following relations containing airline flight information:

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`,
distance: `integer`, *departs:* `time`, *arrives:* `time`)
Aircraft(*aid:* `integer`, *aname:* `string`, *cruisingrange:* `integer`)
Certified(*eid:* `integer`, *aid:* `integer`)
Employees(*eid:* `integer`, *ename:* `string`, *salary:* `integer`)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.

Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed. (See the exercises at the end of Chapter 5 for additional queries over the airline schema.)

1. Find the *eid*s of pilots certified for some Boeing aircraft.
2. Find the *name*s of pilots certified for some Boeing aircraft.
3. Find the *aid*s of all aircraft that can be used on non-stop flights from Bonn to Madras.
4. Identify the flights that can be piloted by every pilot whose salary is more than $100,000.
5. Find the names of pilots who can operate planes with a range greater than 3,000 miles but are not certified on any Boeing aircraft.

6. Find the *eid*s of employees who make the highest salary.
7. Find the *eid*s of employees who make the second highest salary.
8. Find the *eid*s of employees who are certified for the largest number of aircraft.
9. Find the *eid*s of employees who are certified for exactly three aircraft.
10. Find the total amount paid to employees as salaries.
11. Is there a sequence of flights from Madison to Timbuktu? Each flight in the sequence is required to depart from the city that is the destination of the previous flight; the first flight must leave Madison, the last flight must reach Timbuktu, and there is no restriction on the number of intermediate flights. Your query must determine whether a sequence of flights from Madison to Timbuktu exists for *any* input Flights relation instance.

Answer 4.5 In the answers below RA refers to Relational Algebra, TRC refers to Tuple Relational Calculus and DRC refers to Domain Relational Calculus.

1. ■ RA

$$\pi_{eid}(\sigma_{aname='Boeing'}(Aircraft \bowtie Certified))$$

■ TRC

$$\{C.eid \mid C \in Certified \wedge \exists A \in Aircraft(A.aid = C.aid \wedge A.aname = 'Boeing')\}$$

■ DRC

$$\{\langle Ceid \rangle \mid \langle Ceid, Caid \rangle \in Certified \wedge \exists Aid, AN, AR(\langle Aid, AN, AR \rangle \in Aircraft \wedge Aid = Caid \wedge AN = 'Boeing')\}$$

■ SQL

```
SELECT C.eid
FROM   Aircraft A, Certified C
WHERE  A.aid = C.aid AND A.aname = 'Boeing'
```

2. ■ RA

$$\pi_{ename}(\sigma_{aname='Boeing'}(Aircraft \bowtie Certified \bowtie Employees))$$

- TRC
  $\{E.ename \mid E \in Employees \wedge \exists C \in Certified$
  $(\exists A \in Aircraft(A.aid = C.aid \wedge A.aname = `Boeing' \wedge E.eid = C.eid))\}$

- DRC
  $\{\langle EN \rangle \mid \langle Eid, EN, ES \rangle \in Employees \wedge$
  $\exists Ceid, Caid(\langle Ceid, Caid \rangle \in Certified \wedge$
  $\exists Aid, AN, AR(\langle Aid, AN, AR \rangle \in Aircraft \wedge$
  $Aid = Caid \wedge AN = `Boeing' \wedge Eid = Ceid)\}$

- SQL

  ```
  SELECT E.ename
  FROM   Aircraft A, Certified C, Employees E
  WHERE  A.aid = C.aid AND A.aname = 'Boeing' AND E.eid = C.eid
  ```

3. - RA
     $\rho(BonnToMadrid, \sigma_{from=`Bonn' \wedge to=`Madrid'}(Flights))$
     $\pi_{aid}(\sigma_{cruisingrange > distance}(Aircraft \times BonnToMadrid))$

   - TRC
     $\{A.aid \mid A \in Aircraft \wedge \exists F \in Flights$
     $(F.from = `Bonn' \wedge F.to = `Madrid' \wedge A.cruisingrange > F.distance)\}$

   - DRC
     $\{Aid \mid \langle Aid, AN, AR \rangle \in Aircraft \wedge$
     $(\exists FN, FF, FT, FDi, FDe, FA(\langle FN, FF, FT, FDi, FDe, FA \rangle \in Flights \wedge$
     $FF = `Bonn' \wedge FT = `Madrid' \wedge FDi < AR))\}$

   - SQL

     ```
     SELECT A.aid
     FROM   Aircraft A, Flights F
     WHERE  F.from = 'Bonn' AND F.to = 'Madrid' AND
            A.cruisingrange > F.distance
     ```

4. - RA
     $\pi_{flno}(\sigma_{distance < cruisingrange \wedge salary > 100{,}000}(Flights \bowtie Aircraft \bowtie$
     $Certified \bowtie Employees)))$

- TRC $\{F.flno \mid F \in Flights \wedge \exists A \in Aircraft \exists C \in Certified$
  $\exists E \in Employees(A.cruisingrange > F.distance \wedge E.salary > 100,000 \wedge$
  $A.aid = C.aid \wedge E.eid = C.eid)\}$

- DRC
  $\{FN \mid \langle FN, FF, FT, FDi, FDe, FA\rangle \in Flights \wedge$
  $\exists Ceid, Caid(\langle Ceid, Caid\rangle \in Certified \wedge$
  $\exists Aid, AN, AR(\langle Aid, AN, AR\rangle \in Aircraft \wedge$
  $\exists Eid, EN, ES(\langle Eid, EN, ES\rangle \in Employees$
  $(AR > FDi \wedge ES > 100,000 \wedge Aid = Caid \wedge Eid = Ceid)\}$

- SQL

```
SELECT E.ename
FROM   Aircraft A, Certified C, Employees E, Flights F
WHERE  A.aid = C.aid AND E.eid = C.eid AND
       distance < cruisingrange AND salary > 100,000
```

5. - RA $\rho(R1, \pi_{eid}(\sigma_{cruisingrange>3000}(Aircraft \bowtie Certified)))$
  $\pi_{ename}(Employees \bowtie (R1 - \pi_{eid}(\sigma_{aname='Boeing'}(Aircraft \bowtie Certified))))$

- TRC
  $\{E.ename \mid E \in Employees \wedge \exists C \in Certified(\exists A \in Aircraft$
  $(A.aid = C.aid \wedge E.eid = C.eid \wedge A.cruisingrange > 3000)) \wedge$
  $\neg(\exists C2 \in Certified(\exists A2 \in Aircraft(A2.aname = `Boeing' \wedge C2.aid =$
  $A2.aid \wedge C2.eid = E.eid)))\}$

- DRC
  $\{\langle EN\rangle \mid \langle Eid, EN, ES\rangle \in Employees \wedge$
  $\exists Ceid, Caid(\langle Ceid, Caid\rangle \in Certified \wedge$
  $\exists Aid, AN, AR(\langle Aid, AN, AR\rangle \in Aircraft \wedge$
  $Aid = Caid \wedge Eid = Ceid \wedge AR > 3000)) \wedge$
  $\neg(\exists Aid2, AN2, AR2(\langle Aid2, AN2, AR2\rangle \in Aircraft \wedge$
  $\exists Ceid2, Caid2(\langle Ceid2, Caid2\rangle \in Certified$
  $\wedge Aid2 = Caid2 \wedge Eid = Ceid2 \wedge AN2 = `Boeing')))\}$

- SQL

```
SELECT E.ename
FROM   Certified C, Employees E, Aircraft A
WHERE  A.aid = C.aid AND E.eid = C.eid AND A.cruisingrange > 3000
AND E.eid NOT IN ( SELECT C2.eid
FROM Certified C2, Aircraft A2
WHERE C2.aid = A2.aid AND A2.aname = 'Boeing' )
```

6. ■ RA

The approach to take is first find all the employees who do not have the highest salary. Subtract these from the original list of employees and what is left is the highest paid employees.

$\rho(E1, Employees)$
$\rho(E2, Employees)$
$\rho(E3, \pi_{E2.eid}(E1 \bowtie_{E1.salary > E2.salary} E2)$
$(\pi_{eid} E1) - E3$

■ TRC

$\{E1.eid \mid E1 \in Employees \wedge \neg(\exists E2 \in Employees(E2.salary > E1.salary))\}$

■ DRC

$\{\langle Eid1 \rangle \mid \langle Eid1, EN1, ES1 \rangle \in Employees \wedge$
$\neg(\exists Eid2, EN2, ES2(\langle Eid2, EN2, ES2 \rangle \in Employees \wedge ES2 > ES1))\}$

■ SQL

```
SELECT E.eid
FROM   Employees E
WHERE  E.salary = ( Select MAX (E2.salary)
                    FROM  Employees E2 )
```

7. ■ RA

The approach taken is similar to the solution for the previous exercise. First find all the employees who do not have the highest salary. Remove these from the original list of employees and what is left is the highest paid employees. Remove the highest paid employees from the original list. What is left is the second highest paid employees together with the rest of the employees. Then find the highest paid employees of this new list. This is the list of the second highest paid employees.

$\rho(E1, Employees)$
$\rho(E2, Employees)$
$\rho(E3, \pi_{E2.eid}(E1 \bowtie_{E1.salary > E2.salary} E2)$
$\rho(E4, E2 \bowtie E3)$
$\rho(E5, E2 \bowtie E3)$
$\rho(E6, \pi_{E5.eid}(E4 \bowtie_{E1.salary > E5.salary} E5)$
$(\pi_{eid} E3) - E6$

- TRC

  $\{E1.eid \mid E1 \in Employees \wedge \exists E2 \in Employees(E2.salary > E1.salary \wedge \neg(\exists E3 \in Employees(E3.salary > E2.salary)))\}$

- DRC

  $\{\langle Eid1\rangle \mid \langle Eid1, EN1, ES1\rangle \in Employees \wedge \exists Eid2, EN2, ES2(\langle Eid2, EN2, ES2\rangle \in Employees(ES2 > ES1) \wedge \neg(\exists Eid3, EN3, ES3(\langle Eid3, EN3, ES3\rangle \in Employees(ES3 > ES2))))\}$

- SQL

```
SELECT E.eid
FROM   Employees E
WHERE  E.salary = (SELECT MAX (E2.salary)
                   FROM   Employees E2
                   WHERE  E2.salary ≠ (SELECT MAX (E3.salary)
                                       FROM   Employees E3 ))
```

8. This cannot be expressed in relational algebra (or calculus) because there is no operator to count, and this query requires the ability to count up to a number that depends on the data. The query can however be expressed in SQL as follows:

```
SELECT Temp.eid
FROM   ( SELECT   C.eid AS eid, COUNT (C.aid) AS cnt,
         FROM     Certified C
         GROUP BY C.eid) AS Temp
WHERE  Temp.cnt = ( SELECT   MAX (Temp.cnt)
                    FROM     Temp)
```

9. - RA

   The approach behind this query is to first find the employees who are certified for at least three aircraft (they appear at least three times in the Certified relation). Then find the employees who are certified for at least four aircraft. Subtract the second from the first and what is left is the employees who are certified for exactly three aircraft.

   $\rho(R1, Certified)$
   $\rho(R2, Certified)$
   $\rho(R3, Certified)$
   $\rho(R4, Certified)$
   $\rho(R5, \pi_{eid}(\sigma_{(R1.eid=R2.eid=R3.eid)\wedge(R1.aid\neq R2.aid\neq R3.aid)}(R1 \times R2 \times R3)))$
   $\rho(R6, \pi_{eid}(\sigma_{(R1.eid=R2.eid=R3.eid=R4.eid)\wedge(R1.aid\neq R2.aid\neq R3.aid\neq R4.aid)}$

$(R1 \times R2 \times R3 \times R4)))$
$R5 - R6$

- TRC
  $\{C1.eid \mid C1 \in Certified \wedge \exists C2 \in Certified(\exists C3 \in Certified$
  $(C1.eid = C2.eid \wedge C2.eid = C3.eid \wedge$
  $C1.aid \neq C2.aid \wedge C2.aid \neq C3.aid \wedge C3.aid \neq C1.aid \wedge$
  $\neg(\exists C4 \in Certified$
  $(C3.eid = C4.eid \wedge C1.aid \neq C4.aid \wedge$
  $C2.aid \neq C4.aid \wedge C3.aid \neq C4.aid))))\}$

- DRC
  $\{\langle CE1\rangle \mid \langle CE1, CA1\rangle \in Certified \wedge$
  $\exists CE2, CA2(\langle CE2, CA2\rangle \in Certified \wedge$
  $\exists CE3, CA3(\langle CE3, CA3\rangle \in Certified \wedge$
  $(CE1 = CE2 \wedge CE2 = CE3 \wedge$
  $CA1 \neq CA2 \wedge CA2 \neq CA3 \wedge CA3 \neq CA1 \wedge$
  $\neg(\exists CE4, CA4(\langle CE4, CA4\rangle \in Certified \wedge$
  $(CE3 = CE4 \wedge CA1 \neq CA4 \wedge$
  $CA2 \neq CA4 \wedge CA3 \neq CA4))))\}$

- SQL

```
SELECT C1.eid
FROM   Certified C1, Certified C2, Certified C3
WHERE  (C1.eid = C2.eid AND C2.eid = C3.eid AND
        C1.aid ≠ C2.aid AND C2.aid ≠ C3.aid AND C3.aid ≠ C1.aid)
EXCEPT
SELECT C4.eid
FROM   Certified C4, Certified C5, Certified C6, Certified C7,
WHERE  (C4.eid = C5.eid AND C5.eid = C6.eid AND C6.eid = C7.eid AND
        C4.aid ≠ C5.aid AND C4.aid ≠ C6.aid AND C4.aid ≠ C7.aid AND
        C5.aid ≠ C6.aid AND C5.aid ≠ C7.aid AND C6.aid ≠ C7.aid )
```

  This could also be done in SQL using COUNT.

10. This cannot be expressed in relational algebra (or calculus) because there is no operator to sum values. The query can however be expressed in SQL as follows:

```
SELECT SUM (E.salaries)
FROM   Employees E
```

11. This cannot be expressed in relational algebra or relational calculus or SQL. The problem is that there is no restriction on the number of intermediate flights. All of the query methods could find if there was a flight directly from Madison to Timbuktu and if there was a sequence of two flights that started in Madison and ended in Timbuktu. They could even find a sequence of n flights that started in Madison and ended in Timbuktu as long as there is a static (i.e., data-independent) upper bound on the number of intermediate flights. (For large n, this would of course be long and impractical, but at least possible.) In this query, however, the upper bound is not static but dynamic (based upon the set of tuples in the Flights relation).

    In summary, if we had a static upper bound (say k), we could write an algebra or SQL query that repeatedly computes (upto k) joins on the Flights relation. If the upper bound is dynamic, then we cannot write such a query because k is not known when writing the query.

Exercise 4.6 What is *relational completeness*? If a query language is relationally complete, can you write any desired query in that language?

Answer 4.6 *Relational completeness* means that a query language can express all the queries that can be expressed in relational algebra. It does not mean that the language can express any desired query.

Exercise 4.7 What is an *unsafe* query? Give an example and explain why it is important to disallow such queries.

Answer 4.7 An $unsafe$ query is a query in relational calculus that has an infinite number of results. An example of such a query is:

$$\{S \mid \neg(S \in Sailors)\}$$

The query is for all things that are not sailors which of course is everything else. Clearly there is an infinite number of answers, and this query is $unsafe$. It is important to disallow $unsafe$ queries because we want to be able to get back to users with a list of all the answers to a query after a finite amount of time.

5

SQL: QUERIES, CONSTRAINTS, TRIGGERS

Online material is available for all exercises in this chapter on the book's webpage at

`http://www.cs.wisc.edu/~dbbook`

This includes scripts to create tables for each exercise for use with Oracle, IBM DB2, Microsoft SQL Server, Microsoft Access and MySQL.

Exercise 5.1 Consider the following relations:

Student(*snum:* `integer`, *sname:* `string`, *major:* `string`, *level:* `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `string`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid:* `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

Write the following queries in SQL. No duplicates should be printed in any of the answers.

1. Find the names of all Juniors (level = JR) who are enrolled in a class taught by I. Teach.
2. Find the age of the oldest student who is either a History major or enrolled in a course taught by I. Teach.
3. Find the names of all classes that either meet in room R128 or have five or more students enrolled.
4. Find the names of all students who are enrolled in two classes that meet at the same time.

5. Find the names of faculty members who teach in every room in which some class is taught.
6. Find the names of faculty members for whom the combined enrollment of the courses that they teach is less than five.
7. For each level, print the level and the average age of students for that level.
8. For all levels except JR, print the level and the average age of students for that level.
9. For each faculty member that has taught classes only in room R128, print the faculty member's name and the total number of classes she or he has taught.
10. Find the names of students enrolled in the maximum number of classes.
11. Find the names of students not enrolled in any class.
12. For each age value that appears in Students, find the level value that appears most often. For example, if there are more FR level students aged 18 than SR, JR, or SO students aged 18, you should print the pair (18, FR).

Answer 5.1 The answers are given below:

1.
```
SELECT DISTINCT S.Sname
FROM   Student S, Class C, Enrolled E, Faculty F
WHERE  S.snum = E.snum AND E.cname = C.name AND C.fid = F.fid AND
       F.fname = 'I.Teach' AND S.level = 'JR'
```

2.
```
SELECT MAX(S.age)
FROM   Student S
WHERE  (S.major = 'History')
       OR S.snum IN (SELECT E.snum
                     FROM   Class C, Enrolled E, Faculty F
                     WHERE  E.cname = C.name AND C.fid = F.fid
                            AND F.fname = 'I.Teach' )
```

3.
```
SELECT   C.name
FROM     Class C
WHERE    C.room = 'R128'
         OR C.name IN (SELECT   E.cname
                       FROM     Enrolled E
                       GROUP BY E.cname
                       HAVING   COUNT (*) >= 5)
```

4.
```
SELECT DISTINCT S.sname
FROM    Student S
WHERE   S.snum IN (SELECT E1.snum
                   FROM    Enrolled E1, Enrolled E2, Class C1, Class C2
                   WHERE   E1.snum = E2.snum AND E1.cname <> E2.cname
                   AND E1.cname = C1.name
                   AND E2.cname = C2.name AND C1.meets_at = C2.meets_at)
```

5.
```
SELECT DISTINCT F.fname
FROM    Faculty F
WHERE   NOT EXISTS (( SELECT *
                      FROM    Class C )
                      EXCEPT
                      (SELECT C1.room
                      FROM    Class C1
                      WHERE   C1.fid = F.fid ))
```

6.
```
SELECT    DISTINCT F.fname
FROM      Faculty F
WHERE     5 > (SELECT COUNT (E.snum)
               FROM    Class C, Enrolled E
               WHERE   C.name = E.cname
               AND     C.fid = F.fid)
```

7.
```
SELECT    S.level, AVG(S.age)
FROM      Student S
GROUP BY S.level
```

8.
```
SELECT    S.level, AVG(S.age)
FROM      Student S
WHERE     S.level <> 'JR'
GROUP BY S.level
```

9.
```
SELECT    F.fname, COUNT(*) AS CourseCount
FROM      Faculty F, Class C
WHERE     F.fid = C.fid
GROUP BY F.fid, F.fname
HAVING    EVERY ( C.room = 'R128' )
```

10.
```
SELECT    DISTINCT S.sname
FROM      Student S
WHERE     S.snum IN (SELECT    E.snum
                     FROM      Enrolled E
                     GROUP BY E.snum
```

```
                    HAVING   COUNT (*) >= ALL (SELECT   COUNT (*)
                                               FROM     Enrolled E2
                                               GROUP BY E2.snum ))
```

11.
```
SELECT DISTINCT S.sname
FROM    Student S
WHERE   S.snum NOT IN (SELECT E.snum
                       FROM   Enrolled E )
```

12.
```
SELECT    S.age, S.level
FROM      Student S
GROUP BY  S.age, S.level,
HAVING    S.level IN (SELECT   S1.level
                      FROM     Student S1
                      WHERE    S1.age = S.age
                      GROUP BY S1.level, S1.age
                      HAVING   COUNT (*) >= ALL (SELECT   COUNT (*)
                                                 FROM     Student S2
                                                 WHERE s1.age = S2.age
                                                 GROUP BY S2.level, S2.age))
```

Exercise 5.2 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. Find the *pname*s of parts for which there is some supplier.
2. Find the *sname*s of suppliers who supply every part.
3. Find the *sname*s of suppliers who supply every red part.
4. Find the *pname*s of parts supplied by Acme Widget Suppliers and no one else.
5. Find the *sid*s of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
6. For each part, find the *sname* of the supplier who charges the most for that part.
7. Find the *sid*s of suppliers who supply only red parts.
8. Find the *sid*s of suppliers who supply a red part and a green part.

9. Find the *sid*s of suppliers who supply a red part or a green part.

10. For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.

11. For every supplier that supplies a green part and a red part, print the name and price of the most expensive part that she supplies.

Answer 5.2 The answers are given below:

1.
```
SELECT DISTINCT P.pname
FROM   Parts P, Catalog C
WHERE  P.pid = C.pid
```

2.
```
SELECT S.sname
FROM   Suppliers S
WHERE  NOT EXISTS (( SELECT P.pid
                     FROM   Parts P )
                   EXCEPT
                   ( SELECT C.pid
                     FROM   Catalog C
                     WHERE  C.sid = S.sid ))
```

3.
```
SELECT S.sname
FROM   Suppliers S
WHERE  NOT EXISTS (( SELECT P.pid
                     FROM   Parts P
                     WHERE  P.color = 'Red' )
                   EXCEPT
                   ( SELECT C.pid
                     FROM   Catalog C, Parts P
                     WHERE  C.sid = S.sid AND
                            C.pid = P.pid AND P.color = 'Red' ))
```

4.
```
SELECT P.pname
FROM   Parts P, Catalog C, Suppliers S
WHERE  P.pid = C.pid AND C.sid = S.sid
AND    S.sname = 'Acme Widget Suppliers'
AND    NOT EXISTS ( SELECT *
                    FROM   Catalog C1, Suppliers S1
                    WHERE  P.pid = C1.pid AND C1.sid = S1.sid AND
                           S1.sname <> 'Acme Widget Suppliers' )
```

5.
```
SELECT DISTINCT C.sid
FROM   Catalog C
```

```
WHERE  C.cost > ( SELECT AVG (C1.cost)
                  FROM   Catalog C1
                  WHERE  C1.pid = C.pid )
```

6.
```
SELECT P.pid, S.sname
FROM   Parts P, Suppliers S, Catalog C
WHERE  C.pid = P.pid
AND    C.sid = S.sid
AND    C.cost = (SELECT MAX (C1.cost)
                 FROM   Catalog C1
                 WHERE  C1.pid = P.pid)
```

7.
```
SELECT DISTINCT C.sid
FROM   Catalog C
WHERE  NOT EXISTS ( SELECT *
                    FROM   Parts P
                    WHERE  P.pid = C.pid AND P.color <> 'Red' )
```

8.
```
SELECT DISTINCT C.sid
FROM   Catalog C, Parts P
WHERE  C.pid = P.pid AND P.color = 'Red'
INTERSECT
SELECT DISTINCT C1.sid
FROM   Catalog C1, Parts P1
WHERE  C1.pid = P1.pid AND P1.color = 'Green'
```

9.
```
SELECT DISTINCT C.sid
FROM   Catalog C, Parts P
WHERE  C.pid = P.pid AND P.color = 'Red'
UNION
SELECT DISTINCT C1.sid
FROM   Catalog C1, Parts P1
WHERE  C1.pid = P1.pid AND P1.color = 'Green'
```

10.
```
SELECT   S.sname, COUNT(*) as PartCount
FROM     Suppliers S, Parts P, Catalog C
WHERE    P.pid = C.pid AND C.sid = S.sid
GROUP BY S.sname, S.sid
HAVING   EVERY (P.color='Green')
```

11.
```
SELECT   S.sname, MAX(C.cost) as MaxCost
FROM     Suppliers S, Parts P, Catalog C
WHERE    P.pid = C.pid AND C.sid = S.sid
```

```
GROUP BY S.sname, S.sid
HAVING   ANY ( P.color='green' ) AND ANY ( P.color = 'red' )
```

Exercise 5.3 The following relations keep track of airline flight information:

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`, *distance:* `integer`, *departs:* `time`, *arrives:* `time`, *price:* `real`)
Aircraft(*aid:* `integer`, *aname:* `string`, *cruisingrange:* `integer`)
Certified(*eid:* `integer`, *aid:* `integer`)
Employees(*eid:* `integer`, *ename:* `string`, *salary:* `integer`)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft, and only pilots are certified to fly. Write each of the following queries in SQL. (*Additional queries using the same schema are listed in the exercises for Chapter 4.*)

1. Find the names of aircraft such that all pilots certified to operate them have salaries more than $80,000.
2. For each pilot who is certified for more than three aircraft, find the *eid* and the maximum *cruisingrange* of the aircraft for which she or he is certified.
3. Find the names of pilots whose *salary* is less than the price of the cheapest route from Los Angeles to Honolulu.
4. For all aircraft with *cruisingrange* over 1000 miles, find the name of the aircraft and the average salary of all pilots certified for this aircraft.
5. Find the names of pilots certified for some Boeing aircraft.
6. Find the *aid*s of all aircraft that can be used on routes from Los Angeles to Chicago.
7. Identify the routes that can be piloted by every pilot who makes more than $100,000.
8. Print the *ename*s of pilots who can operate planes with *cruisingrange* greater than 3000 miles but are not certified on any Boeing aircraft.
9. A customer wants to travel from Madison to New York with no more than two changes of flight. List the choice of departure times from Madison if the customer wants to arrive in New York by 6 p.m.
10. Compute the difference between the average salary of a pilot and the average salary of all employees (including pilots).

11. Print the name and salary of every nonpilot whose salary is more than the average salary for pilots.
12. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles.
13. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles, but on at least two such aircrafts.
14. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles and who are certified on some Boeing aircraft.

Answer 5.3 The answers are given below:

1.
```
SELECT DISTINCT A.aname
FROM    Aircraft A
WHERE   A.Aid IN (SELECT C.aid
                  FROM    Certified C, Employees E
                  WHERE   C.eid = E.eid AND
                  NOT EXISTS ( SELECT *
                               FROM    Employees E1
                               WHERE   E1.eid = E.eid AND E1.salary < 80000 ))
```

2.
```
SELECT    C.eid, MAX (A.cruisingrange)
FROM      Certified C, Aircraft A
WHERE     C.aid = A.aid
GROUP BY C.eid
HAVING    COUNT (*) > 3
```

3.
```
SELECT DISTINCT E.ename
FROM    Employees E
WHERE   E.salary < ( SELECT MIN (F.price)
                     FROM    Flights F
                     WHERE   F.from = 'Los Angeles' AND F.to = 'Honolulu' )
```

4. Observe that *aid* is the key for Aircraft, but the question asks for aircraft names; we deal with this complication by using an intermediate relation Temp:

```
SELECT Temp.name, Temp.AvgSalary
FROM    ( SELECT    A.aid, A.aname AS name,
                    AVG (E.salary) AS AvgSalary
          FROM      Aircraft A, Certified C, Employees E
          WHERE     A.aid = C.aid AND
                    C.eid = E.eid AND A.cruisingrange > 1000
          GROUP BY A.aid, A.aname ) AS Temp
```

5.
```
SELECT DISTINCT E.ename
FROM   Employees E, Certified C, Aircraft A
WHERE  E.eid = C.eid AND
       C.aid = A.aid AND
       A.aname LIKE 'Boeing%'
```

6.
```
SELECT A.aid
FROM   Aircraft A
WHERE  A.cruisingrange > ( SELECT MIN (F.distance)
                           FROM   Flights F
                           WHERE  F.from = 'Los Angeles' AND F.to = 'Chicago' )
```

7.
```
SELECT DISTINCT F.from, F.to
FROM   Flights F
WHERE  NOT EXISTS ( SELECT *
                    FROM   Employees E
                    WHERE  E.salary > 100000
                    AND
                    NOT EXISTS (SELECT *
                                FROM   Aircraft A, Certified C
                                WHERE  A.cruisingrange > F.distance
                                AND E.eid = C.eid
                                AND A.aid = C.aid) )
```

8.
```
SELECT DISTINCT E.ename
FROM   Employees E
WHERE  E.eid IN ( ( SELECT C.eid
                    FROM   Certified C
                    WHERE  EXISTS ( SELECT A.aid
                                    FROM   Aircraft A
                                    WHERE  A.aid = C.aid
                                    AND    A.cruisingrange > 3000 )
                    AND
                    NOT EXISTS ( SELECT A1.aid
                                 FROM   Aircraft A1
                                 WHERE  A1.aid = C.aid
                                 AND    A1.aname LIKE 'Boeing%' ))
```

9.
```
SELECT F.departs
FROM   Flights F
WHERE  F.flno IN ( ( SELECT F0.flno
```

```
        FROM    Flights F0
        WHERE   F0.from = 'Madison' AND F0.to = 'New York'
                AND F0.arrives < '18:00' )
    UNION
    ( SELECT F0.flno
        FROM    Flights F0, Flights F1
        WHERE   F0.from = 'Madison' AND F0.to <> 'New York'
                AND F0.to = F1.from AND F1.to = 'New York'
                AND F1.departs > F0.arrives
                AND F1.arrives < '18:00' )
    UNION
    ( SELECT F0.flno
        FROM    Flights F0, Flights F1, Flights F2
        WHERE   F0.from = 'Madison'
                AND F0.to = F1.from
                AND F1.to = F2.from
                AND F2.to = 'New York'
                AND F0.to <> 'New York'
                AND F1.to <> 'New York'
                AND F1.departs > F0.arrives
                AND F2.departs > F1.arrives
                AND F2.arrives < '18:00' ))
```

10.

```
SELECT Temp1.avg - Temp2.avg
FROM   (SELECT AVG (E.salary) AS avg
        FROM   Employees E
        WHERE  E.eid IN (SELECT DISTINCT C.eid
                         FROM Certified C )) AS Temp1,
       (SELECT AVG (E1.salary) AS avg
        FROM   Employees E1 ) AS Temp2
```

11.

```
SELECT E.ename, E.salary
FROM   Employees E
WHERE  E.eid NOT IN ( SELECT DISTINCT C.eid
                      FROM    Certified C )
AND E.salary > ( SELECT AVG (E1.salary)
                 FROM   Employees E1
                 WHERE  E1.eid IN
                        ( SELECT DISTINCT C1.eid
                          FROM    Certified C1 ) )
```

12.

```
SELECT    E.ename
```

```
FROM      Employees E, Certified C, Aircraft A
WHERE     C.aid = A.aid AND E.eid = C.eid
GROUP BY  E.eid, E.ename
HAVING    EVERY (A.cruisingrange > 1000)
```

13.
```
SELECT    E.ename
FROM      Employees E, Certified C, Aircraft A
WHERE     C.aid = A.aid AND E.eid = C.eid
GROUP BY  E.eid, E.ename
HAVING    EVERY (A.cruisingrange > 1000) AND COUNT (*) > 1
```

14.
```
SELECT    E.ename
FROM      Employees E, Certified C, Aircraft A
WHERE     C.aid = A.aid AND E.eid = C.eid
GROUP BY  E.eid, E.ename
HAVING    EVERY (A.cruisingrange > 1000) AND ANY (A.aname = 'Boeing')
```

Exercise 5.4 Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the Works relation shows the percentage of time that a given employee works in a given department.

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *budget:* `real`, *managerid:* `integer`)

Write the following queries in SQL:

1. Print the names and ages of each employee who works in both the Hardware department and the Software department.
2. For each department with more than 20 full-time-equivalent employees (i.e., where the part-time and full-time employees add up to at least that many full-time employees), print the *did* together with the number of employees that work in that department.
3. Print the name of each employee whose salary exceeds the budget of all of the departments that he or she works in.
4. Find the *managerid*s of managers who manage only departments with budgets greater than $1 million.
5. Find the *ename*s of managers who manage the departments with the largest budgets.
6. If a manager manages more than one department, he or she *controls* the sum of all the budgets for those departments. Find the *managerid*s of managers who control more than $5 million.

7. Find the *managerid*s of managers who control the largest amounts.

8. Find the *ename*s of managers who manage only departments with budgets larger than $1 million, but at least one department with budget less than $5 million.

Answer 5.4 The answers are given below:

1.
```
SELECT E.ename, E.age
FROM   Emp E, Works W1, Works W2, Dept D1, Dept D2
WHERE  E.eid = W1.eid AND W1.did = D1.did AND D1.dname = 'Hardware' AND
       E.eid = W2.eid AND W2.did = D2.did AND D2.dname = 'Software'
```

2.
```
SELECT   W.did, COUNT (W.eid)
FROM     Works W
GROUP BY W.did
HAVING   2000 < ( SELECT SUM (W1.pct_time)
                  FROM   Works W1
                  WHERE  W1.did = W.did )
```

3.
```
SELECT E.ename
FROM   Emp E
WHERE  E.salary > ALL (SELECT D.budget
                       FROM   Dept D, Works W
                       WHERE  E.eid = W.eid AND D.did = W.did)
```

4.
```
SELECT DISTINCT D.managerid
FROM   Dept D
WHERE  1000000 < ALL (SELECT D2.budget
                      FROM Dept D2
                      WHERE D2.managerid = D.managerid )
```

5.
```
SELECT E.ename
FROM   Emp E
WHERE  E.eid IN (SELECT D.managerid
                 FROM   Dept D
                 WHERE  D.budget = (SELECT MAX (D2.budget)
                                    FROM Dept D2))
```

6.
```
SELECT D.managerid
FROM   Dept D
WHERE  5000000 < (SELECT SUM (D2.budget)
                  FROM Dept D2
                  WHERE D2.managerid = D.managerid )
```

sid	*sname*	*rating*	*age*
18	jones	3	30.0
41	jonah	6	56.0
22	ahab	7	44.0
63	moby	*null*	15.0

Figure 5.1 An Instance of Sailors

7.
```
SELECT DISTINCT tempD.managerid
FROM   (SELECT DISTINCT D.managerid, SUM (D.budget) AS tempBudget
        FROM   Dept D
        GROUP BY D.managerid ) AS tempD
WHERE  tempD.tempBudget = (SELECT MAX (tempD.tempBudget)
                           FROM   tempD)
```

8.
```
SELECT E.ename
FROM   Emp E, Dept D
WHERE  E.eid = D.managerid GROUP BY E.Eid, E.ename
HAVING EVERY (D.budget > 1000000) AND ANY (D.budget < 5000000)
```

Exercise 5.5 Consider the instance of the Sailors relation shown in Figure 5.1.

1. Write SQL queries to compute the average rating, using `AVG`; the sum of the ratings, using `SUM`; and the number of ratings, using `COUNT`.
2. If you divide the sum just computed by the count, would the result be the same as the average? How would your answer change if these steps were carried out with respect to the *age* field instead of *rating*?
3. Consider the following query: *Find the names of sailors with a higher rating than all sailors with age* < 21. The following two SQL queries attempt to obtain the answer to this question. Do they both compute the result? If not, explain why. Under what conditions would they compute the same result?

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS ( SELECT *
                    FROM   Sailors S2
                    WHERE  S2.age < 21
                           AND S.rating <= S2.rating )
```

```
SELECT *
FROM   Sailors S
```

```
WHERE  S.rating > ANY ( SELECT S2.rating
                        FROM   Sailors S2
                        WHERE  S2.age < 21 )
```

4. Consider the instance of Sailors shown in Figure 5.1. Let us define instance S1 of Sailors to consist of the first two tuples, instance S2 to be the last two tuples, and S to be the given instance.
   (a) Show the left outer join of S with itself, with the join condition being *sid=sid*.
   (b) Show the right outer join of S with itself, with the join condition being *sid=sid*.
   (c) Show the full outer join of S with itself, with the join condition being *sid=sid*.
   (d) Show the left outer join of S1 with S2, with the join condition being *sid=sid*.
   (e) Show the right outer join of S1 with S2, with the join condition being *sid=sid*.
   (f) Show the full outer join of S1 with S2, with the join condition being *sid=sid*.

Answer 5.5 The answers are shown below:

1.
```
SELECT AVG (S.rating) AS AVERAGE
FROM   Sailors S

SELECT SUM (S.rating)
FROM   Sailors S

SELECT COUNT (S.rating)
FROM   Sailors S
```

2. The result using SUM and COUNT would be smaller than the result using AVERAGE if there are tuples with rating = NULL. This is because all the aggregate operators, except for COUNT, ignore NULL values. So the first approach would compute the average over all tuples while the second approach would compute the average over all tuples with non-NULL rating values. However, if the aggregation is done on the age field, the answers using both approaches would be the same since the age field does not take NULL values.
3. Only the first query is correct. The second query returns the names of sailors with a higher rating than *at least one* sailor with age < 21. Note that the answer to the second query does not necessarily contain the answer to the first query. In particular, if all the sailors are at least 21 years old, the second query will return an empty set while the first query will return all the sailors. This is because the NOT EXISTS predicate in the first query will evaluate to *true* if its subquery evaluates to an empty set, while the ANY predicate in the second query will evaluate to *false* if its subquery evaluates to an empty set. The two queries give the same results if and only if one of the following two conditions hold:

4. (a)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
18	jones	3	30.0	18	jones	3	30.0
41	jonah	6	56.0	41	jonah	6	56.0
22	ahab	7	44.0	22	ahab	7	44.0
63	moby	*null*	15.0	63	moby	*null*	15.0

(b)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
18	jones	3	30.0	18	jones	3	30.0
41	jonah	6	56.0	41	jonah	6	56.0
22	ahab	7	44.0	22	ahab	7	44.0
63	moby	*null*	15.0	63	moby	*null*	15.0

(c)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
18	jones	3	30.0	18	jones	3	30.0
41	jonah	6	56.0	41	jonah	6	56.0
22	ahab	7	44.0	22	ahab	7	44.0
63	moby	*null*	15.0	63	moby	*null*	15.0

- The *Sailors* relation is empty, or
- There is at least one sailor with age > 21 in the *Sailors* relation, and for every sailor s, either s has a higher rating than all sailors under 21 or s has a rating no higher than all sailors under 21.

Exercise 5.6 Answer the following questions:

(d)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
18	jones	3	30.0	*null*	*null*	*null*	*null*
41	jonah	6	56.0	*null*	*null*	*null*	*null*

(e)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
null	*null*	*null*	*null*	22	ahab	7	44.0
null	*null*	*null*	*null*	63	moby	*null*	15.0

(f)

sid	*sname*	*rating*	*age*	*sid*	*sname*	*rating*	*age*
18	jones	3	30.0	*null*	*null*	*null*	*null*
41	jonah	6	56.0	*null*	*null*	*null*	*null*
null	*null*	*null*	*null*	22	ahab	7	44.0
null	*null*	*null*	*null*	63	moby	*null*	15.0

1. Explain the term *impedance mismatch* in the context of embedding SQL commands in a host language such as C.
2. How can the value of a host language variable be passed to an embedded SQL command?
3. Explain the `WHENEVER` command's use in error and exception handling.
4. Explain the need for cursors.
5. Give an example of a situation that calls for the use of embedded SQL; that is, interactive use of SQL commands is not enough, and some host language capabilities are needed.
6. Write a C program with embedded SQL commands to address your example in the previous answer.
7. Write a C program with embedded SQL commands to find the standard deviation of sailors' ages.
8. Extend the previous program to find all sailors whose age is within one standard deviation of the average age of all sailors.
9. Explain how you would write a C program to compute the transitive closure of a graph, represented as an SQL relation Edges(*from, to*), using embedded SQL commands. (You need not write the program, just explain the main points to be dealt with.)
10. Explain the following terms with respect to cursors: *updatability, sensitivity*, and *scrollability*.
11. Define a cursor on the Sailors relation that is updatable, scrollable, and returns answers sorted by *age*. Which fields of Sailors can such a cursor *not* update? Why?
12. Give an example of a situation that calls for dynamic SQL; that is, even embedded SQL is not sufficient.

Answer 5.6 Each question is answered in turn:

1. The *impedance mismatch* between SQL and many host languages such as C or Java arises because SQL operates on *sets*, and there is no clean abstraction for sets in a host language.
2. Variables in a host language must first be declared between `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` commands. Once variables are declared, they can be used by prefixing the variable name with a colon (:).

3. The `WHENEVER` command in SQL allows for easy error and exception checking after an embedded SQL statement is executed. `WHENEVER` checks the value of `SQLSTATE` for a specified error. If an error has occurred, the `WHENEVER` command will transfer control to a specified section of error handling code.

4. Cursors provide a mechanism for retrieving rows one at a time from a relation. A cursor is the abstraction that is missing from most host languages, causing an *impedance mismatch*.

5. One example where SQL is insufficient is when the variance of some data is needed. Although SQL has many useful aggregate functions such as `COUNT` and `AVG` , these are not powerful enough to compute variances. In this case, users can use embedded SQL to perform more involved aggregates.

6. Assume we have a table of data describing the height of different trees with the following schema:

   Tree(*tid:* `integer`, *tname:* `string`, *theight:* `real`)

   A C function that computes and returns the variance is:

```
float Variance (void) {
       float mean;
       float variance;
       int count;
       EXEC SQL BEGIN DECLARE SECTION
       float height;
       EXEC SQL END DECLARE SECTION

       mean = 0.0;
       variance = 0.0;
       count = 0;

       DECLARE trees CURSOR FOR
       SELECT T.theight
       FROM Tree T;

       OPEN trees;
       FETCH trees INTO :height;
       while (strcmp(SQLSTATE, "02000") != 0) {
              count = count + 1;
              mean = mean + height;
```

```
                variance = variance + pow(height, 2);
                FETCH trees INTO :height;
        }
        CLOSE trees;

        mean = mean / count;
        variance = variance / count - pow(mean, 2);

        return variance;
}
```

7. A C function that computes and returns the standard deviation of the sailors' ages is:

```
float StdDev (void) {
        float mean;
        float variance;
        int count;
        EXEC SQL BEGIN DECLARE SECTION
        float age;
        EXEC SQL END DECLARE SECTION

        mean = 0.0;
        variance = 0.0;
        count = 0;

        DECLARE ages CURSOR FOR
        SELECT S.age
        FROM Sailors S;

        OPEN ages;
        FETCH ages INTO :age;
        while (strcmp(SQLSTATE, "02000") != 0) {
                count = count + 1;
                mean = mean + age;
                variance = variance + pow(age, 2);
                FETCH ages INTO :age;
        }
        CLOSE ages;

        mean = mean / count;
        variance = variance / count - pow(mean, 2);
```

```
        return sqrt(variance);
}
```

8. A program that prints out all the sailors whose age is within one standard deviation of the average age:

```
void main (void) {
        float stdDev;
        float mean;
        float variance;
        int count;

        EXEC SQL BEGIN DECLARE SECTION
        int sid;
        char sname[50];
        float age;
        EXEC SQL END DECLARE SECTION

        mean = 0.0;
        variance = 0.0;
        count = 0;

        DECLARE ages CURSOR FOR
        SELECT S.age
        FROM Sailors S;

        OPEN ages;
        FETCH ages INTO :age;
        while (strcmp(SQLSTATE, "02000") != 0) {
                count = count + 1;
                mean = mean + age;
                variance = variance + pow(age, 2);
                FETCH ages INTO :age;
        }
        CLOSE ages;

        mean = mean / count;
        variance = variance / count - pow(mean, 2);

        stdDev = sqrt(variance);

        DECLARE sailors CURSOR FOR
        SELECT S.sid, S.sname, S.age
```

```
        FROM Sailors S;

        OPEN sailors ;
        FETCH sailors INTO :sid, :sname, :age;
        while (strcmp(SQLSTATE, "02000") != 0) {
               if(fabs(:age - mean) < stdDev)
                      printf("%d/t%s/t%f/n", :sid, :sname, :age);
               FETCH sailors INTO :sid, :sname, :age;
        }
        CLOSE sailors;
}
```

9. One popular way to find the transitive closure of a directed graph is to use variations of the Floyd-Warshall algorithm. This involve create an adjacency matrix, which can easily be done by selecting all of the records in the Edges table to create the matrix. Then we can execute any transitive closure algorithm we wish.

10. *Updatablity* is a property of cursors that determines whether or not the cursor can be used to perform `UPDATE` or `DELETE` SQL statements. Cursors explicity declared `FOR UPDATE` (the default value for a cursor declaration is for it to be updatable) can perform queries that modify relations or views. *Sensitivity* involves concurrancy issues. If a cursor is declared as `INSENSITIVE`, then it behaves as if it has a private, read-only copy of the query results (concurent updates to the data do not change the cursor). Without the `INSENSITIVE` property, there are many concurency issues that arise and behavior is implementation specific. *Scrollability* refers to how the `FETCH` command works. A cursor with the keyword `SCROLL` can be positioned in a variety of ways, otherwise only the next row can be retrieved from a cursor.

11.
```
DECLARE SailorAge SCROLL CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
ORDER BY S.age ASC
FOR UPDATE
```

    In this case, the cursor cannot update the age field because that is the field the data is sorted on.

12. Dynamic SQL is used when the SQL statements are not known at compile time. A situation in which this occurs is when a user has a graphical user interface to a table, and can select various values to update, delete, or insert.

Exercise 5.7 Consider the following relational schema and briefly answer the questions that follow:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *budget:* `real`, *managerid:* `integer`)

1. Define a table constraint on Emp that will ensure that every employee makes at least $10,000.
2. Define a table constraint on Dept that will ensure that all managers have $age > 30$.
3. Define an assertion on Dept that will ensure that all managers have $age > 30$. Compare this assertion with the equivalent table constraint. Explain which is better.
4. Write SQL statements to delete all information about employees whose salaries exceed that of the manager of one or more departments that they work in. Be sure to ensure that all the relevant integrity constraints are satisfied after your updates.

Answer 5.7 The answers are given below:

1. Define a table constraint on Emp that will ensure that every employee makes at least $10,000

```
CREATE TABLE Emp ( eid      INTEGER,
                   ename    CHAR(10),
                   age      INTEGER ,
                   salary   REAL,
                   PRIMARY KEY (eid),
                   CHECK ( salary >= 10000 ))
```

2. Define a table constraint on Dept that will ensure that all managers have $age > 30$

```
CREATE TABLE Dept ( did          INTEGER,
                    buget        REAL,
                    managerid    INTEGER ,
                    PRIMARY KEY (did),
                    FOREIGN KEY (managerid) REFERENCES Emp,
                    CHECK (      ( SELECT E.age FROM Emp E, Dept D)
                                 WHERE E.eid = D.managerid ) > 30 )
```

3. Define an assertion on Dept that will ensure that all managers have age > 30

```
CREATE TABLE Dept ( did          INTEGER,
                    budget       REAL,
                    managerid    INTEGER ,
                    PRIMARY KEY (did) )
```

```
CREATE ASSERTION managerAge
CHECK ((SELECT E.age
        FROM   Emp E, Dept D
        WHERE  E.eid = D.managerid ) > 30 )
```

Since the constraint involves two relations, it is better to define it as an assertion, independent of any one relation, rather than as a check condition on the Dept relation. The limitation of the latter approach is that the condition is checked only when the Dept relation is being updated. However, since age is an attribute of the Emp relation, it is possible to update the age of a manager which violates the constraint. So the former approach is better since it checks for potential violation of the assertion whenever one of the relations is updated.

4. To write such statements, it is necessary to consider the constraints defined over the tables. We will assume the following:

```
CREATE TABLE Emp (    eid         INTEGER,
                      ename       CHAR(10),
                      age         INTEGER,
                      salary      REAL,
                      PRIMARY KEY (eid) )

CREATE TABLE Works (  eid         INTEGER,
                      did         INTEGER,
                      pcttime     INTEGER,
                      PRIMARY KEY (eid, did),
                      FOREIGN KEY (did) REFERENCES Dept,
                      FOREIGN KEY (eid) REFERENCES Emp,
                      ON DELETE CASCADE)

CREATE TABLE Dept (   did         INTEGER,
                      buget       REAL,
                      managerid   INTEGER ,
                      PRIMARY KEY (did),
                      FOREIGN KEY (managerid) REFERENCES Emp,
                      ON DELETE SET NULL)
```

Now, we can define statements to delete employees who make more than one of their managers:

```
DELETE
FROM      Emp E
WHERE     E.eid IN (  SELECT W.eid
                      FROM   Work W, Emp E2, Dept D
                      WHERE  W.did = D.did
```

```
AND     D.managerid = E2.eid
AND     E.salary > E2.salary )
```

Exercise 5.8 Consider the following relations:

Student(*snum:* `integer`, *sname:* `string`, *major:* `string`,
level: `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid:* `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

1. Write the SQL statements required to create these relations, including appropriate versions of all primary and foreign key integrity constraints.
2. Express each of the following integrity constraints in SQL unless it is implied by the primary and foreign key constraint; if so, explain how it is implied. If the constraint cannot be expressed in SQL, say so. For each constraint, state what operations (inserts, deletes, and updates on specific relations) must be monitored to enforce the constraint.
   (a) Every class has a minimum enrollment of 5 students and a maximum enrollment of 30 students.
   (b) At least one class meets in each room.
   (c) Every faculty member must teach at least two courses.
   (d) Only faculty in the department with *deptid=33* teach more than three courses.
   (e) Every student must be enrolled in the course called Math101.
   (f) The room in which the earliest scheduled class (i.e., the class with the smallest *meets_at* value) meets should not be the same as the room in which the latest scheduled class meets.
   (g) Two classes cannot meet in the same room at the same time.
   (h) The department with the most faculty members must have fewer than twice the number of faculty members in the department with the fewest faculty members.
   (i) No department can have more than 10 faculty members.
   (j) A student cannot add more than two courses at a time (i.e., in a single update).
   (k) The number of CS majors must be more than the number of Math majors.

(l) The number of distinct courses in which CS majors are enrolled is greater than the number of distinct courses in which Math majors are enrolled.

(m) The total enrollment in courses taught by faculty in the department with *deptid=33* is greater than the number of Math majors.

(n) There must be at least one CS major if there are any students whatsoever.

(o) Faculty members from different departments cannot teach in the same room.

Answer 5.8 Answers are given below.

1. The SQL statements needed to create the tables are given below:

```
CREATE TABLE Student ( snum    INTEGER,
                       sname   CHAR(20),
                       major   CHAR(20),
                       level   CHAR(20),
                       age     INTEGER,
                       PRIMARY KEY (snum))

CREATE TABLE Faculty ( fid     INTEGER,
                       fname   CHAR(20),
                       deptid  INTEGER,
                       PRIMARY KEY (fnum))

CREATE TABLE Class (   name    CHAR(20),
                       meets_atTIME,
                       room    CHAR(10),
                       fid     INTEGER,
                       PRIMARY KEY (name),
                       FOREIGN KEY (fid) REFERENCES Faculty)

CREATE TABLE Enrolled (snum    INTEGER,
                       cname   CHAR(20),
                       PRIMARY KEY (snum, cname),
                       FOREIGN KEY (snum) REFERENCES Student),
                       FOREIGN KEY (cname) REFERENCES Class)
```

2. The answer to each question is given below

   (a) The Enrolled table should be modified as follows:

```
CREATE TABLE Enrolled (snum      INTEGER,
                       cname     CHAR(20),
```

```
                                PRIMARY KEY (snum, cname),
                                FOREIGN KEY (snum) REFERENCES Student),
                                FOREIGN KEY (cname) REFERENCES Class,),
                                CHECK ((  SELECT    COUNT (E.snum)
                                          FROM      Enrolled E
                                          GROUP BY  E.cname) >= 5),
                                CHECK ((  SELECT    COUNT (E.snum)
                                          FROM      Enrolled E
                                          GROUP BY  E.cname) <= 30))
```

(b) This constraint is already guaranteed because rooms are associated with classes, and thus a new room cannot be declared without an associated class in it.

(c) Create an assertion as follows:

```
CREATE ASSERTION TeachTwo
CHECK ( (  SELECT    COUNT (*)
           FROM      Facult F, Class C
           WHERE     F.fid = C.fid
           GROUP BY  C.fid
           HAVING    COUNT (*) < 2) = 0)
```

(d) Create an assertion as follows:

```
CREATE ASSERTION NoTeachThree
CHECK ( (  SELECT    COUNT (*)
           FROM      Facult F, Class C
           WHERE     F.fid = C.fid AND F.deptid ≠ 33
           GROUP BY  C.fid
           HAVING    COUNT (*) > 3) = 0)
```

(e) Create an assertion as follows:

```
CREATE ASSERTION InMath101
CHECK ((  SELECT    COUNT (*)
          FROM      Student S
          WHERE     S.snum NOT IN (  SELECT     E.snum
                                     FROM       Enrolled E
                                     WHERE      E.cname = 'Math101')) = 0)
```

(f) The Class table should be modified as follows:

```
CREATE TABLE Class (   name       CHAR(20),
                       meets_at   TIME,
                       room       CHAR(10),
                       fid        INTEGER,
```

```
                                PRIMARY KEY (name),
                                FOREIGN KEY (fid) REFERENCES Faculty),
                                CHECK (   (SELECT   MIN (meets_at)
                                          FROM      Class) <>
                                          (SELECT   MAX (meets_at)
                                          FROM      Class)))
```

(g) The Class table should be modified as follows:

```
CREATE TABLE Class (   name      CHAR(20),
                       meets_at  TIME,
                       room      CHAR(10),
                       fid       INTEGER,
                       PRIMARY KEY (name),
                       FOREIGN KEY (fid) REFERENCES Faculty),
                       CHECK ((SELECT COUNT (*)
                                FROM (    SELECT C.room, C.meets
                                          FROM Class C
                                          GROUP BY C.room, C.meets
                                          HAVING COUNT (*) > 1)) = 0))
```

(h) The Faculty table should be modified as follows:

```
CREATE TABLE Faculty ( fid       INTEGER,
                       fname     CHAR(20),
                       deptid    INTEGER,
                       PRIMARY KEY (fnum),
                       CHECK (   (SELECT MAX (*)
                                 FROM ( SELECT COUNT (*)
                                        FROM Faculty F
                                        GROUP BY F.deptid))
                                 < 2 *
                                 (SELECT MIN (*)
                                 FROM ( SELECT COUNT (*)
                                        FROM Faculty F
                                        GROUP BY F.deptid))))
```

(i) The Faculty table should be modified as follows:

```
CREATE TABLE Faculty (fid        INTEGER,
                      fname      CHAR(20),
                      deptid     INTEGER,
                      PRIMARY KEY (fnum),
                      CHECK ( ( SELECT    COUNT (*)
                                FROM      Faculty F
                                GROUP BY  F.deptid
                                HAVING    COUNT (*) > 10) = 0))
```

(j) This constraint cannot be done because integratey constraints and assertions only affect the content of a table, not how that content is manipulated.

(k) The Student table should be modified as follows:

```
CREATE TABLE Student ( snum    INTEGER,
                       sname   CHAR(20),
                       major   CHAR(20),
                       level   CHAR(20),
                       age     INTEGER,
                       PRIMARY KEY (snum),
                       CHECK ((SELECT COUNT (*)
                               FROM Student S
                               WHERE S.major = 'CS') >
                               (SELECT COUNT (*)
                               FROM Student S
                               WHERE S.major = 'Math')))
```

(l) Create an assertion as follows:

```
CREATE ASSERTION MoreCSMajors
CHECK (   (SELECT   COUNT (E.cname)
           FROM     Enrolled E, Student S
           WHERE    S.snum = E.snum AND S.major = 'CS') >
           (SELECT  COUNT (E.cname)
           FROM     Enrolled E, Student S
           WHERE    S.snum = E.snum AND S.major = 'Math'))
```

(m) Create an assertion as follows:

```
CREATE ASSERTION MoreEnrolledThanMath
CHECK (   (SELECT   COUNT (E.snum)
           FROM     Enrolled E, Faculty F, Class C
           WHERE    E.cname = C.name
           AND C.fid = F.fid AND F.deptid = 33) >
           (SELECT  COUNT (E.snum)
           FROM     Student S
           WHERE    S.major = 'Math'))
```

(n) The Student table should be modified as follows:

```
CREATE TABLE Student ( snum    INTEGER,
                       sname   CHAR(20),
                       major   CHAR(20),
```

```
level    CHAR(20),
age      INTEGER,
PRIMARY KEY (snum),
CHECK ( (SELECT COUNT (S.snum)
         FROM Student S
         WHERE S.major = 'CS') > 0 ))
```

(o) Create an assertion as follows:

```
CREATE ASSERTION NotSameRoom
CHECK (   (SELECT   COUNT (*)
           FROM     Faculty F1, Faculty F2, Class C1, Class C2
           WHERE    F1.fid = C1.fid
           AND      F2.fid = C2.fid
           AND      C1.room = C2.room
           AND      F1.deptid ≠ F2.deptid) = 0)
```

Exercise 5.9 Discuss the strengths and weaknesses of the trigger mechanism. Contrast triggers with other integrity constraints supported by SQL.

Answer 5.9 A trigger is a procedure that is automatically invoked in response to a specified change to the database. The advantages of the trigger mechanism include the ability to perform an action based on the result of a query condition. The set of actions that can be taken is a superset of the actions that integrity constraints can take (i.e. report an error). Actions can include invoking new update, delete, or insert queries, perform data definition statements to create new tables or views, or alter security policies. Triggers can also be executed before or after a change is made to the database (that is, use old or new data).

There are also disadvantages to triggers. These include the added complexity when trying to match database modifications to trigger events. Also, integrity constraints are incorporated into database performance optimization; it is more difficult for a database to perform automatic optimization with triggers. If database consistency is the primary goal, then integrity constraints offer the same power as triggers. Integrity constraints are often easier to understand than triggers.

Exercise 5.10 Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the Works relation shows the percentage of time that a given employee works in a given department.

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *budget:* `real`, *managerid:* `integer`)

Write SQL-92 integrity constraints (domain, key, foreign key, or CHECK constraints; or assertions) or SQL:1999 triggers to ensure each of the following requirements, considered independently.

1. Employees must make a minimum salary of $1000.
2. Every manager must be also be an employee.
3. The total percentage of all appointments for an employee must be under 100%.
4. A manager must always have a higher salary than any employee that he or she manages.
5. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much.
6. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much. Further, whenever an employee is given a raise, the department's budget must be increased to be greater than the sum of salaries of all employees in the department.

Answer 5.10 The answer to each question is given below.

1. This constraint can be added by modifying the Emp table:

```
CREATE TABLE Emp (  eid        INTEGER,
                    ename      CHAR(20),
                    age        INTEGER,
                    salary     REAL,
                    PRIMARY KEY (eid),
                    CHECK      ( salary > 1000))
```

2. Create an assertion as follows:

```
CREATE ASSERTION ManagerIsEmployee
CHECK ( ( SELECT    COUNT (*)
          FROM      Dept D
          WHERE     D.managerid NOT IN
                    (SELECT * FROM Emp))
          = 0)
```

3. This constraint can be added by modifying the Works table:

```
CREATE TABLE Works ( eid        INTEGER,
                     did        INTEGER,
                     pct_time   INTEGER,
                     PRIMARY KEY (eid, did),
                     CHECK (    (SELECT   COUNT (W.eid)
                                 FROM     Works W
                                 GROUP BY W.eid
                                 HAVING   Sum(pct_time) > 100) = 0))
```

4. Create an assertion as follows:

```
CREATE ASSERTION ManagerHigherSalary
CHECK (   SELECT    E.eid
          FROM      Emp E, Emp M, Works W, Dept D
          WHERE     E.eid = W.eid
          AND       W.did = D.did
          AND       D.managerid = M.eid
          AND       E.salary > M.salary)
```

5. This constraint can be satisfied by creating a trigger that increases a manager's salary to be equal to the employee who received the raise, if the manager's salary is less than the employee's new salary.

```
CREATE TRIGGER GiveRaise AFTER UPDATE ON Emp
          WHEN      old.salary < new.salary
          FOR EACH ROW
          BEGIN
                    UPDATE    Emp M
                    SET       M.Salary = new.salary
                    WHERE     M.salary < new.salary
                    AND       M.eid IN (SELECT   D.mangerid
                                        FROM     Emp E, Works W, Dept D
                                        WHERE    E.eid = new.eid
                                        AND      E.eid = W.eid
                                        AND      W.did = D.did);
          END
```

6. This constraint can be satisfied by extending the trigger in the previous question. We must add an `UPDATE` command to increase the budget by the amount of the raise if the budget is less than the sum of all employee salaries.

```
CREATE TRIGGER GiveRaise AFTER UPDATE ON Emp
```

```
WHEN    old.salary < new.salary
FOR EACH ROW
DECLARE
        _raise REAL;
BEGIN
        _raise := new.salary - old.salary;
        UPDATE  Emp M
        SET     M.Salary = new.salary
        WHERE   M.salary < new.salary
        AND     M.eid IN (SELECT  D.mangerid
                          FROM    Emp E, Works W, Dept D
                          WHERE   E.eid = new.eid
                          AND     E.eid = W.eid
                          AND     W.did = D.did);
        UPDATE  Dept D
        SET     D.budget = D.budget + _raise
        WHERE   D.did IN (SELECT  W.did
                          FROM    Emp E, Works W, Dept D
                          WHERE   E.eid = new.eid
                          AND     E.eid = W.eid
                          AND     D.did = W.did
                          AND     D.budget <
                          (SELECT  Sum(E2.salary)
                          FROM    Emp E2, Works W2
                          WHERE   E2.eid = W2.eid
                          AND     W2.dept = D.did));
        END
```

6

DATABASE APPLICATION DEVELOPMENT

Exercise 6.1 Briefly answer the following questions.

1. Explain the following terms: Cursor, Embedded SQL, JDBC, SQLJ, stored procedure.
2. What are the differences between JDBC and SQLJ? Why do they both exist?
3. Explain the term *stored procedure*, and give examples why stored procedures are useful.

Answer 6.1 The answers are given below:

1. A **cursor** enables individual row access of a relation by positioning itself at a row and reading its contents. **Embedded SQL** refers to the usage of SQL commands within a host program. **JDBC** stands for Java DataBase Connectivity and is an interface that allows a Java program to easily connect to any database system. **SQLJ** is a tool that allows SQL to be embedded directly into a Java program. A **stored procedure** is program that runs on the database server and can be called with a single SQL statement.
2. SQLJ provides embedded SQL statements. These SQL statements are static in nature and thus are preprocessed and precompiled. For instance, syntax checking and schema checking are done at compile time. JDBC allows dynamic queries that are checked at runtime. SQLJ is easier to use than JDBC and is often a better option for static queries. For dynamic queries, JDBC must still be used.
3. **Stored procedures** are programs that run on the database server and can be called with a single SQL statement. They are useful in situations where the processing should be done on the server side rather than the client side. Also, since the procedures are centralized to the server, code writing and maintenance is simplified, because the client programs do not have to duplicate the application logic. Stored procedures can also be used to reduce network communication; the results of a stored procedure can be analyzed and kept on the database server.

Exercise 6.2 Explain how the following steps are performed in JDBC:

1. Connect to a data source.
2. Start, commit, and abort transactions.
3. Call a stored procedure.

How are these steps performed in SQLJ?

Answer 6.2 The answers are given below:

1. Connecting to a data source in JDBC involves the creation of a *Connection* object. Parameters for the connection are specified using a *JDBC URL* that contains things like the network address of the database server and the username and password for connecting.
   SQLJ makes calls to the same JDBC drver for connecting to a data source and uses the same type of JDBC URL.
2. Each connection can specify how to handle transactions. If the *autocommit* flag is set, each SQL statement is treated as a separate transaction. If the flag is turned off, there is a *commit()* function call that will actually commit the transaction.
   The autocommit flag can also be set in SQLJ. If the flag is not set, transactions are committed by passing a *COMMIT* SQL statement to the DBMS.
3. Stored procedures are called from JDBC using the *CallableStatement* class with the SQL command {*CALL StoredProcedureName*}.
   SQLJ also uses *CALL StoredProcedureName* to execute stored prodecures at the DBMS.

Exercise 6.3 Compare exception handling and handling of warnings in embedded SQL, dynamic SQL, JDBC, and SQLJ.

Answer 6.3 The answers are given below:

- Embedded SQL: The SQLSTATE variable is used to check for errors after each Embedded SQL statement is executed. If an error has occurred, program control is transferred to a separate statement. This is done during the precompilation step for static queries.
- Dynamic SQL: For dynamic SQL, the SQL statement can change at runtime and thus the error handling must also occur at runtime.
- JDBC: In JDBC, programmers can use the *try ... catch* syntax to handle exceptions of type *SQLException*. The *SQLWarning* class is used for problems not as severe as errors. They are not caught in the *try ... catch* statement and must be checked independently with a *getWarnings()* function call.

- SQLJ: SQLJ uses the same mechanisms as JDBC to catch error and warnings.

Exercise 6.4 Answer the following questions.

1. Why do we need a precompiler to translate embedded SQL and SQLJ? Why do we not need a precompiler for JDBC?
2. SQLJ and embedded SQL use variables in the host language to pass parameters to SQL queries, whereas JDBC uses placeholders marked with a '?'. Explain the difference, and why the different mechanisms are needed.

Answer 6.4 The answers are given below:

1. Since embedded SQL and SQLJ use static queries, they allow compile-time syntax checking and schema validation. SQL statements of these types are written using a simplified syntax; the precompiler will translate that syntax into the equivalent JDBC calls. With pure JDBC, the SQL statements are fully dynamic and a preprocessor cannot be used since the SQL may change at run time.
2. With SQLJ and embedded SQL, host variables are bound to the SQL queries and cannot be swapped for different variables. This supports the precompilation processing step for static queries. With JDBC, the ? placeholders can be filled with any variable chosen at runtime. The ability to change variables allows for more dynamic queries in the program.

Exercise 6.5 A dynamic web site generates HTML pages from information stored in a database. Whenever a page is requested, is it dynamically assembled from static data and data in a database, resulting in a database access. Connecting to the database is usually a time-consuming process, since resources need to be allocated, and the user needs to be authenticated. Therefore, **connection pooling**—setting up a pool of persistent database connections and then reusing them for different requests can significantly improve the performance of database-backed websites. Since servlets can keep information beyond single requests, we can create a connection pool, and allocate resources from it to new requests.

Write a connection pool class that provides the following methods:

- Create the pool with a specified number of open connections to the database system.
- Obtain an open connection from the pool.
- Release a connection to the pool.
- Destroy the pool and close all connections.

Answer 6.5 The answer for this exercise is available online for instructors. To find out how to get access to instructor's material, visit the book homepage at `http://www.cs.wisc.edu/~dbbook`.

7

INTERNET APPLICATIONS

Exercise 7.1 Briefly answer the following questions:

1. Explain the following terms and describe what they are used for: HTML, URL, XML, Java, JSP, XSL, XSLT, servlet, cookie, HTTP, CSS, DTD.
2. What is CGI? Why was CGI introduced? What are the disadvantages of an architecture using CGI scripts?
3. What is the difference between a webserver and an application server? What funcionality do typical application servers provide?
4. When is an XML document well-formed? When is an XML document valid?

Answer 7.1 The answers are as follows.

1. *HTTP* (HyperText Transfer Protocol) is the communication protocol used to connect clients with servers over the Internet. *URL* (Universal Resource Locator) is a string that uniquely identifies an internet address. *HTML* (HyperText Markup Language) is a simple language used to enhance regular text by including special tags. *CSS* (Cascading Style Sheets) are used to define how to display HTML documents. *XML* (Extensible Markup Language) allows users to define their own markup tags in a document. *XSL* (Extensible Style Language) can be used to describe how an XML document should be displayed. *XSLT* (XML Transformation Language) is a language that can transform input XML into differently structured XML. A *DTD* (Document Type Declaration) is a grammar that describes how to use new tags in an XML document. *Java* is cross-platform interpreted programming language. *Servlets* are pieces of Java code that run on the middle tier or server layers and be used for any functionality that Java provides. *JSP* (JavaServer Pages) are HTML pages with embedded servlet code.. *Cookies* are a simple way to store persistent data at the client level.

Figure 7.1 Solution to Exercise 7.2 (d)

.

2. *CGI* (Common Gateway Interface) specifies how the web server communicates other programs on the server. CGI programs are used to pass HTML form data to other programs that process that data. Each page request will create a new process on the server, which is a performance issue when requests are scaled up.

3. A web server handles the interaction with the client's web browser. Application servers are used to maintain a pool of processes for handling requests. Typically, they are the middleware tier between the web server and the data sources such as database systems. Application servers eliminate the problems with process-creation overload and can also provide extra functionality like abstracting away heterogeneous data sources and maintaining session state information.

4. An XML document is *valid* if it has an associated DTD and the document follows the rules of the DTD. An XML document is *well-formed* if it follows three guidelines: (1) it starts with an XML declaration, (2) it contains a root element that contains all other elements and (3) all elements are properly nested.

Exercise 7.2 Briefly answer the following questions about the HTTP protocol:

1. What is a communication protocol?

2. What is the structure of an HTTP request message? What is the structure of an HTTP response message? Why do HTTP messages carry a version field?

3. What is a stateless protocol? Why was HTTP designed to be stateless?

4. Show the HTTP request message generated when you request the home page of this book (`http://www.cs.wisc.edu/~dbbook`). Show the HTTP response message that the server generates for that page.

Answer 7.2 The answers are as follows.

1. A *communication protocol* defines the message structure for communication between two entities.

2. HTTP request messages have the following structure:

   - The Request line that contains the HTTP method field, the URI field, and the version field
   - The User Agent line that specifies the type of client
   - The Accept line that indicates what types of files the client will accept

HTTP response messages have three parts:

- The Status line that contains the HTTP version number, a status code and the associated message text
- Several header lines with information such as the creation date, the content length, and the content type
- Actual message body

The version field is reserved for future versions of the protocol that may enforce compatibility between the client and server HTTP versions.

3. A *stateless protocol* means that each message is self-contained (i.e. no information is maintained between messages). HTTP was designed to be stateless for simplicity of protocol design. Application layers above HTTP can implement more complicated functionality.

4. The HTTP request message looks as follows:

```
GET / dbbook/ HTTP/1.1
Host:  www.cs.wisc.edu
Connection:  close
Accept-Encoding:  gzip
Accept:  image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language:  en-us
User-Agent:  Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2)
```

The HTTP response message looks as follows:

```
HTTP/1.1 200 OK
Date:  Sun, 30 Nov 2003 21:09:41 GMT
Server:  Apache/1.3.28 (Unix) mod_perl/1.28 PHP/4.3.2
mod_ssl/2.8.15 OpenSSL/0.9.6k
Last-Modified:  Wed, 05 Nov 2003 05:13:32 GMT
Accept-Ranges:  bytes
Content-Length:  5434
Connection:  close Content-Type:  text/html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- saved from url=(0031)http://www.cs.wisc.edu/ dbbook/ -->
<HTML> ...  </HTML>
```

Exercise 7.3 In this exercise, you are asked to write the functionality of a generic shopping basket; you will use this in several subsequent project exercises. Write a set of JSP pages that displays a shopping basket of items and allows users to add, remove, and change the quantity of items. To do this, use a cookie storage scheme that stores the following information:

- The UserId of the user who owns the shopping basket.

- The number of products stored in the shopping basket.
- A product id and a quantity for each product.

When manipulating cookies, remember to set the `Expires` property such that the cookie can persist for a session or indefinitely. Experiment with cookies using JSP and make sure you know how to retrieve, set values, and delete the cookie.

You need to create five JSP pages to make your prototype complete:

- **Index Page** (`index.jsp`): This is the main entry point. It has a link that directs the user to the Products page so they can start shopping.
- **Products Page** (`products.jsp`): Shows a listing of all products in the database with their descriptions and prices. This is the main page where the user fills out the shopping basket. Each listed product should have a button next to it, which adds it to the shopping basket. (If the item is already in the shopping basket, it increments the quantity by one.) There should also be a counter to show the total number of items currently in the shopping basket. Note that if a user has a quantity of five of a single item in the shopping basket, the counter should indicate a total quantity of five. The page also contains a button that directs the user to the Cart page.
- **Cart Page** (`cart.jsp`): Shows a listing of all items in the shopping basket cookie. The listing for each item should include the product name, price, a text box for the quantity (the user can change the quantity of items here), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to the Products page), a second button to update the cookie with the altered quantities from the text boxes, and a third button to place or confirm the order, which directs the user to the Confirm page.
- **Confirm Page** (`confirm.jsp`): Lists the final order. There are two buttons on this page. One button cancels the order and the other submits the completed order. The cancel button just deletes the cookie and returns the user to the Index page. The submit button updates the database with the new order, deletes the cookie, and returns the user to the Index page.

Exercise 7.4 In the previous exercise, replace the page `products.jsp` with the following *search page* `search.jsp`. This page allows users to search products by name or description. There should be both a text box for the search text and radio buttons to allow the user to choose between search-by-name and search-by-description (as well as a submit button to retrieve the results). The page that handles search results should be modeled after `products.jsp` (as described in the previous exercise) and be called `products.jsp`. It should retrieve all records where the search text is a substring of

the name or description (as chosen by the user). To integrate this with the previous exercise, simply replace all the links to `products.jsp` with `search.jsp`.

Exercise 7.5 Write a simple authentication mechanism (without using encrypted transfer of passwords, for simplicity). We say a user is authenticated if she has provided a valid username-password combination to the system; otherwise, we say the user is not authenticated. Assume for simplicity that you have a database schema that stores only a customer id and a password:

Passwords(cid: `integer`, username: `string`, password: `string`)

1. How and where are you going to track when a user is 'logged on' to the system?
2. Design a page that allows a registered user to log on to the system.
3. Design a page header that checks whether the user visiting this page is logged in.

Exercise 7.6 (Due to Jeff Derstadt) TechnoBooks.com is in the process of reorganizing its website. A major issue is how to efficiently handle a large number of search results. In a human interaction study, it found that modem users typically like to view 20 search results at a time, and it would like to program this logic into the system. Queries that return batches of sorted results are called *top N queries*. (See Section 23 for a discussion of database support for top N queries.) For example, results 1-20 are returned, then results 21-40, then 41-60, and so on. Different techniques are used for performing top N queries and TechnoBooks.com would like you to implement two of them.

Infrastructure: Create a database with a table called Books and populate it with some books, using the format that follows. This gives you 111 books in your database with a title of AAA, BBB, CCC, DDD, or EEE, but the keys are not sequential for books with the same title.

Books(*bookid:* `INTEGER`, *title:* `CHAR(80)`, *author:* `CHAR(80)`, *price:* `REAL`)

```
For i = 1 to 111 {
        Insert the tuple (i, "AAA", "AAA Author", 5.99)
        i = i + 1
        Insert the tuple (i, "BBB", "BBB Author", 5.99)
        i = i + 1
        Insert the tuple (i, "CCC", "CCC Author", 5.99)
        i = i + 1
        Insert the tuple (i, "DDD", "DDD Author", 5.99)
        i = i + 1
        Insert the tuple (i, "EEE", "EEE Author", 5.99)
}
```

Placeholder Technique: The simplest approach to top N queries is to store a placeholder for the first and last result tuples, and then perform the same query. When the new query results are returned, you can iterate to the placeholders and return the previous or next 20 results.

Tuples Shown	Lower Placeholder	Previous Set	Upper Placeholder	Next Set
1-20	1	None	20	21-40
21-40	21	1-20	40	41-60
41-60	41	21-40	60	61-80

Write a webpage in JSP that displays the contents of the Books table, sorted by the Title and BookId, and showing the results 20 at a time. There should be a link (where appropriate) to get the previous 20 results or the next 20 results. To do this, you can encode the placeholders in the Previous or Next Links as follows. Assume that you are displaying records 21–40. Then the previous link is `display.jsp?lower=21` and the next link is `display.jsp?upper=40`.

You should not display a previous link when there are no previous results; nor should you show a Next link if there are no more results. When your page is called again to get another batch of results, you can perform the same query to get all the records, iterate through the result set until you are at the proper starting point, then display 20 more results.

What are the advantages and disadvantages of this technique?

Query Constraints Technique: A second technique for performing top N queries is to push boundary constraints into the query (in the `WHERE` clause) so that the query returns only results that have not yet been displayed. Although this changes the query, fewer results are returned and it saves the cost of iterating up to the boundary. For example, consider the following table, sorted by (title, primary key).

Batch	Result Number	Title	Primary Key
1	1	AAA	105
1	2	BBB	13
1	3	CCC	48
1	4	DDD	52
1	5	DDD	101
2	6	DDD	121
2	7	EEE	19
2	8	EEE	68
2	9	FFF	2
2	10	FFF	33
3	11	FFF	58
3	12	FFF	59
3	13	GGG	93
3	14	HHH	132
3	15	HHH	135

In batch 1, rows 1 through 5 are displayed, in batch 2 rows 6 through 10 are displayed, and so on. Using the placeholder technique, all 15 results would be returned for each batch. Using the constraint technique, batch 1 displays results 1-5 but returns results 1-15, batch 2 will display results 6-10 but returns only results 6-15, and batch 3 will display results 11-15 but return only results 11-15.

The constraint can be pushed into the query because of the sorting of this table. Consider the following query for batch 2 (displaying results 6-10):

```
EXEC SQL SELECT B.Title
FROM     Books B
WHERE    (B.Title = 'DDD' AND B.BookId > 101) OR (B.Title > 'DDD')
ORDER BY B.Title, B.BookId
```

This query first selects all books with the title 'DDD,' but with a primary key that is greater than that of record 5 (record 5 has a primary key of 101). This returns record 6. Also, any book that has a title after 'DDD' alphabetically is returned. You can then display the first five results.

The following information needs to be retained to have Previous and Next buttons that return more results:

- **Previous:** The title of the *first* record in the previous set, and the primary key of the *first* record in the previous set.
- **Next:** The title of the *first* record in the next set; the primary key of the *first* record in the next set.

These four pieces of information can be encoded into the Previous and Next buttons as in the previous part. Using your database table from the first part, write a JavaServer Page that displays the book information 20 records at a time. The page should include *Previous* and *Next* buttons to show the previous or next record set if there is one. Use the constraint query to get the Previous and Next record sets.

8

OVERVIEW OF STORAGE AND INDEXING

Exercise 8.1 Answer the following questions about data on external storage in a DBMS:

1. Why does a DBMS store data on external storage?
2. Why are I/O costs important in a DBMS?
3. What is a record id? Given a record's id, how many I/Os are needed to fetch it into main memory?
4. What is the role of the buffer manager in a DBMS? What is the role of the disk space manager? How do these layers interact with the file and access methods layer?

Answer 8.1 The answer to each question is given below.

1. A DBMS stores data on external storage because the quantity of data is vast, and must persist across program executions.
2. I/O costs are of primary important to a DBMS because these costs typically dominate the time it takes to run most database operations. Optimizing the amount of I/O's for an operation can result in a substantial increase in speed in the time it takes to run that operation.
3. A record id, or rid for short, is a unique identifier for a particular record in a set of records. An rid has the property that we can identify the disk address of the page containing the record by using the rid. The number of I/O's required to read a record, given a rid, is therefore 1 I/O.
4. In a DBMS, the buffer manager reads data from persistent storage into memory as well as writes data from memory into persistent storage. The disk space manager manages the available physical storage space of data for the DBMS. When the file

and access methods layer needs to process a page, it asks the buffer manager to fetch the page and put it into memory if it is not all ready in memory. When the files and access methods layer needs additional space to hold new records in a file, it asks the disk space manager to allocate an additional disk page.

Exercise 8.2 Answer the following questions about files and indexes:

1. What operations are supported by the file of records abstraction?
2. What is an index on a file of records? What is a search key for an index? Why do we need indexes?
3. What alternatives are available for the data entries in an index?
4. What is the difference between a primary index and a secondary index? What is a duplicate data entry in an index? Can a primary index contain duplicates?
5. What is the difference between a clustered index and an unclustered index? If an index contains data records as 'data entries,' can it be unclustered?
6. How many clustered indexes can you create on a file? Would you always create at least one clustered index for a file?
7. Consider Alternatives (1), (2) and (3) for 'data entries' in an index, as discussed in Section 8.2. Are all of them suitable for secondary indexes? Explain.

Answer 8.2 The answer to each question is given below.

1. The file of records abstraction supports file creation and deletion, record creation and deletion, and scans of of individual records in a file one at a time.
2. An index is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. A search key for an index is the fields stored in the index that we can search on to efficiently retrieve all records satisfy the search conditions. Without indexes, every search would to a DBMS would require a scan of all records and be extremely costly.
3. The three main alternatives for what to store as a data entry in an index are as follows:
   (a) A data entry k^* is an actual data record (with search key value k).
   (b) A data entry is a $\langle\ k,\ rid\ \rangle$ pair, where *rid* is the record id of a data record with search key value k.
   (c) A data entry is a $\langle\ k,\ rid\text{-}list\ \rangle$ pair, where *rid-list* is a list of record ids of data records with search key value k.

4. A primary index is an index on a set of fields that includes the unique primary key for the field and is guaranteed not to contain duplicates. A secondary index is an index that is not a primary index and may have duplicates. Two entries are said to be duplicates if they have the same value for the search key field associated with the index.

5. A clustered index is one in which the ordering of data entries is the same as the ordering of data records. We can have at most one clustered index on a data file. An unclustered index is an index that is not clustered. We can have several unclustered indexes on a data file. If the index contains data records as 'data entries', it means the index uses Alternative (1). By definition of clustered indexes, the index is clustered.

6. At most one, because we want to avoid replicating data records. Sometimes, we may not create any clustered indexes because no query requires a clustered index for adequate performance, and clustered indexes are more expensive to maintain than unclustered indexes.

7. No. An index using alternative (1) has actual data records as data entries. It must be a primary index and has no duplicates. It is not suitable for a secondary index because we do not want to replicate data records.

Exercise 8.3 Consider a relation stored as a randomly ordered file for which the only index is an unclustered index on a field called *sal*. If you want to retrieve all records with $sal > 20$, is using the index always the best alternative? Explain.

Answer 8.3 No. In this case, the index is unclustered, each qualifying data entry could contain an rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range query. In this situation, using index is actually worse than file scan.

Exercise 8.4 Consider the instance of the Students relation shown in Figure 8.1, sorted by *age*: For the purposes of this question, assume that these tuples are stored in a sorted file in the order shown; the first tuple is on page 1 the second tuple is also on page 1; and so on. Each page can store up to three data records; so the fourth tuple is on page 2.

Explain what the data entries in each of the following indexes contain. If the order of entries is significant, say so and explain why. If such an index cannot be constructed, say so and explain why.

1. An unclustered index on *age* using Alternative (1).

2. An unclustered index on *age* using Alternative (2).

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8

Figure 8.1 An Instance of the Students Relation, Sorted by *age*

3. An unclustered index on *age* using Alternative (3).
4. A clustered index on *age* using Alternative (1).
5. A clustered index on *age* using Alternative (2).
6. A clustered index on *age* using Alternative (3).
7. An unclustered index on *gpa* using Alternative (1).
8. An unclustered index on *gpa* using Alternative (2).
9. An unclustered index on *gpa* using Alternative (3).
10. A clustered index on *gpa* using Alternative (1).
11. A clustered index on *gpa* using Alternative (2).
12. A clustered index on *gpa* using Alternative (3).

Answer 8.4 The answer to each question is given below. For Alternative (2), the notation $\langle A, (B,C) \rangle$ is used where A is the search key for the entry and (B,C) is the *rid* for data entry, with B being the page number of the entry and C being the location on page B of the entry. For Alternative (3), the notation is the same, but with the possibility of additional *rid*'s listed.

1. Contradiction. Cannot build unclustered index using Alternative (1) since method is inherently clustered.
2. $\langle$ *11*, (*1,1*) $\rangle$, $\langle$ *12*, (*1,2*) $\rangle$, $\langle$ *18*, (*1,3*) $\rangle$, $\langle$ *19*, (*2,1*) $\rangle$, $\langle$ *19*, (*2,2*) $\rangle$. The order of entries is not significant.
3. $\langle$ *11*, (*1,1*) $\rangle$, $\langle$ *12*, (*1,2*) $\rangle$, $\langle$ *18*, (*1,3*) $\rangle$, $\langle$ *19*, (*2,1*), (*2,2*) $\rangle$, The order of entries is not significant.
4. 11, 19. The order of entries is significant since the order of the entries is the same as the order of data record.

5. ⟨ *11*, (*1,1*) ⟩, ⟨ *19*, (*2,1*) ⟩. The order of entries is significant since the order of the entries is the same as the order of data record.

6. ⟨ *11*, (*1,1*) ⟩, ⟨ *19*, (*2,1*), (*2,2*) ⟩. The order of entries is significant since the order of the entries is the same as the order of data record.

7. Contradiction. Cannot build unclustered index using Alternative (1) since method is inherently clustered.

8. ⟨ *1.8*, (*1,1*) ⟩, ⟨ *2.0*, (*1,2*) ⟩, ⟨ *3.2*, (*2,1*) ⟩, ⟨ *3.4*, (*1,3*) ⟩, ⟨ *3.8*, (*2,2*) ⟩. The order of entries is not significant.

9. ⟨ *1.8*, (*1,1*) ⟩, ⟨ *2.0*, (*1,2*) ⟩, ⟨ *3.2*, (*2,1*) ⟩, ⟨ *3.4*, (*1,3*) ⟩, ⟨ *3.8*, (*2,2*) ⟩. The order of entries is not significant.

10. Alternative (1) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* as previously defined.

11. Alternative (2) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* as previously defined.

12. Alternative (3) cannot be used to build a clustered index on *gpa* because the records in the file are not sorted in order of *gpa*. Only if the entries in (*1,3*) and (*2,1*) were switched would this possible, but then the data would no longer be sorted on *age* previously defined.

Exercise 8.5 Explain the difference between Hash indexes and B+-tree indexes. In particular, discuss how equality and range searches work, using an example.

Answer 8.5 A Hash index is constructed by using a hashing function that quickly maps an search key value to a specific location in an array-like list of elements called buckets. The buckets are often constructed such that there are more bucket locations than there are possible search key values, and the hashing function is chosen so that it is not often that two search key values hash to the same bucket. A B+-tree index is constructed by sorting the data on the search key and maintaining a hierarchical search data structure that directs searches to the correct page of data entries.

Insertions and deletions in a hash based index are relatively simple. If two search values hash to the same bucket, called a collision, a linked list is formed connecting multiple records in a single bucket. In the case that too many of these collisions occur, the number of buckets is increased. Alternatively, maintaining a B+-tree's hierarchical search data structure is considered more costly since it must be updated whenever there

are insertions and deletions in the data set. In general, most insertions and deletions will not modify the data structure severely, but every once in awhile large portions of the tree may need to be rewritten when they become over-filled or under-filled with data entries.

Hash indexes are especially good at equality searches because they allow a record look up very quickly with an average cost of 1.2 I/Os. B+-tree indexes, on the other hand, have a cost of 3-4 I/Os per individual record lookup. Assume we have the employee relation with primary key *eid* and 10,000 records total. Looking up all the records individually would cost 12,000 I/Os for Hash indexes, but 30,000-40,000 I/Os for B+-tree indexes.

For range queries, hash indexes perform terribly since they could conceivably read as many pages as there are records since the data is not sorted in any clear grouping or set. On the other hand, B+-tree indexes have a cost of 3-4 I/Os plus the number of qualifying pages or tuples, for clustered or unclustered B+-trees respectively. Assume we have the employees example again with 10,000 records and 10 records per page. Also assume that there is an index on *sal* and query of *age* ¿ 20,000, such that there are 5,000 qualifying tuples. The hash index could cost as much as 100,000 I/Os since every page could be read for every record. It is not clear with a hash index how we even go about searching for every possible number greater than 20,000 since decimals could be used. An unclustered B+-tree index would have a cost of 5,004 I/Os, while a clustered B+-tree index would have a cost of 504 I/Os. It helps to have the index clustered whenever possible.

Exercise 8.6 Fill in the I/O costs in Figure 8.2.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file					
Sorted file					
Clustered file					
Unclustered tree index					
Unclustered hash index					

Figure 8.2 I/O Cost Comparison

Answer 8.6 The answer to the question is given in Figure 8.3. We use B to denote the number of data pages total, R to denote the number of records per page, and D to denote the average time to read or write a page.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap file	*BD*	*0.5BD*	*BD*	*2D*	*Search+* D
Sorted file	*BD*	$Dlog_2B$	$Dlog_2B$+# *matching pages*	*Search+* BD	*Search+* BD
Clustered file	*1.5BD*	$Dlog_F1.5B$	$Dlog_FB$+# *matching pages*	*Search+ D*	*Search+ D*
Unclustered tree index	*BD(R+ 0.15)*	*D(1+* $log_F0.15B$*)*	*D(*$log_F0.15B$+# *matching records)*	*D(3+* $log_F0.15B$*)*	*Search+ 2D*
Unclustered hash index	*BD(R+ 0.125)*	*2D*	*BD*	*4D*	*Search+ 2D*

Figure 8.3 I/O Cost Comparison

Exercise 8.7 If you were about to create an index on a relation, what considerations would guide your choice? Discuss:

1. The choice of primary index.
2. Clustered versus unclustered indexes.
3. Hash versus tree indexes.
4. The use of a sorted file rather than a tree-based index.
5. Choice of search key for the index. What is a composite search key, and what considerations are made in choosing composite search keys? What are index-only plans, and what is the influence of potential index-only evaluation plans on the choice of search key for an index?

Answer 8.7 The answer to each question is given below.

1. The choice of the primary key is made based on the semantics of the data. If we need to retrieve records based on the value of the primary key, as is likely, we should build an index using this as the search key. If we need to retrieve records based on the values of fields that do not constitute the primary key, we build (by definition) a secondary index using (the combination of) these fields as the search key.
2. A clustered index offers much better range query performance, but essentially the same equality search performance (modulo duplicates) as an unclustered index.

Further, a clustered index is typically more expensive to maintain than an unclustered index. Therefore, we should make an index be clustered only if range queries are important on its search key. At most one of the indexes on a relation can be clustered, and if range queries are anticipated on more than one combination of fields, we have to choose the combination that is most important and make that be the search key of the clustered index.

3. If it is likely that ranged queries are going to be performed often, then we should use a B+-tree on the index for the relation since hash indexes cannot perform range queries. If it is more likely that we are only going to perform equality queries, for example the case of social security numbers, than hash indexes are the best choice since they allow for the faster retrieval than B+-trees by 2-3 I/Os per request.

4. First of all, both sorted files and tree-based indexes offer fast searches. Insertions and deletions, though, are much faster for tree-based indexes than sorted files. On the other hand scans and range searches with many matches are much faster for sorted files than tree-based indexes. Therefore, if we have read-only data that is not going to be modified often, it is better to go with a sorted file, whereas if we have data that we intend to modify often, then we should go with a tree-based index.

5. A composite search key is a key that contains several fields. A composite search key can support a broader range as well as increase the possibility for an index-only plan, but are more costly to maintain and store. An index-only plan is query evaluation plan where we only need to access the indexes for the data records, and not the data records themselves, in order to answer the query. Obviously, index-only plans are much faster than regular plans since it does not require reading of the data records. If it is likely that we are going to performing certain operations repeatly that only require accessing one field, for example the average value of a field, it would be an advantage to create a search key on this field since we could then accomplish it with an index-only plan.

Exercise 8.8 Consider a delete specified using an equality condition. For each of the five file organizations, what is the cost if no record qualifies? What is the cost if the condition is not on a key?

Answer 8.8 If the search key is not a candidate key, there may be several qualifying records. In a heap file, this means we have to search the entire file to be sure that we've found all qualifying records; the cost is $B(D + RC)$. In a sorted file, we find the first record (cost is that of equality search; $Dlog_2B + Clog_2R$) and then retrieve and delete successive records until the key value changes. The cost of the deletions is C per deleted record, and D per page containing such a record. In a hashed file, we hash to find the appropriate bucket (cost H), then retrieve the page (cost D; let's assume

no overflow pages), then write the page back if we find a qualifying record and delete it (cost D).

If no record qualifies, in a heap file, we have to search the entire file. So the cost is $B(D + RC)$. In a sorted file, even if no record qualifies, we have to do equality search to verify that no qualifying record exists. So the cost is the same as equality search, $Dlog_2B + Clog_2R$. In a hashed file, if no record qualifies, assuming no overflow page, we compute the hash value to find the bucket that would contain such a record (cost is H), bring that page in (cost is D), and search the entire page to verify that the record is not there (cost is RC). So the total cost is $H + D + RC$.

In all three file organizations, if the condition is not on the search key we have to search the entire file. There is an additional cost of C for each record that is deleted, and an additional D for each page containing such a record.

Exercise 8.9 What main conclusions can you draw from the discussion of the five basic file organizations discussed in Section 8.4? Which of the five organizations would you choose for a file where the most frequent operations are as follows?

1. Search for records based on a range of field values.
2. Perform inserts and scans, where the order of records does not matter.
3. Search for a record based on a particular field value.

Answer 8.9 The main conclusion about the five file organizations is that all five have their own advantages and disadvantages. No one file organization is uniformly superior in all situations. The choice of appropriate structures for a given data set can have a significant impact upon performance. An unordered file is best if only full file scans are desired. A hash indexed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired and the data is static; a clustered B+ tree is best if range selections are important and the data is dynamic. An unclustered B+ tree index is useful for selections over small ranges, especially if we need to cluster on another search key to support some common query.

1. Using these fields as the search key, we would choose a sorted file organization or a clustered B+ tree depending on whether the data is static or not.
2. Heap file would be the best fit in this situation.
3. Using this particular field as the search key, choosing a hash indexed file would be the best.

Exercise 8.10 Consider the following relation:

Emp(*eid:* integer, *sal:* integer, *age:* real, *did:* integer)

There is a clustered index on *eid* and an unclustered index on *age*.

1. How would you use the indexes to enforce the constraint that *eid* is a key?
2. Give an example of an update that is *definitely speeded up* because of the available indexes. (English description is sufficient.)
3. Give an example of an update that is *definitely slowed down* because of the indexes. (English description is sufficient.)
4. Can you give an example of an update that is neither speeded up nor slowed down by the indexes?

Answer 8.10 The answer to each question is given below.

1. To enforce the constraint that *eid* is a key, all we need to do is make the clustered index on *eid unique* and *dense*. That is, there is at least one data entry for each *eid* value that appears in an Emp record (because the index is dense). Further, there should be exactly one data entry for each such *eid* value (because the index is unique), and this can be enforced on inserts and updates.
2. If we want to change the salaries of employees whose *eid*'s are in a particular range, it would be sped up by the index on *eid*. Since we could access the records that we want much quicker and we wouldn't have to change any of the indexes.
3. If we were to add 1 to the ages of all employees then we would be slowed down, since we would have to update the index on age.
4. If we were to change the *sal* of those employees with a particular *did* then no advantage would result from the given indexes.

Exercise 8.11 Consider the following relations:

Emp(*eid:* integer, *ename:* varchar, *sal:* integer, *age:* integer, *did:* integer)
Dept(*did:* integer, *budget:* integer, *floor:* integer, *mgr_eid:* integer)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.

For each of the following queries, which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.

1. Query: *Print ename, age, and sal for all employees.*
   (a) Clustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (b) Unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (c) Clustered B+ tree index on $\langle ename, age, sal \rangle$ fields of Emp.
   (d) Unclustered hash index on $\langle eid, did \rangle$ fields of Emp.
   (e) No index.
2. Query: *Find the dids of departments that are on the 10th floor and have a budget of less than $15,000.*
   (a) Clustered hash index on the $floor$ field of Dept.
   (b) Unclustered hash index on the $floor$ field of Dept.
   (c) Clustered B+ tree index on $\langle floor, budget \rangle$ fields of Dept.
   (d) Clustered B+ tree index on the *budget* field of Dept.
   (e) No index.

Answer 8.11 The answer to each question is given below.

1. We should create an unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp (b) since then we could do an index only scan. If our system does not include index only plans then we shouldn't create an index for this query (e). Since this query requires us to access all the Emp records, an index won't help us any, and so should we access the records using a filescan.
2. We should create a clustered dense B+ tree index (c) on $\langle floor, budget \rangle$ fields of Dept, since the records would be ordered on these fields then. So when executing this query, the first record with $floor = 10$ must be retrieved, and then the other records with $floor = 10$ can be read in order of budget. Note that this plan, which is the best for this query, is not an index-only plan (must look up dids).

9

STORING DATA: DISKS AND FILES

Exercise 9.1 What is the most important difference between a disk and a tape?

Answer 9.1 *Tapes* are sequential devices that do not support direct access to a desired page. We must essentially step through all pages in order. *Disks* support direct access to a desired page.

Exercise 9.2 Explain the terms *seek time, rotational delay*, and *transfer time*.

Answer 9.2 They are all used to describe (different parts of) the cost to access a disk page.

1. *Seek time* is the time taken to move the disk heads to the track on which a desired block is located.
2. *Rotational delay* is the waiting time for the desired block to rotate under the disk head; it is the time required for half a rotation on average, and is usually less than the seek time.
3. *Transfer time* is the time to actually read or write the data in the block once the head is positioned, i.e., the time for the disk to rotate over the block.

Exercise 9.3 Both disks and main memory support direct access to any desired location (page). On average, main memory accesses are faster, of course. What is the other important difference between the two (from the perspective of the time required to access a desired page)?

Answer 9.3 The time to access a disk page is not constant. It depends on the location of the data. Accessing to some data might be much faster than to others. It is different for memory. The time to access memory is uniform for most computer systems.

Exercise 9.4 If you have a large file that is frequently scanned sequentially, explain how you would store the pages in the file on a disk.

Answer 9.4 The pages in the file should be stored 'sequentially' on a disk. We should put two 'logically' adjacent pages as close as possible. In decreasing order of closeness, they could be on the same track, the same cylinder, or an adjacent cylinder.

Exercise 9.5 Consider a disk with a sector size of 512 bytes, 2000 tracks per surface, 50 sectors per track, five double-sided platters, and average seek time of 10 msec.

1. What is the capacity of a track in bytes? What is the capacity of each surface? What is the capacity of the disk?
2. How many cylinders does the disk have?
3. Give examples of valid block sizes. Is 256 bytes a valid block size? 2048? 51200?
4. If the disk platters rotate at 5400 rpm (revolutions per minute), what is the maximum rotational delay?
5. If one track of data can be transferred per revolution, what is the transfer rate?

Answer 9.5 1.

$$bytes/track = bytes/sector \times sectors/track = 512 \times 50 = 25K$$

$$bytes/surface = bytes/track \times tracks/surface = 25K \times 2000 = 50,000K$$

$$bytes/disk = bytes/surface \times surfaces/disk = 50,000K \times 5 \times 2 = 500,000K$$

2. The number of cylinders is the same as the number of tracks on each platter, which is 2000.
3. The block size should be a multiple of the sector size. We can see that 256 is not a valid block size while 2048 is. 51200 is not a valid block size in this case because block size cannot exceed the size of a track, which is 25600 bytes.
4. If the disk platters rotate at 5400rpm, the time required for one complete rotation, which is the maximum rotational delay, is

$$\frac{1}{5400} \times 60 = 0.011 seconds$$

. The average rotational delay is half of the rotation time, 0.006 seconds.

5. The capacity of a track is 25K bytes. Since one track of data can be transferred per revolution, the data transfer rate is

$$\frac{25K}{0.011} = 2,250 Kbytes/second$$

Exercise 9.6 Consider again the disk specifications from Exercise 9.5, and suppose that a block size of 1024 bytes is chosen. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

1. How many records fit onto a block?
2. How many blocks are required to store the entire file? If the file is arranged sequentially on the disk, how many surfaces are needed?
3. How many records of 100 bytes each can be stored using this disk?
4. If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what page is stored on block 1 of track 1 on the next disk surface? How would your answer change if the disk were capable of reading and writing from all heads in parallel?
5. What time is required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how would your answer change if the disk were capable of reading/writing from all heads in parallel (and the data was arranged optimally)?
6. What is the time required to read a file containing 100,000 records of 100 bytes each in a random order? To read a record, the block containing the record has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

Answer 9.6 1. $1024/100 = 10$. We can have at most 10 records in a block.

2. There are 100,000 records all together, and each block holds 10 records. Thus, we need 10,000 blocks to store the file. One track has 25 blocks, one cylinder has 250 blocks. we need 10,000 blocks to store this file. So we will use more than one cylinders, that is, need 10 surfaces to store this file.
3. The capacity of the disk is 500,000K, which has 500,000 blocks. Each block has 10 records. Therefore, the disk can store no more than 5,000,000 records.
4. There are 25K bytes, or we can say, 25 blocks in each track. It is block 26 on block 1 of track 1 on the next disk surface.

   If the disk were capable of reading/writing from all heads in parallel, we can put the first 10 pages on the block 1 of track 1 of all 10 surfaces. Therefore, it is block 2 on block 1 of track 1 on the next disk surface.
5. A file containing 100,000 records of 100 bytes needs 40 cylinders or 400 tracks in this disk. The transfer time of one track of data is 0.011 seconds. Then it takes $400 \times 0.011 = 4.4 seconds$ to transfer 400 tracks.

   This access seeks the track 40 times. The seek time is $40 \times 0.01 = 0.4 seconds$. Therefore, total access time is $4.4 + 0.4 = 4.8 seconds$.

If the disk were capable of reading/writing from all heads in parallel, the disk can read 10 tracks at a time. The transfer time is 10 times less, which is 0.44 seconds. Thus total access time is $0.44 + 0.4 = 0.84 seconds$

6. For any block of data, $averageaccesstime = seektime + rotationaldelay + transfertime$.

$$seektime = 10msec$$

$$rotationaldelay = 6msec$$

$$transfertime = \frac{1K}{2,250K/sec} = 0.44msec$$

The average access time for a block of data would be 16.44 msec. For a file containing 100,000 records of 100 bytes, the total access time would be 164.4 seconds.

Exercise 9.7 Explain what the buffer manager must do to process a read request for a page. What happens if the requested page is in the pool but not pinned?

Answer 9.7 When a page is requested the buffer manager does the following:

1. The buffer pool is checked to see if it contains the requested page. If the page is in the pool, skip to step 2. If the page is not in the pool, it is brought in as follows:
   (a) A frame is chosen for replacement, using the replacement policy.
   (b) If the frame chosen for replacement is dirty, it is *flushed* (the page it contains is written out to disk).
   (c) The requested page is read into the frame chosen for replacement.
2. The requested page is *pinned* (the *pin_count* of the chosen frame is incremented) and its address is returned to the requester.

Note that if the page is not pinned, it could be removed from buffer pool even if it is actually needed in main memory. *Pinning* a page prevents it from being removed from the pool.

Exercise 9.8 When does a buffer manager write a page to disk?

Answer 9.8 If a page in the buffer pool is chosen to be replaced and this page is dirty, the buffer manager must write the page to the disk. This is also called flushing the page to the disk.

Sometimes the buffer manager can also force a page to disk for recovery-related purposes (intuitively, to ensure that the log records corresponding to a modified page are written to disk before the modified page itself is written to disk).

Exercise 9.9 What does it mean to say that a page is *pinned* in the buffer pool? Who is responsible for pinning pages? Who is responsible for unpinning pages?

Answer 9.9 1. *Pinning* a page means the *pin_count* of its frame is incremented. Pinning a page guarantees higher-level DBMS software that the page will not be removed from the buffer pool by the buffer manager. That is, another file page will not be read into the frame containing this page until it is unpinned by this requestor.

2. It is the buffer manager's responsibility to pin a page.

3. It is the responsibility of the requestor of that page to tell the buffer manager to unpin a page.

Exercise 9.10 When a page in the buffer pool is modified, how does the DBMS ensure that this change is propagated to the disk? (Explain the role of the buffer manager as well as the modifier of the page.)

Answer 9.10 The modifier of the page tells the buffer manager that the page is modified by setting the *dirty bit* of the page.

The buffer manager flushes the page to disk when necessary.

Exercise 9.11 What happens if a page is requested when all pages in the buffer pool are dirty?

Answer 9.11 If there are some unpinned pages, the buffer manager chooses one by using a *replacement policy*, flushes this page, and then replaces it with the requested page.

If there are no unpinned pages, the buffer manager has to wait until an unpinned page is available (or signal an error condition to the page requestor).

Exercise 9.12 What is *sequential flooding* of the buffer pool?

Answer 9.12 Some database operations (e.g., certain implementations of the *join* relational algebra operator) require repeated sequential scans of a relation. Suppose that there are 10 frames available in the buffer pool, and the file to be scanned has 11 or more pages (i.e., at least one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading in every page of the file! In this situation, called 'sequential flooding', LRU is the *worst* possible replacement strategy.

Exercise 9.13 Name an important capability of a DBMS buffer manager that is not supported by a typical operating system's buffer manager.

Answer 9.13 1. Pinning a page to prevent it from being replaced.

2. Ability to explicitly force a single page to disk.

Exercise 9.14 Explain the term *prefetching*. Why is it important?

Answer 9.14 Because most page references in a DBMS environment are with a known reference pattern, the buffer manager can anticipate the next several page requests and fetch the corresponding pages into memory before the pages are requested. This is *prefetching*.

Benefits include the following:

1. The pages are available in the buffer pool when they are requested.

2. Reading in a contiguous block of pages is much faster than reading the same pages at different times in response to distinct requests.

Exercise 9.15 Modern disks often have their own main memory caches, typically about 1 MB, and use this to prefetch pages. The rationale for this technique is the empirical observation that, if a disk page is requested by some (not necessarily database!) application, 80% of the time the next page is requested as well. So the disk gambles by reading ahead.

1. Give a nontechnical reason that a DBMS may not want to rely on prefetching controlled by the disk.

2. Explain the impact on the disk's cache of several queries running concurrently, each scanning a different file.

3. Is this problem addressed by the DBMS buffer manager prefetching pages? Explain.

4. Modern disks support *segmented caches*, with about four to six segments, each of which is used to cache pages from a different file. Does this technique help, with respect to the preceding problem? Given this technique, does it matter whether the DBMS buffer manager also does prefetching?

Answer 9.15 1. The pre-fetching done at the disk level varies widely across different drives and manufacturers, and pre-fetching is sufficiently important to a DBMS that one would like it to be independent of specific hardware support.

2. If there are many queries running concurrently, the request of a page from different queries can be interleaved. In the worst case, it cause the cache miss on every page request, even with disk pre-fetching.
3. If we have pre-fetching offered by DBMS buffer manager, the buffer manager can predict the reference pattern more accurately. In particular, a certain number of buffer frames can be allocated *per* active scan for pre-fetching purposes, and interleaved requests would not compete for the same frames.
4. *Segmented caches* can work in a similar fashion to allocating buffer frames for each active scan (as in the above answer). This helps to solve some of the concurrency problem, but will not be useful at all if more files are being accessed than the number of segments. In this case, the DBMS buffer manager should still prefer to do pre-fetching on its own to handle a larger number of files, and to predict more complicated access patterns.

Exercise 9.16 Describe two possible record formats. What are the trade-offs between them?

Answer 9.16 Two possible record formats are: *fixed length records* and *variable length records*. (For details, see the text.)

Fixed length record format is easy to implement. Since the record size is fixed, records can be stored contiguously. Record address can be obtained very quickly.

Variable length record format is much more flexible.

Exercise 9.17 Describe two possible page formats. What are the trade-offs between them?

Answer 9.17 Two possible page formats are: *consecutive slots* and *slot directory*

The consecutive slots organization is mostly used for fixed length record formats. It handles the deletion by using bitmaps or linked lists.

The slot directory organization maintains a directory of slots for each page, with a ⟨*record offset*, *record length*⟩ pair per slot.

The slot directory is an indirect way to get the offset of an entry. Because of this indirection, deletion is easy. It is accomplished by setting the length field to 0. And records can easily be moved around on the page without changing their external identifier.

Exercise 9.18 Consider the page format for variable-length records that uses a slot directory.

1. One approach to managing the slot directory is to use a maximum size (i.e., a maximum number of slots) and allocate the directory array when the page is created. Discuss the pros and cons of this approach with respect to the approach discussed in the text.

2. Suggest a modification to this page format that would allow us to sort records (according to the value in some field) without moving records and without changing the record ids.

Answer 9.18 The answer to each question is given below.

1. This approach is simpler, but less flexible. We can easily either allocate too much space for the slot directory or too little, since record lengths are variable and it is hard to estimate how many records are likely to fit on a given page.

2. One modification that would allow records to be sorted by a particular field is to store slot entries as <logical record number within page, offset> pairs and sort these based on the record's field value.

Exercise 9.19 Consider the two internal organizations for heap files (using lists of pages and a directory of pages) discussed in the text.

1. Describe them briefly and explain the trade-offs. Which organization would you choose if records are variable in length?

2. Can you suggest a single page format to implement both internal file organizations?

Answer 9.19 1. The linked-list approach is a little simpler, but finding a page with sufficient free space for a new record (especially with variable length records) is harder. We have to essentially scan the list of pages until we find one with enough space, whereas the directory organization allows us to find such a page by simply scanning the directory, which is much smaller than the entire file. The directory organization is therefore better, especially with variable length records.

2. A page format with *previous* and *next* page pointers would help in both cases. Obviously, such a page format allows us to build the linked list organization; it is also useful for implementing the directory in the directory organization.

Exercise 9.20 Consider a list-based organization of the pages in a heap file in which two lists are maintained: a list of *all* pages in the file and a list of all pages with free space. In contrast, the list-based organization discussed in the text maintains a list of full pages and a list of pages with free space.

1. What are the trade-offs, if any? Is one of them clearly superior?
2. For each of these organizations, describe a suitable page format.

Answer 9.20 1. In the first approach (a list of all pages and a list of pages with free space) a page with free space belongs to both lists. Thus, we need to have one set of pointers (*previous* and *next*) per list, per page. In the second approach, each page belongs to exactly one of these lists, and it suffices to have a single pair of *previous* and *next* pointers per page. Other than this, the two approaches are quite similar. The second approach therefore, is superior overall.

2. This is outlined in the answer to the previous part.

Exercise 9.21 Modern disk drives store more sectors on the outer tracks than the inner tracks. Since the rotation speed is constant, the sequential data transfer rate is also higher on the outer tracks. The seek time and rotational delay are unchanged. Considering this information, explain good strategies for placing files with the following kinds of access patterns:

1. Frequent, random accesses to a small file (e.g., catalog relations).
2. Sequential scans of a large file (e.g., selection from a relation with no index).
3. Random accesses to a large file via an index (e.g., selection from a relation via the index).
4. Sequential scans of a small file.

Answer 9.21 1. Place the file in the middle tracks. Sequential speed is not an issue due to the small size of the file, and the seek time is minimized by placing files in the center.

2. Place the file in the outer tracks. Sequential speed is most important and outer tracks maximize it.

3. Place the file and index on the inner tracks. The DBMS will alternately access pages of the index and of the file, and so the two should reside in close proximity to reduce seek times. By placing the file and the index on the inner tracks we also save valuable space on the faster (outer) tracks for other files that are accessed sequentially.

4. Place small files in the inner half of the disk. A scan of a small file is effectively random I/O because the cost is dominated by the cost of the initial seek to the beginning of the file.

Exercise 9.22 Why do frames in the buffer pool have a pin count instead of a pin flag?

Answer 9.22 Not yet available.

10

TREE-STRUCTURED INDEXING

Exercise 10.1 Consider the B+ tree index of order $d = 2$ shown in Figure 10.1.

1. Show the tree that would result from inserting a data entry with key 9 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes does the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
6. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.

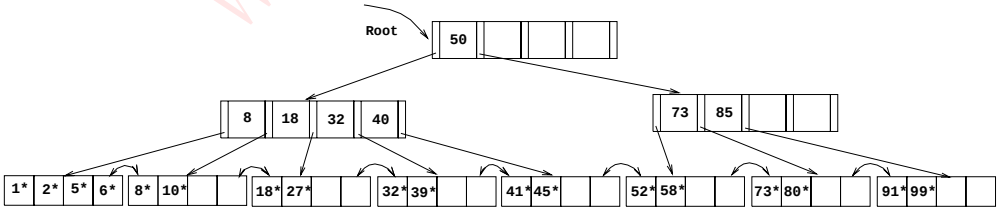


Figure 10.1 Tree for Exercise 10.1

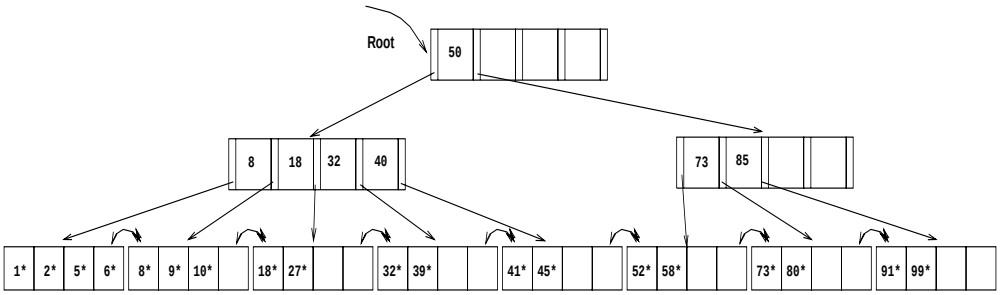


Figure 10.2

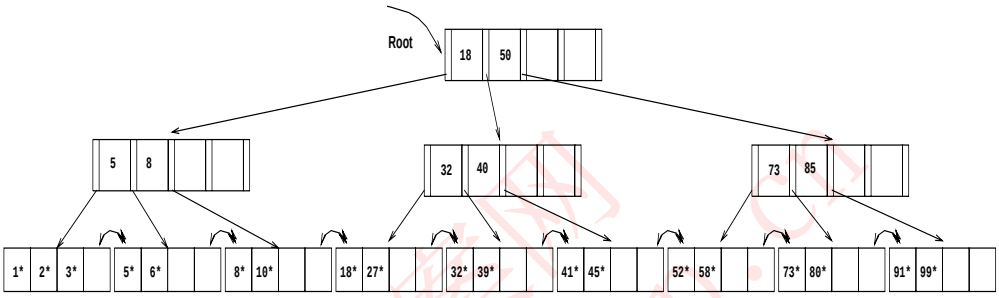


Figure 10.3

7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.

8. Show the B+ tree that would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

Answer 10.1 1. The data entry with key 9 is inserted on the second leaf page. The resulting tree is shown in figure 10.2.

2. The data entry with key 3 goes on the first leaf page F. Since F can accommodate at most four data entries ($d = 2$), F splits. The lowest data entry of the new leaf is given up to the ancestor which also splits. The result can be seen in figure 10.3. The insertion will require 5 page writes, 4 page reads and allocation of 2 new pages.

3. The data entry with key 8 is deleted, resulting in a leaf page N with less than two data entries. The left sibling L is checked for redistribution. Since L has more than two data entries, the remaining keys are redistributed between L and N, resulting in the tree in figure 10.4.

4. As is part 3, the data entry with key 8 is deleted from the leaf page N. N's right sibling R is checked for redistribution, but R has the minimum number of

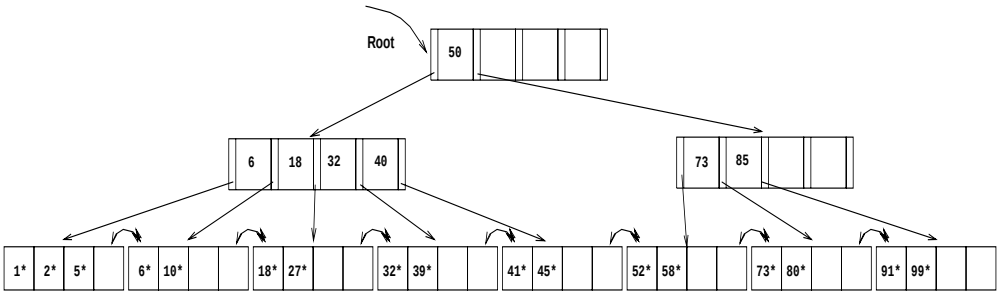


Figure 10.4

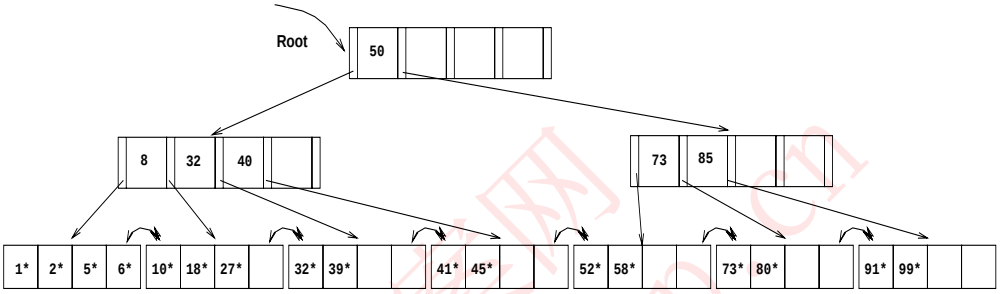


Figure 10.5

keys. Therefore the two siblings merge. The key in the ancestor which distinguished between the newly merged leaves is deleted. The resulting tree is shown in figure 10.5.

5. The data entry with key 46 can be inserted without any structural changes in the tree. But the removal of the data entry with key 52 causes its leaf page L to merge with a sibling (we chose the right sibling). This results in the removal of a key in the ancestor A of L and thereby lowering the number of keys on A below the minimum number of keys. Since the left sibling B of A has more than the minimum number of keys, redistribution between A and B takes place. The final tree is depicted in figure 10.6.

6. Deleting the data entry with key 91 causes a scenario similar to part 5. The result can be seen in figure 10.7.

7. The data entry with key 59 can be inserted without any structural changes in the tree. No sibling of the leaf page with the data entry with key 91 is affected by the insert. Therefore deleting the data entry with key 91 changes the tree in a way very similar to part 6. The result is depicted in figure 10.8.

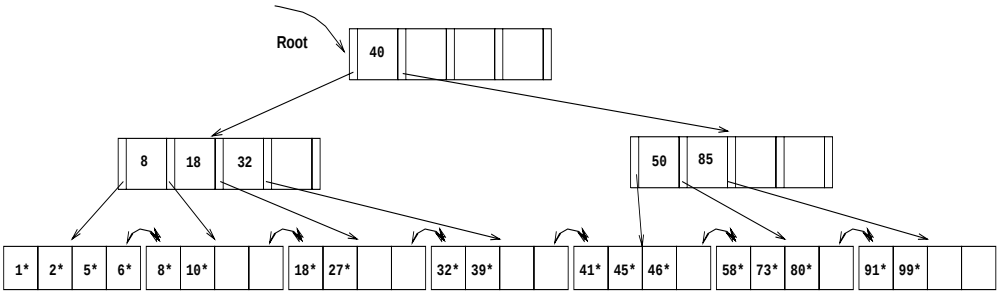


Figure 10.6

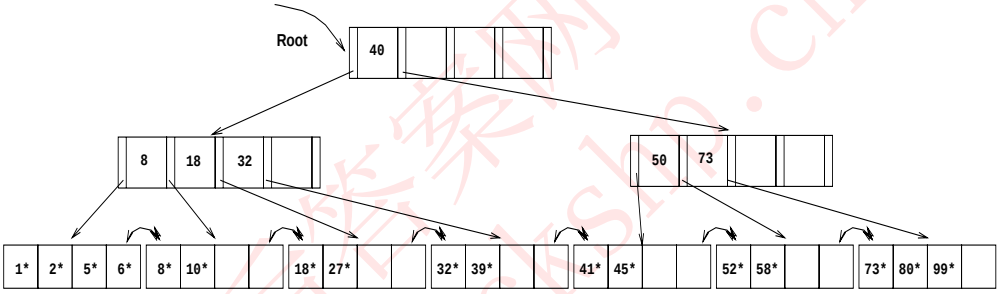


Figure 10.7

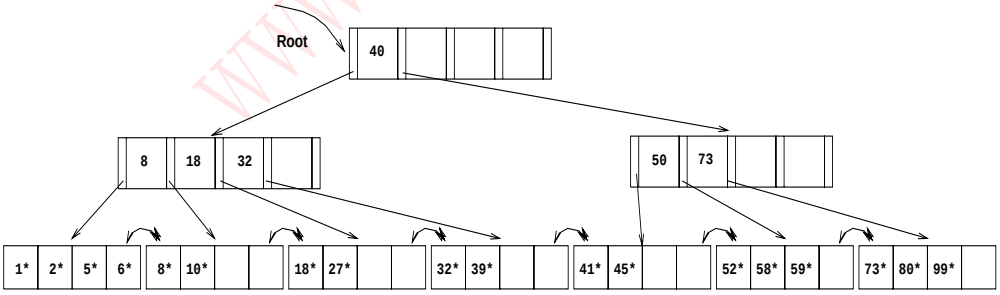


Figure 10.8

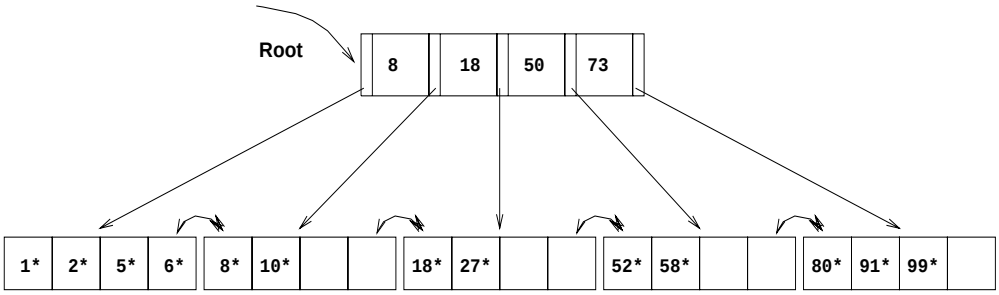


Figure 10.9

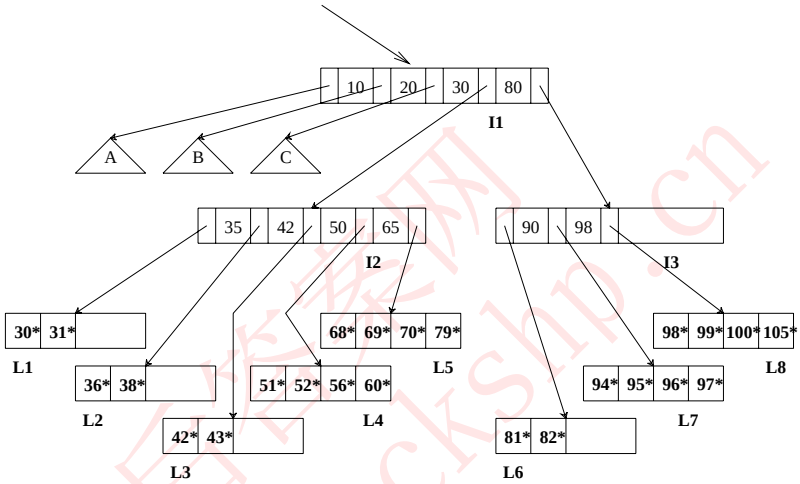


Figure 10.10 Tree for Exercise 10.2

8. Considering checking the right sibling for possible merging first, the successive deletion of the data entries with keys 32, 39, 41, 45 and 73 results in the tree shown in figure 10.9.

Exercise 10.2 Consider the B+ tree index shown in Figure 10.10, which uses Alternative (1) for data entries. Each intermediate node can hold up to five pointers and four key values. Each leaf can hold up to four records, and leaf nodes are doubly linked as usual, although these links are not shown in the figure. Answer the following questions.

1. Name all the tree nodes that must be fetched to answer the following query: "Get all records with search key greater than 38."

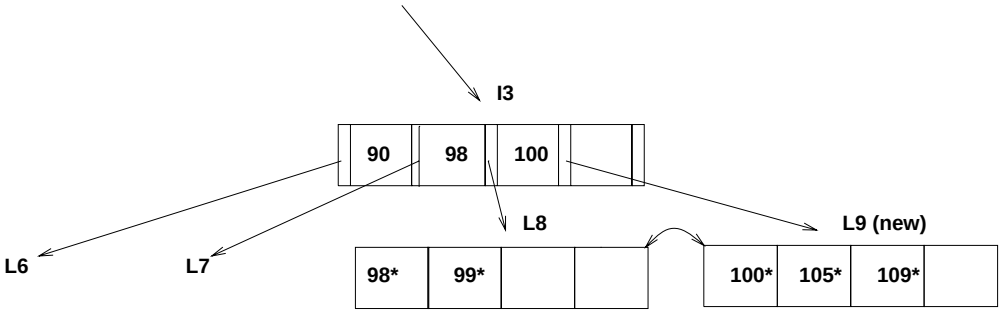


Figure 10.11

2. Show the B+ tree that would result from inserting a record with search key 109 into the tree.
3. Show the B+ tree that would result from deleting the record with search key 81 from the original tree.
4. Name a search key value such that inserting it into the (original) tree would cause an increase in the height of the tree.
5. Note that subtrees A, B, and C are not fully specified. Nonetheless, what can you infer about the contents and the shape of these trees?
6. How would your answers to the preceding questions change if this were an ISAM index?
7. Suppose that this is an ISAM index. What is the minimum number of insertions needed to create a chain of three overflow pages?

Answer 10.2 The answer to each question is given below.

1. I1, I2, and everything in the range [L2..L8].
2. See Figure 10.11. Notice that node L8 is split into two nodes.
3. Assuming that there is redistribution from the right sibling, the solution can be seen in Figure 10.12.
4. There are many search keys X such that inserting X would increase the height of the tree. Any search key in the range [65..79] would suffice. A key in this range would go in L5 if there were room for it, but since L5 is full already and since it can't redistribute any data entries over to L4 (L4 is full also), it must split; this in turn causes I2 to split, which causes I1 to split, and assuming I1 is the root node, a new root is created and the tree becomes taller.

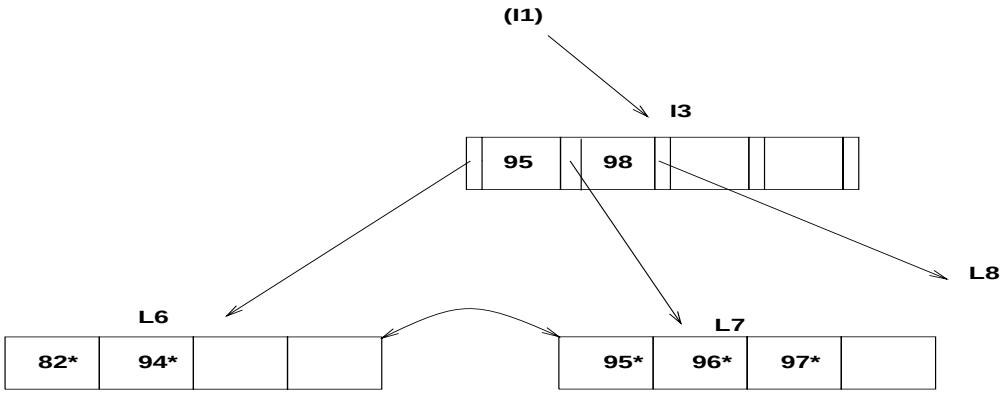


Figure 10.12

5. We can infer several things about subtrees A, B, and C. First of all, they each must have height one, since their "sibling" trees (those rooted at I2 and I3) have height one. Also, we know the ranges of these trees (assuming duplicates fit on the same leaf): subtree A holds search keys less than 10, B contains keys ≥ 10 and < 20, and C has keys ≥ 20 and < 30. In addition, each intermediate node has at least 2 key values and 3 pointers.

6. The answers for the questions above would change as follows if we were dealing with ISAM trees instead of B+ trees.

   (a) This is only a search, so the answer is the same. (The tree structure is not modified.)

   (b) Because we can never split a node in ISAM, we must create an overflow page to hold inserted key 109.

   (c) Search key 81 would simply be erased from L6; no redistribution would occur (ISAM has no minimum occupation requirements).

   (d) Being a *static* tree structure, an ISAM tree will never change height in normal operation, so there are no search keys which when inserted will increase the tree's height. (If we inserted an X in [65..79] we would have to create an overflow page for L5.)

   (e) We can infer several things about subtrees A, B, and C. First of all, they each must have height one, since their "sibling" trees (those rooted at I2 and I3) have height one. Here we suppose that we create a balanced ISAM tree. Also, we know the ranges of these trees (assuming duplicates fit on the same leaf): subtree A holds search keys less than 10, B contains keys ≥ 10 and < 20, and C has keys ≥ 20 and < 30. Additionally, each of A, B, and C contains five leaf nodes (which may be of arbitrary fullness), and these nodes are the first 15 consecutive pages prior to L1.

7. If this is an ISAM tree, we would have to insert at least nine search keys in order to develop an overflow chain of length three. These keys could be any that would map to L4, L5, L7, or L8, all of which are full and thus would need overflow pages on the next insertion. The first insert to one of these pages would create the first overflow page, the fifth insert would create the second overflow page, and the ninth insert would create the third overflow page (for a total of one leaf and three overflow pages).

Exercise 10.3 Answer the following questions:

1. What is the minimum space utilization for a B+ tree index?
2. What is the minimum space utilization for an ISAM index?
3. If your database system supported both a static and a dynamic tree index (say, ISAM and B+ trees), would you ever consider using the *static* index in preference to the *dynamic* index?

Answer 10.3 The answer to each question is given below.

1. By the definition of a B+ tree, each index page, except for the root, has at least d and at most $2d$ key entries. Therefore—with the exception of the root—the minimum space utilization guaranteed by a B+ tree index is 50 percent.
2. The minimum space utilization by an ISAM index depends on the design of the index and the data distribution over the lifetime of ISAM index. Since an ISAM index is static, empty spaces in index pages are never filled (in contrast to a B+ tree index, which is a dynamic index). Therefore the space utilization of ISAM index pages is usually close to 100 percent by design. However, there is no guarantee for leaf pages' utilization.
3. A static index without overflow pages is faster than a dynamic index on inserts and deletes, since index pages are only read and never written. If the set of keys that will be inserted into the tree is known in advance, then it is possible to build a static index which reserves enough space for all possible future inserts. Also if the system goes periodically off line, static indices can be rebuilt and scaled to the current occupancy of the index. Infrequent or scheduled updates are flags for when to consider a static index structure.

Exercise 10.4 Suppose that a page can contain at most four data values and that all data values are integers. Using only B+ trees of order 2, give examples of each of the following:

1. A B+ tree whose height changes from 2 to 3 when the value 25 is inserted. Show your structure before and after the insertion.

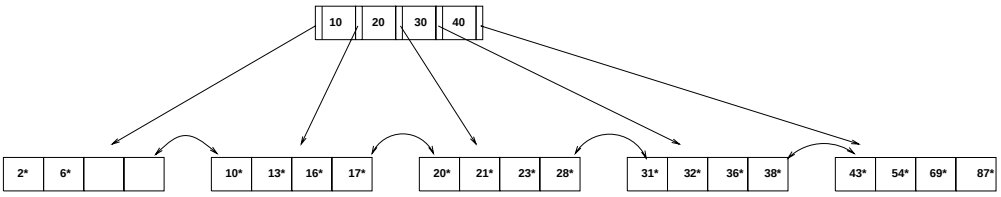


Figure 10.13

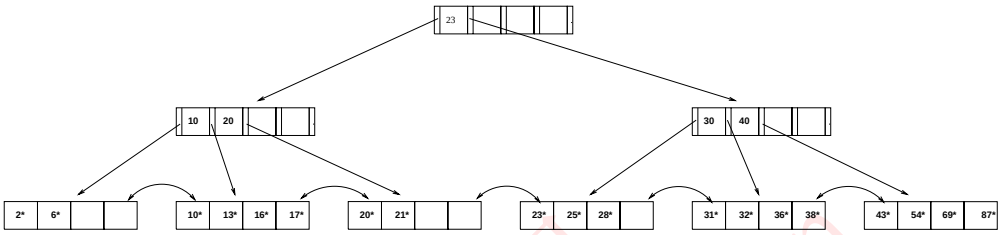


Figure 10.14

2. A B+ tree in which the deletion of the value 25 leads to a redistribution. Show your structure before and after the deletion.

3. A B+ tree in which the deletion of the value 25 causes a merge of two nodes but without altering the height of the tree.

4. An ISAM structure with four buckets, none of which has an overflow page. Further, every bucket has space for exactly one more entry. Show your structure before and after inserting two additional values, chosen so that an overflow page is created.

Answer 10.4 For these answers, two illustrations are given, one showing the tree before the specified change and one showing it after.

1. See Figures 10.13 and 10.14.

2. See Figures 10.15 and 10.16.

3. See Figures 10.17 and 10.18.

4. See Figures 10.19 and 10.20 (inserted 27 and 29).

Exercise 10.5 Consider the B+ tree shown in Figure 10.21.

1. Identify a list of five data entries such that:

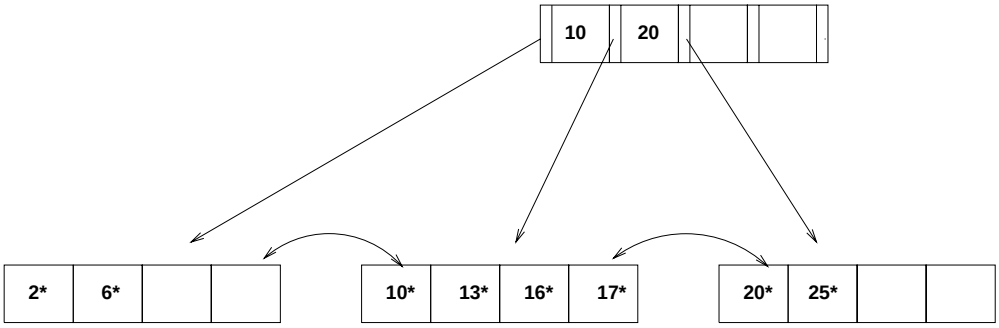


Figure 10.15

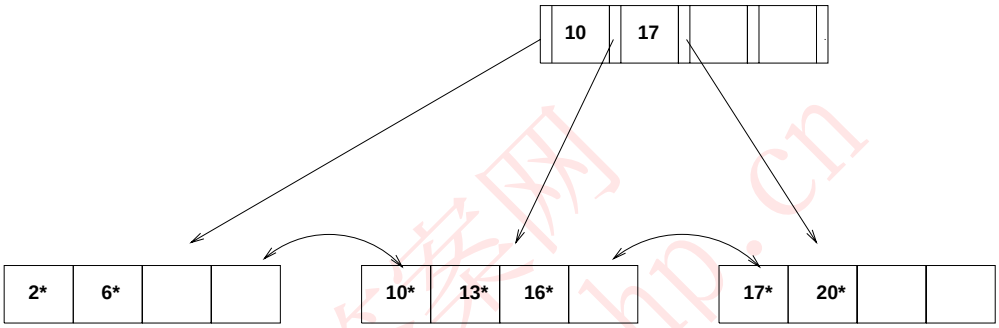


Figure 10.16

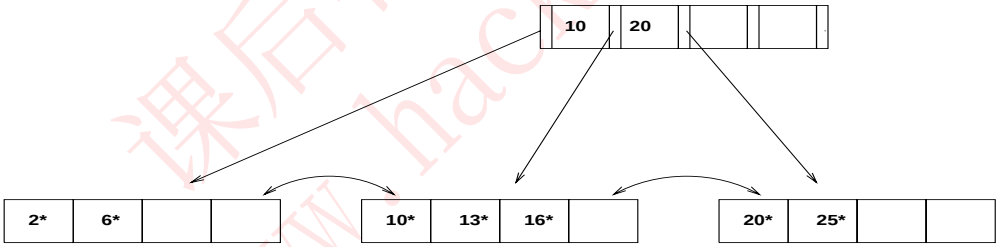


Figure 10.17

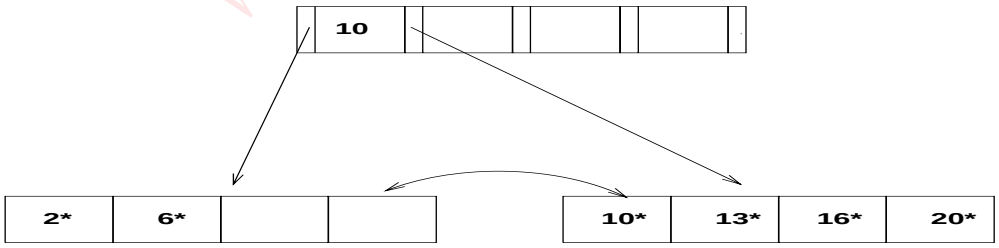


Figure 10.18

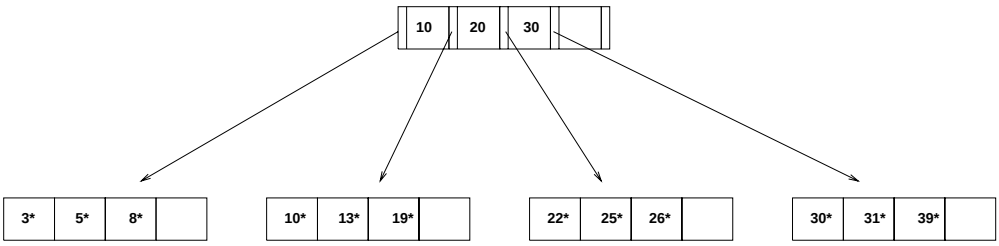


Figure 10.19

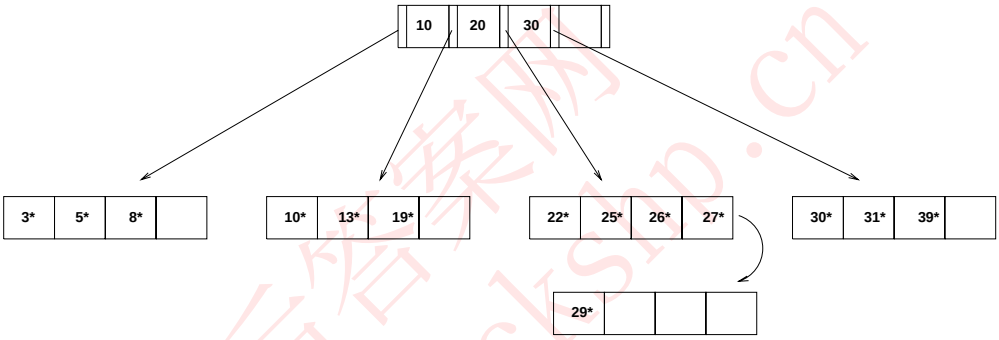


Figure 10.20

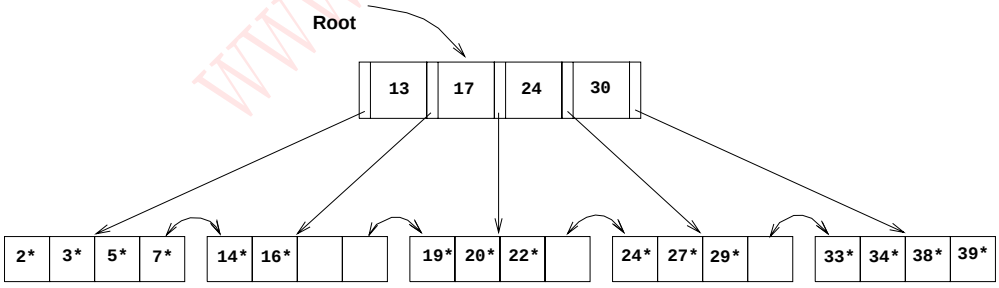


Figure 10.21 Tree for Exercise 10.5

(a) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in the original tree.

(b) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert a, insert b, delete b, delete a) results in a different tree.

2. What is the minimum number of insertions of data entries with distinct keys that will cause the height of the (original) tree to change from its current value (of 1) to 3?

3. Would the minimum number of insertions that will cause the original tree to increase to height 3 change if you were allowed to insert duplicates (multiple data entries with the same key), assuming that overflow pages are not used for handling duplicates?

Answer 10.5 The answer to each question is given below.

1. The answer to each part is given below.

   (a) One example is the set of five data entries with keys 17, 18, 13, 15, and 25. Inserting 17 and 18 will cause the tree to split and gain a level. Inserting 13, 15, and 25 does change the tree structure any further, so deleting them in reverse order causes no structure change. When 18 is deleted, redistribution will be possible from an adjacent node since one node will contain only the value 17, and its right neighbor will contain 19, 20, and 22. Finally, when 17 is deleted, no redistribution will be possible so the tree will loose a level and will return to the original tree.

   (b) Inserting and deleting the set 13, 15, 18, 25, and 4 will cause a change in the tree structure. When 4 is inserted, the right most leave will split causing the tree to gain a level. When it is deleted, the tree will not shrink in size. Since inserts 13, 15, 18, and 25 did not affect the right most node, their deletion will not change the altered structure either.

2. Let us call the current tree depicted in Figure 10.21 T. T has 16 data entries. The smallest tree S of height 3 which is created exclusively through inserts has $(1*2*3*3)*2+1 = 37$ data entries in its leaf pages. S has 18 leaf pages with two data entries each and one leaf page with three data entries. T has already four leaf pages which have more than two data entries; they can be filled and made to split, but after each spilt, one of the two pages will still has three data entries remaining. Therefore the smallest tree of height 3 which can possibly be created from T only through inserts has $(1*2*3*3)*2+4 = 40$ data entries. Therefore the minimum number of entries that will cause the height of T to change to 3 is $40 - 16$=24.

3. The argument in part 2 does not assume anything about the data entries to be inserted; it is valid if duplicates can be inserted as well. Therefore the solution does not change.

Exercise 10.6 Answer Exercise 10.5 assuming that the tree is an ISAM tree! (Some of the examples asked for may not exist—if so, explain briefly.)

Answer 10.6 The answer to each question is given below.

1. The answer to each part is given below
   (a) Since ISAM trees use overflow buckets, any series of five inserts and deletes will result in the same tree.
   (b) If the leaves are not sorted, there is no sequence of inserts and deletes that will change the overall structure of an ISAM index. This is because inserts will create overflow buckets, and these overflow buckets will be removed when the elements are deleted, giving the original tree.
2. The height of the tree does never change since an ISAM index is static. If a leaf page becomes full, an overflow page is allocated; if a leaf page becomes empty, it remains empty.
3. See part 2.

Exercise 10.7 Suppose that you have a sorted file and want to construct a dense primary B+ tree index on this file.

1. One way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure. What performance and storage utilization problems are there with this approach?
2. Explain how the bulk-loading algorithm described in the text improves upon this scheme.

Answer 10.7 1. This approach is likely to be quite expensive, since each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable. Also, according to the insertion algorithm, each time a node splits, the data entries are redistributed evenly to both nodes. This leads to a fixed page utilization of 50%

2. The bulk loading algorithm has good performance and space utilization compared with the repeated inserts approach. Since the B+ tree is grown from the bottom up, the bulk loading algorithm allows the administrator to pre-set the amount each index and data page should be filled. This allows good performance for future inserts, and supports some desired space utilization.

Exercise 10.8 Assume that you have just built a dense B+ tree index using Alternative (2) on a heap file containing 20,000 records. The key field for this B+ tree index is a 40-byte string, and it is a candidate key. Pointers (i.e., record ids and page ids) are (at most) 10-byte values. The size of one disk page is 1000 bytes. The index was built in a bottom-up fashion using the bulk-loading algorithm, and the nodes at each level were filled up as much as possible.

1. How many levels does the resulting tree have?
2. For each level of the tree, how many nodes are at that level?
3. How many levels would the resulting tree have if key compression is used and it reduces the average size of each key in an entry to 10 bytes?
4. How many levels would the resulting tree have without key compression but with all pages 70 percent full?

Answer 10.8 The answer to each question is given below.

1. Since the index is a primary dense index, there are as many data entries in the B+ tree as records in the heap file. An index page consists of at most $2d$ keys and $2d+1$ pointers. So we have to maximize d under the condition that $2d \cdot 40 + (2d+1) \cdot 10 \leq 1000$. The solution is $d = 9$, which means that we can have 18 keys and 19 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (1000/50)=20 data entries. The resulting tree has $\lceil \log_{19}(20000/20) + 1 \rceil = 4$ levels.
2. Since the nodes at each level are filled as much as possible, there are $\lceil 20000/20 \rceil = 1000$ leaf nodes (on level 4). (A full index node has $2d+1 = 19$ children.) Therefore there are $\lceil 1000/19 \rceil = 53$ index pages on level 3, $\lceil 53/19 \rceil = 3$ index pages on level 2, and there is one index page on level 1 (the root of the tree).
3. Here the solution is similar to part 1, except the key is of size 10 instead of size 40. An index page consists of at most $2d$ keys and $2d + 1$ pointers. So we have to maximize d under the condition that $2d \cdot 10 + (2d + 1) \cdot 10 \leq 1000$. The solution is $d = 24$, which means that we can have 48 keys and 49 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 10+10=20 bytes. Therefore a leaf page has space for (1000/20)=50 data entries. The resulting tree has $\lceil \log_{49}(20000/50) + 1 \rceil = 3$ levels.
4. Since each page should be filled only 70 percent, this means that the usable size of a page is $1000 \cdot 0.70 = 700$ bytes. Now the calculation is the same as in part 1 but using pages of size 700 instead of size 1000. An index page consists of at most $2d$ keys and $2d + 1$ pointers. So we have to maximize d under the condition that $2d \cdot 40 + (2d+1) \cdot 10 \leq 700$. The solution is $d = 6$, which means that we can have 12 keys and 13 pointers on an index page. A record on a leaf page consists of the key

field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (700/50)=14 data entries. The resulting tree has $\lceil \log_{13}(20000/14) + 1 \rceil = 4$ levels.

Exercise 10.9 The algorithms for insertion and deletion into a B+ tree are presented as recursive algorithms. In the code for *insert*, for instance, a call is made at the parent of a node N to insert into (the subtree rooted at) node N, and when this call returns, the current node is the parent of N. Thus, we do not maintain any 'parent pointers' in nodes of B+ tree. Such pointers are not part of the B+ tree structure for a good reason, as this exercise demonstrates. An alternative approach that uses parent pointers—again, remember that such pointers are *not* part of the standard B+ tree structure!—in each node appears to be simpler:

> Search to the appropriate leaf using the search algorithm; then insert the entry and split if necessary, with splits propagated to parents if necessary (using the parent pointers to find the parents).

Consider this (unsatisfactory) alternative approach:

1. Suppose that an internal node N is split into nodes N and $N2$. What can you say about the parent pointers in the children of the original node N?
2. Suggest two ways of dealing with the inconsistent parent pointers in the children of node N.
3. For each of these suggestions, identify a potential (major) disadvantage.
4. What conclusions can you draw from this exercise?

Answer 10.9 The answer to each question is given below.

1. The parent pointers in either d or $d+1$ of the children of the original node N are not valid any more: they still point to N, but they should point to $N2$.
2. One solution is to adjust all parent pointers in the children of the original node N which became children of $N2$. Another solution is to leave the pointers during the insert operation and to adjust them later when the page is actually needed and read into memory anyway.
3. The first solution requires at least $d+1$ additional page reads (and sometime later, page writes) on an insert, which would result in a remarkable slowdown. In the second solution mentioned above, a child M, which has a parent pointer to be adjusted, is updated if an operation is performed which actually reads M into

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	3.8
53666	Jones	jones@cs	18	3.4
53901	Jones	jones@toy	18	3.4
53902	Jones	jones@physics	18	3.4
53903	Jones	jones@english	18	3.4
53904	Jones	jones@genetics	18	3.4
53905	Jones	jones@astro	18	3.4
53906	Jones	jones@chem	18	3.4
53902	Jones	jones@sanitation	18	3.8
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8
54001	Smith	smith@ee	19	3.5
54005	Smith	smith@cs	19	3.8
54009	Smith	smith@astro	19	2.2

Figure 10.22 An Instance of the Students Relation

memory (maybe on a down path from the root to a leaf page). But this solution modifies M and therefore requires sometime later a write of M, which might not have been necessary if there were no parent pointers.

4. In conclusion, to add parent pointers to the B+ tree data structure is not a good modification. Parent pointers cause unnecessary page updates and so lead to a decrease in performance.

Exercise 10.10 Consider the instance of the Students relation shown in Figure 10.22. Show a B+ tree of order 2 in each of these cases below, assuming that duplicates are handled using overflow pages. Clearly indicate what the data entries are (i.e., do not use the $k*$ convention).

1. A B+ tree index on *age* using Alternative (1) for data entries.
2. A dense B+ tree index on *gpa* using Alternative (2) for data entries. For this question, assume that these tuples are stored in a sorted file in the order shown in Figure 10.22: The first tuple is in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use $\langle$*page-id, slot*$\rangle$ to identify a tuple.

Answer 10.10 The answer to each question is given below.

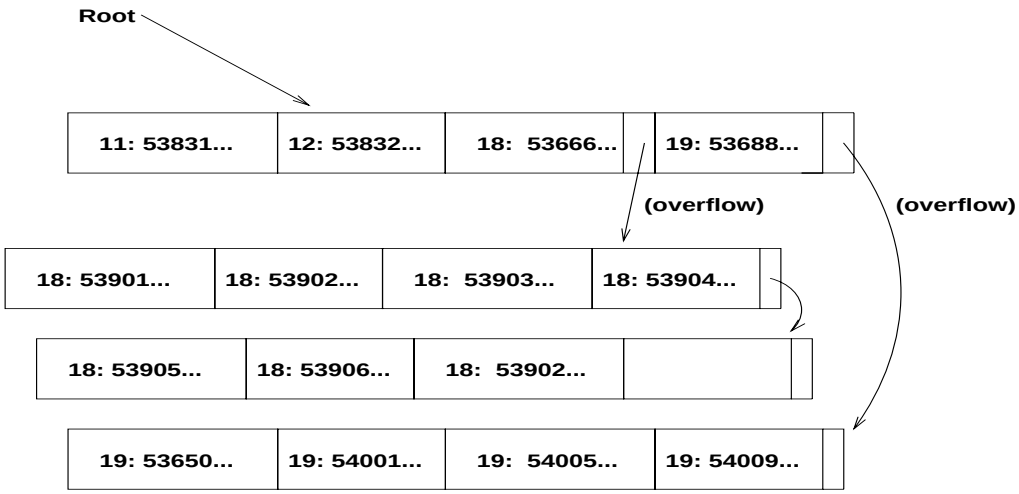


Figure 10.23

1. See Figure 10.23.
2. See Figure 10.24. Note that the data entries are not necessarily stored in the same order as the data records, reflecting the fact that they may have been inserted in a different order. We assume a simple insertion algorithm that locates a leaf in the usual way, and if the leaf already contains a data entry with the given key value, puts the new data entry into the overflow chain associated with the leaf. Thus, the data entries in a leaf have distinct key values. An obvious problem that arises here is that when the leaf splits (because a data entry with a new key value is inserted into the leaf when the leaf is full), the overflow chain must be scanned to ensure that when a data entry is moved to the new leaf node, all data entries with that key value are moved. An alternative is to maintain a separate overflow chain for each key value with duplicates, but considering the capacity of a page (high), and the likely number of duplicates for a given key value (probably low), this may lead to poor space utilization.

Exercise 10.11 Suppose that duplicates are handled using the approach without overflow pages discussed in Section 10.7. Describe an algorithm to search for the left-most occurrence of a data entry with search key value K.

Answer 10.11 The key to understanding this problem is to observe that when a leaf splits due to inserted duplicates, then of the two resulting leaves, it may happen that the left leaf contains other search key values less than the duplicated search key value. Furthermore, it could happen that the least element on the right leaf could be the duplicated value. (This scenario could arise, for example, when the majority of data entries on the original leaf were for search keys of the duplicated value.) The

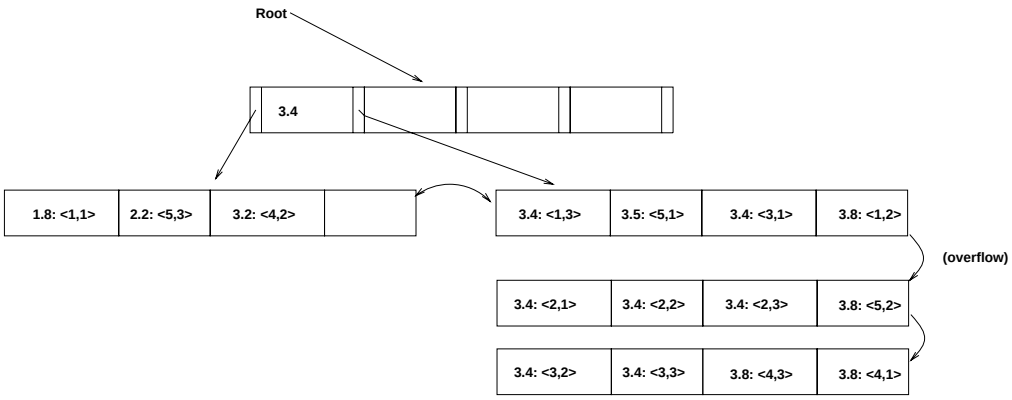


Figure 10.24

parent index node (assuming the tree is of at least height 2) will have an entry for the duplicated value with a pointer to the rightmost leaf.

If this leaf continues to be filled with entries having the same duplicated key value, it could split again causing another entry with the same key value to be inserted in the parent node. Thus, the same key value could appear many times in the index nodes as well. While searching for entries with a given key value, the search should proceed by using the left-most of the entries on an index page such that the key value is less than or equal to the given key value. Moreover, on reaching the leaf level, it is possible that there are entries with the given key value (call it k) on the page to the *left* of the current leaf page, unless some entry with a smaller key value is present on this leaf page. Thus, we must scan to the left using the neighbor pointers at the leaf level until we find an entry with a key value *less than* k (or come to the beginning of the leaf pages). Then, we must scan forward along the leaf level until we find an entry with a key value *greater than* k.

Exercise 10.12 Answer Exercise 10.10 assuming that duplicates are handled without using overflow pages, using the alternative approach suggested in Section 9.7.

Answer 10.12 The answer to each question is given below.

1. See Figure 10.25.
2. See Figure 10.26, which was constructed with the assumption that data entries for the records were inserted in the order that records are shown in Figure 10.22. An important point to note is that when we search for data entries with key value 3.8, the path from the root leads to the right-most leaf node, *even though there is an entry with key value 3.8 on the previous leaf node!* Thus, to retrieve all data

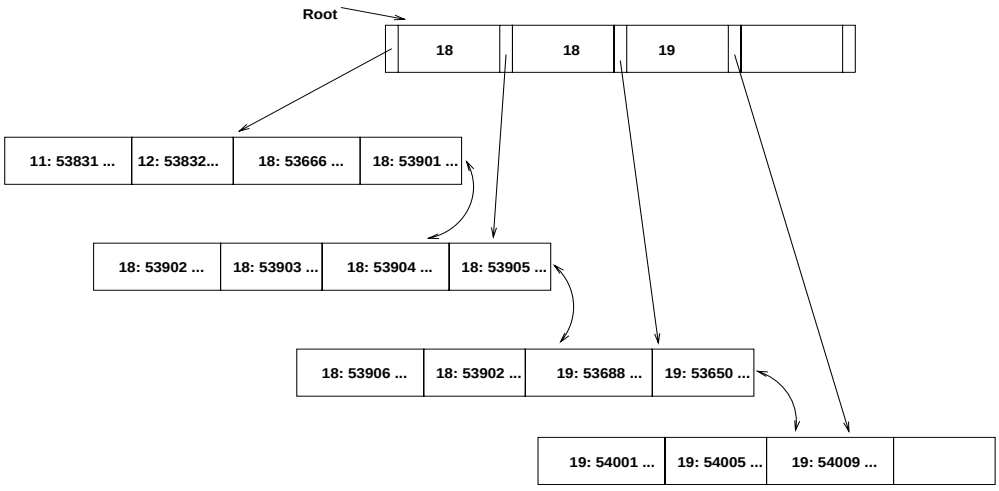


Figure 10.25

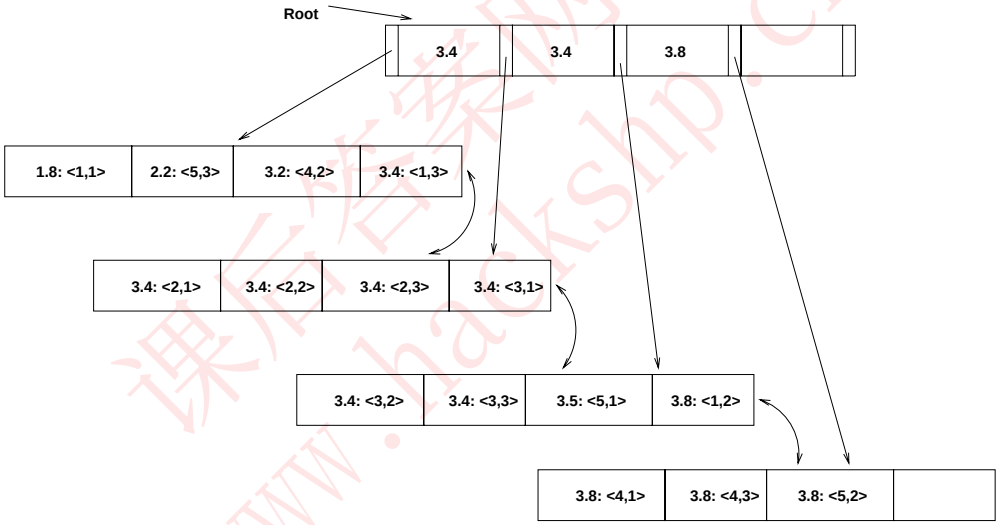


Figure 10.26

entries with a given key value (in this example query, the value 3.8), we must first scan the leaf nodes to the left, using the neighbor pointers, until we find a data entry with key value *less* than the query (i.e., less than 3.8), or until we have searched the left-most leaf node. Then, we must scan forward, again using the neighbor pointers, until we find a data entry with key value greater than the query, or until we have examined the right-most leaf node.

11

HASH-BASED INDEXING

Exercise 11.1 Consider the Extendible Hashing index shown in Figure 11.1. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you are told that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 68.

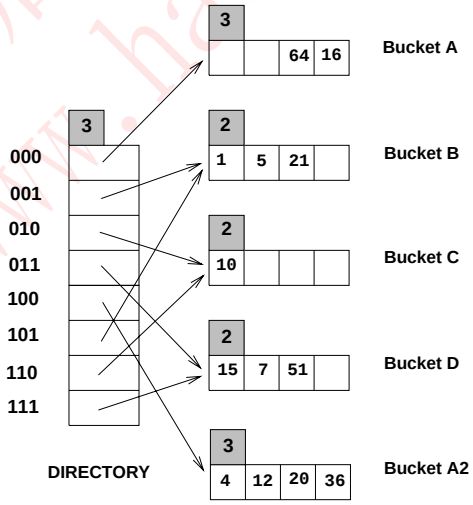


Figure 11.1 Figure for Exercise 11.1

5. Show the index after inserting entries with hash values 17 and 69 into the original tree.

6. Show the index after deleting the entry with hash value 21 into the original tree. (Assume that the full deletion algorithm is used.)

7. Show the index after deleting the entry with hash value 10 into the original tree. Is a merge triggered by this deletion? If not, explain why. (Assume that the full deletion algorithm is used.)

Answer 11.1 The answer to each question is given below.

1. It could be any one of the data entries in the index. We can always find a sequence of insertions and deletions with a particular key value, among the key values shown in the index as the last insertion. For example, consider the data entry 16 and the following sequence:
   1 5 21 10 15 7 51 4 12 36 64 8 24 56 16 56_D 24_D 8_D
   The last insertion is the data entry 16 and it also causes a split. But the sequence of deletions following this insertion cause a merge leading to the index structure shown in Fig 11.1.

2. The last insertion could not have caused a split because the total number of data entries in the buckets A and A_2 is 6. If the last entry caused a split the total would have been 5.

3. The last insertion which caused a split cannot be in bucket C. Buckets B and C or C and D could have made a possible bucket-split image combination but the total number of data entries in these combinations is 4 and the absence of deletions demands a sum of at least 5 data entries for such combinations. Buckets B and D can form a possible bucket-split image combination because they have a total of 6 data entries between themselves. So do A and A_2. But for the B and D to be split images the starting global depth should have been 1. If the starting global depth is 2, then the last insertion causing a split would be in A or A_2.

4. See Fig 11.2.

5. See Fig 11.3.

6. See Fig 11.4.

7. The deletion of the data entry 10 which is the only data entry in bucket C doesn't trigger a merge because bucket C is a primary page and it is left as a place holder. Right now, directory element 010 and its split image 110 already point to the same bucket C. We can't do a further merge.

   See Fig 11.5.

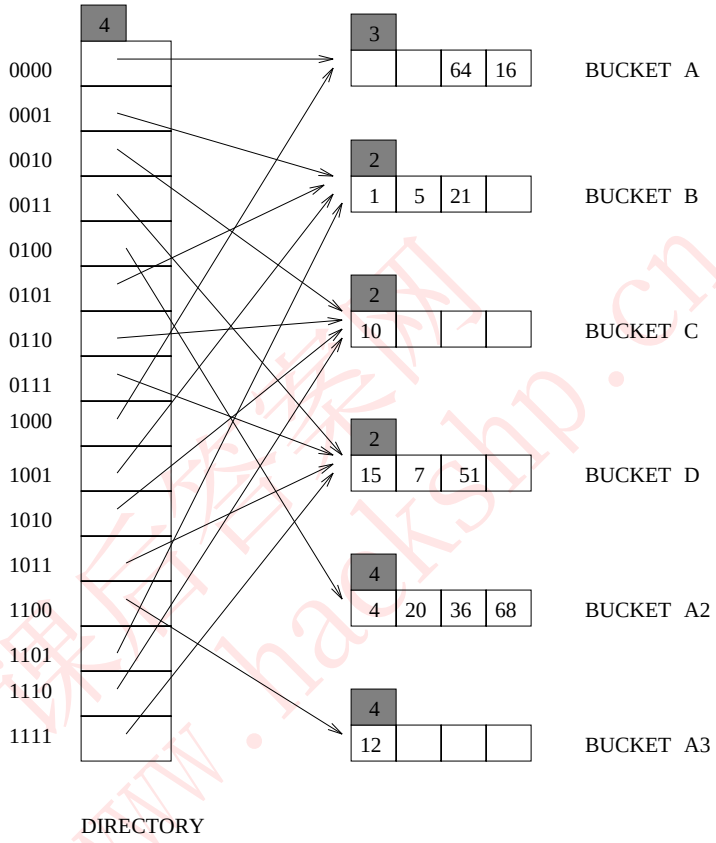


Figure 11.2

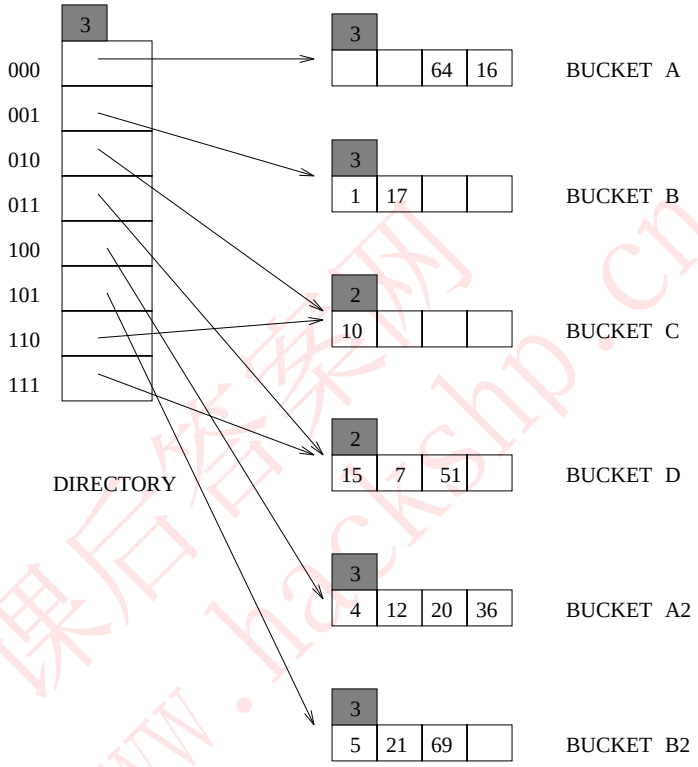


Figure 11.3

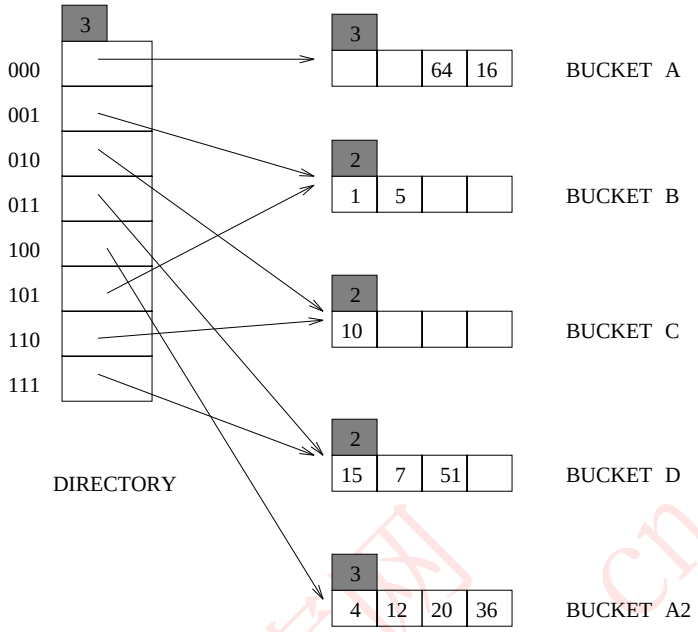


Figure 11.4

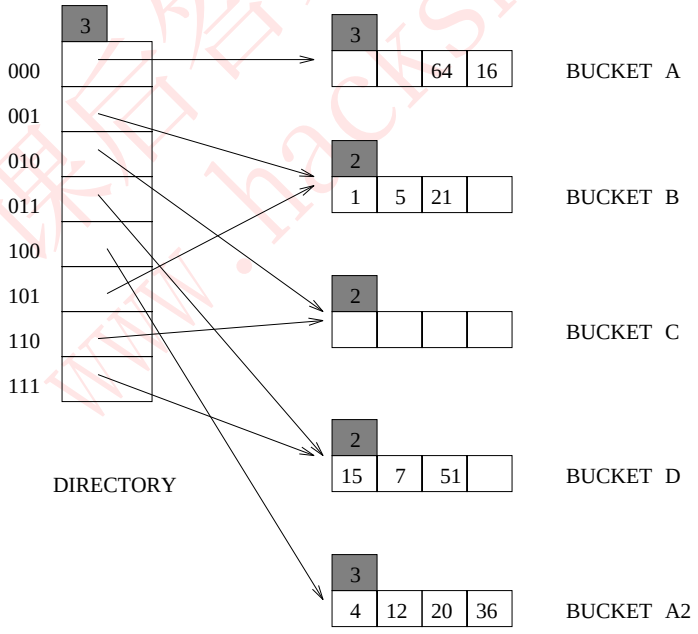


Figure 11.5

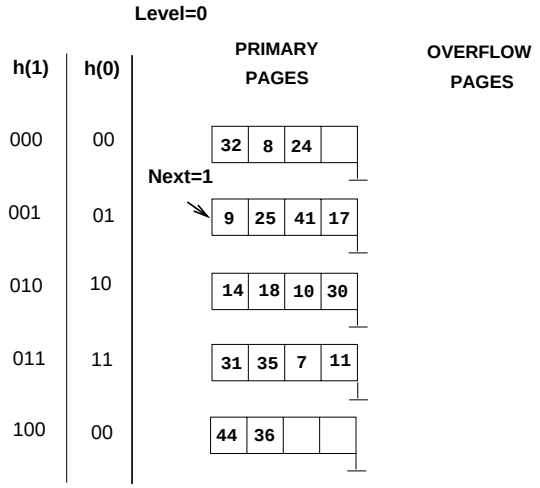


Figure 11.6 Figure for Exercise 11.2

Exercise 11.2 Consider the Linear Hashing index shown in Figure 11.6. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you know that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 4.
5. Show the index after inserting an entry with hash value 15 into the original tree.
6. Show the index after deleting the entries with hash values 36 and 44 into the original tree. (Assume that the full deletion algorithm is used.)
7. Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this. What is the maximum number of entries that can be inserted into this bucket before a split occurs that reduces the length of this overflow chain?

Answer 11.2 The answer to each question is given below.

1. Nothing can be said about the last entry into the index: it can be any of the data entries in the index.

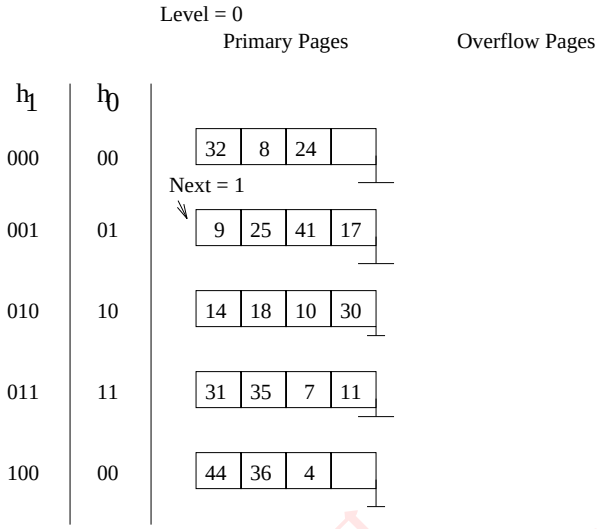


Figure 11.7 Index from Figure 11.6 after insertion of an entry with hash value 4

2. If the last item that was inserted had a hashcode $h_0(keyvalue) = 00$ then it caused a split, otherwise, any value could have been inserted.

3. The last data entry which caused a split satisfies the condition

$$h_0(keyvalue) = 00$$

as there are no overflow pages for any of the other buckets.

4. See Fig 11.7

5. See Fig 11.8

6. See Fig 11.9

7. The following constitutes the minimum list of entries to cause two overflow pages in the index :

$$63, 127, 255, 511, 1023$$

The first insertion causes a split and causes an update of $Next$ to 2. The insertion of 1023 causes a subsequent split and $Next$ is updated to 3 which points to this bucket.

This overflow chain will not be redistributed until three more insertions (a total of 8 entries) are made. In principle if we choose data entries with key values of the form $2^k + 3$ with sufficiently large k, we can take the maximum number of entries that can be inserted to reduce the length of the overflow chain to be greater than

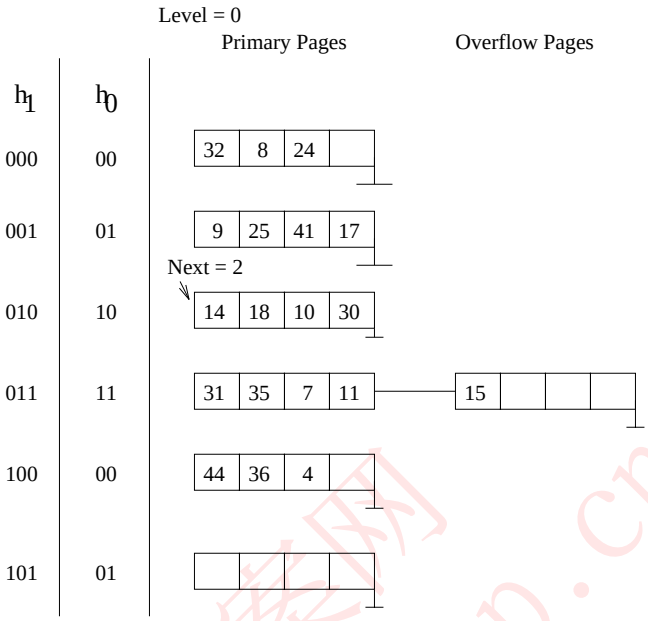


Figure 11.8 Index from Figure 11.6 after insertion of an entry with hash value 15

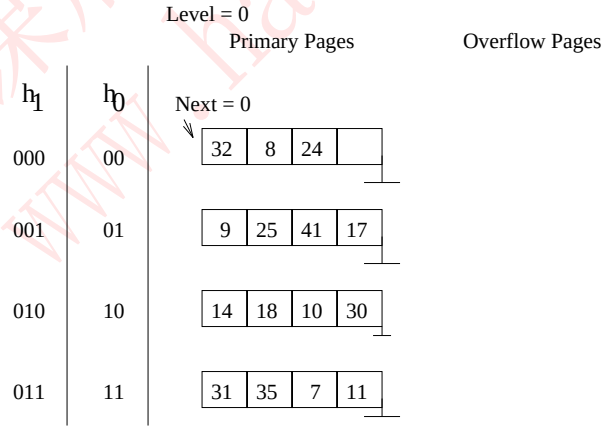


Figure 11.9 Index from Figure 11.6 after deletion of entries with hash values 36 and 44

any arbitrary number. This is so because the initial index has 31(binary 11111), 35(binary 10011),7(binary 111) and 11(binary 1011). So by an appropriate choice of data entries as mentioned above we can make a split of this bucket cause just two values (7 and 31) to be redistributed to the new bucket. By choosing a sufficiently large k we can delay the reduction of the length of the overflow chain till any number of splits of this bucket.

Exercise 11.3 Answer the following questions about Extendible Hashing:

1. Explain why local depth and global depth are needed.
2. After an insertion that causes the directory size to double, how many buckets have exactly one directory entry pointing to them? If an entry is then deleted from one of these buckets, what happens to the directory size? Explain your answers briefly.
3. Does Extendible Hashing guarantee at most one disk access to retrieve a record with a given key value?
4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the size of the directory? What can you say about the space utilization in data pages (i.e., non-directory pages)?
5. Does doubling the directory require us to examine all buckets with local depth equal to global depth?
6. Why is handling duplicate key values in Extendible Hashing harder than in ISAM?

Answer 11.3 The answer to each question is given below.

1. Extendible hashing allows the size of the directory to increase and decrease depending on the number and variety of inserts and deletes. Once the directory size changes, the hash function applied to the search key value should also change. So there should be some information in the index as to which hash function is to be applied. This information is provided by the *global depth*.

   An increase in the directory size doesn't cause the creation of new buckets for each new directory entry. All the new directory entries except one share buckets with the old directory entries. Whenever a bucket which is being shared by two or more directory entries is to be split the directory size need not be doubled. This means for each bucket we need to know whether it is being shared by two or more directory entries. This information is provided by the *local depth* of the bucket. The same information can be obtained by a scan of the directory, but this is costlier.

2. Exactly two directory entries have only one directory entry pointing to them after a doubling of the directory size. This is because when the directory is doubled, one of the buckets must have split causing a directory entry to point to each of these two new buckets.

   If an entry is then deleted from one of these buckets, a merge may occur, but this depends on the deletion algorithm. If we try to merge two buckets only when a bucket becomes empty, then it is not necessary that the directory size decrease after the deletion that was considered in the question. However, if we try to merge two buckets whenever it is possible to do so then the directory size decreases after the deletion.

3. No "minimum disk access" guarantee is provided by extendible hashing. If the directory is not already in memory it needs to be fetched from the disk which may require more than one disk access depending upon the size of the directory. Then the required bucket has to be brought into the memory. Also, if alternatives 2 and 3 are followed for storing the data entries in the index then another disk access is possibly required for fetching the actual data record.

4. Consider the index in Fig 11.1. Let us consider a list of data entries with search key values of the form 2^i where $i > k$. By an appropriate choice of k, we can get all these elements mapped into the *Bucket A*. This creates 2^k elements in the directory which point to just $k + 3$ different buckets. Also, we note there are k buckets (data pages), but just one bucket is used. So the utilization of data pages $= 1/k$.

5. No. Since we are using extendible hashing, only the local depth of the bucket being split needs be examined.

6. Extendible hashing is not supposed to have overflow pages (overflow pages are supposed to be dealt with using redistribution and splitting). When there are many duplicate entries in the index, overflow pages may be created that can never be redistributed (they will always map to the same bucket). Whenever a "split" occurs on a bucket containing only duplicate entries, an empty bucket will be created since all of the duplicates remain in the same bucket. The overflow chains will never be split, which makes inserts and searches more costly.

Exercise 11.4 Answer the following questions about Linear Hashing:

1. How does Linear Hashing provide an average-case search cost of only slightly more than one disk I/O, given that overflow buckets are part of its data structure?

2. Does Linear Hashing guarantee at most one disk access to retrieve a record with a given key value?

3. If a Linear Hashing index using Alternative (1) for data entries contains N records, with P records per page and an average storage utilization of 80 percent, what is the worst-case cost for an equality search? Under what conditions would this cost be the actual search cost?

4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the space utilization in data pages?

Answer 11.4 The answer to each question is given below.

1. If we start with an index which has B buckets, during the round all the buckets will be split in order, one after the other. A hash function is expected to distribute the search key values uniformly in all the buckets. This kind of split during the round causes a redistribution of key values in all the buckets. If a bucket has overflow pages, after the redistribution it is likely that the length of the overflow chain reduces. If the hash function is good, the length of the overflow chains in most buckets is zero because in each round there will be at least one redistribution of the values in each bucket. The number of overflow pages during the round is not expected to go beyond one because the hash function distributes the incoming entries uniformly.

2. No. Overflow chains are part of the structure, so no such guarantees are provided.

3. $$\frac{N}{0.8P}$$

   This can be achieved when all the keys map into the same bucket. (The effect of 80% occupancy is to increase the number of pages in the file, relative to a file with 100% occupancy.)

4. Consider the index in Fig 11.6. Let us consider a list of data entries with search key values of the form 2^i where $i > k$. By an appropriate choice of k, we can get all these elements mapped into $Bucket0$. Suppose we have m primary data pages, each time we need to add one more overflow page to $Bucket0$, it will cause a page split. So if we add n overflow pages to $Bucket0$, the space utilization $= (n+1)/(m+n+n)$, which is less than 50%.

Exercise 11.5 Give an example of when you would use each element (A or B) for each of the following 'A versus B' pairs:

1. A hashed index using Alternative (1) versus heap file organization.

2. Extendible Hashing versus Linear Hashing.

3. Static Hashing versus Linear Hashing.
4. Static Hashing versus ISAM.
5. Linear Hashing versus B+ trees.

Answer 11.5 The answer to each question is given below.

1. **Example 1:** Consider a situation in which most of the queries are equality queries based on the search key field. It pays to build a hashed index on this field in which case we can get the required record in one or two disk accesses. A heap file organisation may require a full scan of the file to access a particular record.
   Example 2: Consider a file on which only sequential scans are done. It may fare better if it is organised as a heap file. A hashed index built on it may require more disk accesses because the occupancy of the pages may not be 100%.
2. **Example 1:** Consider a set of data entries with search keys which lead to a skewed distribution of hash key values. In this case, extendible hashing causes splits of buckets at the necessary bucket whereas linear hashing goes about splitting buckets in a round-robin fashion which is useless. Here extendible hashing has a better occupancy and shorter overflow chains than linear hashing. So equality search is cheaper for extendible hashing.
   Example 2: Consider a very large file which requires a directory spanning several pages. In this case extendible hashing requires $d + 1$ disk accesses for equality selections where d is the number of directory pages. Linear hashing is cheaper.
3. **Example 1:** Consider a situation in which the number of records in the file is constant. Let all the search key values be of the form $2^n + k$ for various values of n and a few values of k. The traditional hash functions used in *linear hashing* like taking the last d bits of the search key lead to a skewed distribution of the hash key values. This leads to long overflow chains. A static hashing index can use the hash function defined as
   $$h(2^n + k) = n$$
   A family of hash functions can't be built based on this hash function as k takes only a few values. In this case static hashing is better.
   Example 2: Consider a situation in which the number of records in the file varies a lot and the hash key values have a uniform distribution. Here linear hashing is clearly better than static hashing which might lead to long overflow chains thus considerably increasing the cost of equality search.
4. **Example 1:** Consider a situation in which the number of records in the file is constant and only equality selections are performed. Static hashing requires one or two disk accesses to get to the data entry. ISAM may require more than one depending on the height of the ISAM tree.
   Example 2: Consider a situation in which the search key values of data entries

can be used to build a clustered index and most of the queries are range queries on this field. Then ISAM definitely wins over static hashing.

5. **Example 1:** Again consider a situation in which only equality selections are performed on the index. Linear hashing is better than B+ tree in this case.
   Example 2: When an index which is clustered and most of the queries are range searches, B+ indexes are better.

Exercise 11.6 Give examples of the following:

1. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Linear Hashing index has more pages.
2. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Extendible Hashing index has more pages.

Answer 11.6 1. Let us take the data entries

$$8, 16, 24, 32, 40, 48, 56, 64, 128, 7, 15, 31, 63, 127, 1, 10, 4$$

and the indexes shown in Fig 11.10 and Fig 11.11. Extendible hashing uses 9 pages including the directory page(assuming it spans just one page) and linear hashing uses 10 pages.

2. Consider the list of data entries

$$0, 4, 1, 5, 2, 6, 3, 7$$

and the usual hash functions for both and a page capacity of 4 records per page. Extendible hashing takes 4 data pages and also a directory page whereas linear hashing takes just 4 pages.

Exercise 11.7 Consider a relation R(*a, b, c, d*) containing 1 million records, where each page of the relation holds 10 records. R is organized as a heap file with unclustered indexes, and the records in R are randomly ordered. Assume that attribute a is a candidate key for R, with values lying in the range 0 to 999,999. For each of the following queries, name the approach that would most likely require the fewest I/Os for processing the query. The approaches to consider follow:

- Scanning through the whole heap file for R.
- Using a B+ tree index on attribute R.a.
- Using a hash index on attribute R.a.

The queries are:

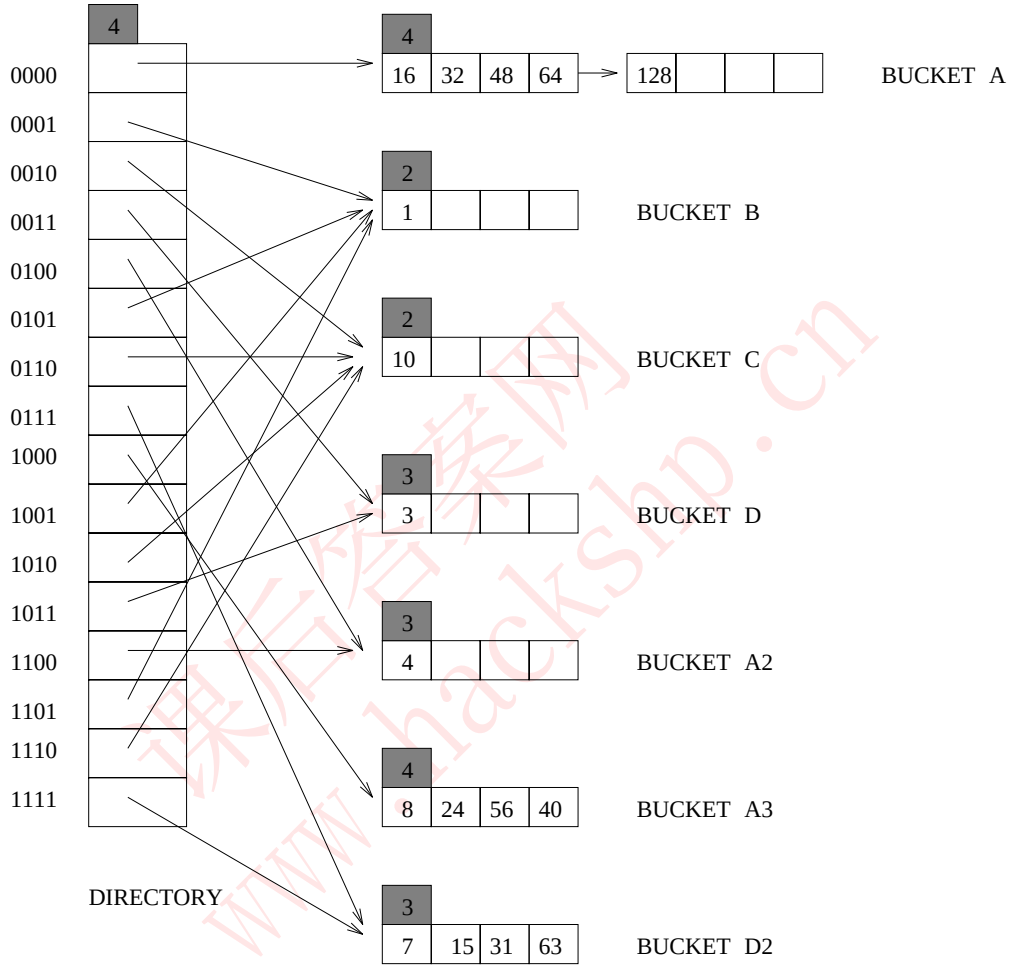


Figure 11.10

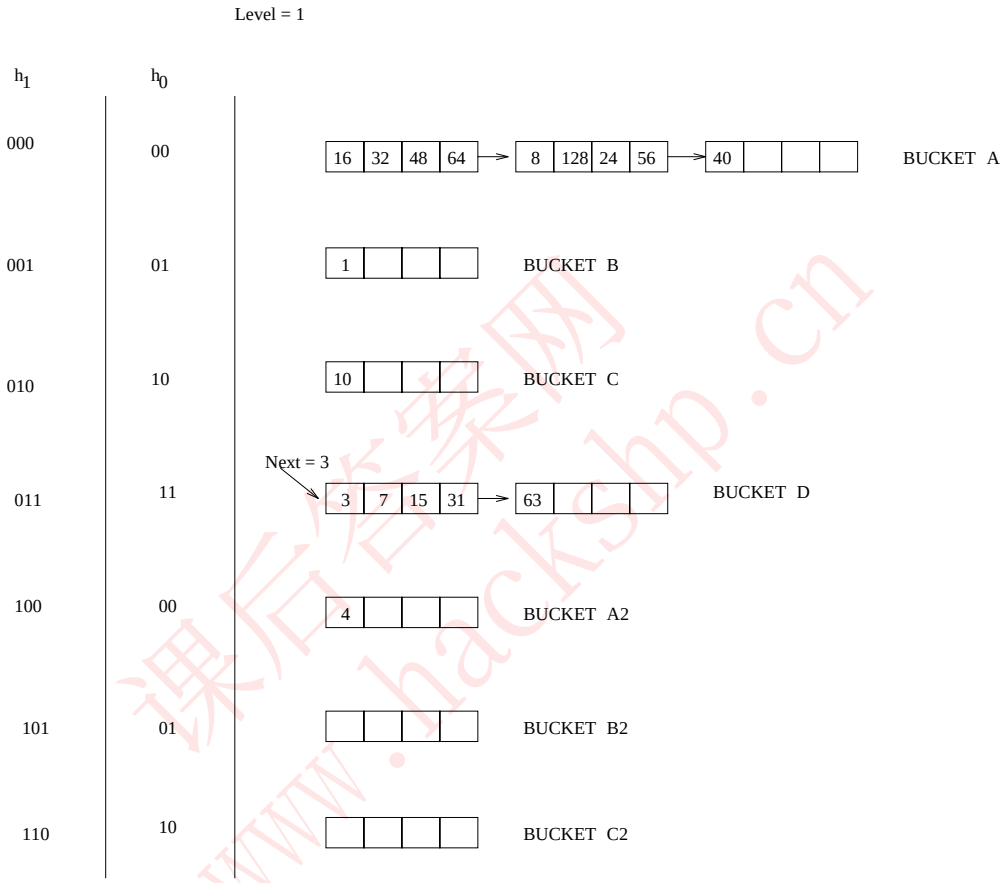


Figure 11.11

1. Find all R tuples.
2. Find all R tuples such that $a < 50$.
3. Find all R tuples such that $a = 50$.
4. Find all R tuples such that $a > 50$ and $a < 100$.

Answer 11.7 Let h be the height of the B+ tree (usually 2 or 3) and M be the number of data entries per page ($M > 10$). Let us assume that after accessing the data entry it takes one more disk access to get the actual record. Let c be the occupancy factor in hash indexing.

Consider the table shown below (disk accesses):

Problem	Heap File	B+ Tree	Hash Index
1. All tuples	10^5	$h + \frac{10^6}{M} + 10^6$	$\frac{10^6}{cM} + 10^6$
2. $a < 50$	10^5	$h + \frac{50}{M} + 50$	100
3. $a = 50$	10^5	$h + 1$	2
4. $a > 50$ and $a < 100$	10^5	$h + \frac{50}{M} + 49$	98

1. From the first row of the table, we see that heap file organization is the best (has the fewest disk accesses).
2. From the second row of the table, with typical values for h and M, the B+ Tree has the fewest disk accesses.
3. From the third row of the table, hash indexing is the best.
4. From the fourth row or the table, again we see that B+ Tree is the best.

Exercise 11.8 How would your answers to Exercise 11.7 change if a is not a candidate key for R? How would they change if we assume that records in R are sorted on a?

Answer 11.8 If attribute a is not a candidate key for R, the result will be changed depending on selectivity of attribute a.

Problem	Heap File	B+ Tree	Hash Index
1. All tuples	10^5	$h + \frac{10^6}{M} + 10^6$	$\frac{10^6}{cM} + 10^6$
2. $a < 50$	10^5	$h + \frac{50sel}{M} + 50sel$	$50 + 50sel$
3. $a = 50$	10^5	$h + sel$	$1 + sel$
4. $a > 50$ and $a < 100$	10^5	$h + \frac{50sel}{M} + 49sel$	$49 + 49sel$

1. From the first row of the table, heap file organisation is still the best.

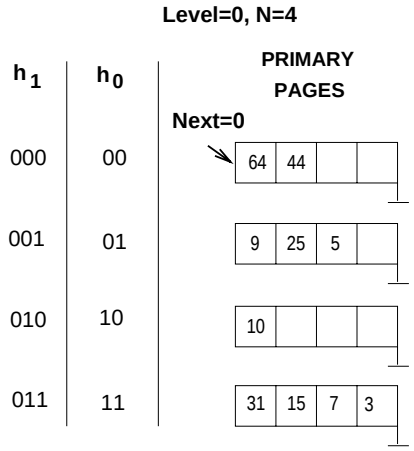


Figure 11.12 Figure for Exercise 11.9

2. From the second row, we are not sure which one is the best: the answer depends on the selectivity of a.
3. From the third row, hash indexing is the best.
4. From the fourth row, again we are not sure which is the best: the answer depends on the selectivity of a.

Problem	Heap File	B+ Tree	Hash Index
1. All tuples	10^5	$h + \frac{10^6}{M} + 10^6$	$\frac{10^6}{cM} + 10^6$
2. $a < 50$	5	$h + \frac{50}{M} + 5$	$50 + 5$
3. $a = 50$	$log_2(10^5)$	$h + 1$	2
4. $a > 50$ and $a < 100$	$log_2(10^5) + 5$	$h + \frac{50}{M} + 49$	98

The answers can be seen from the table.

Exercise 11.9 Consider the snapshot of the Linear Hashing index shown in Figure 11.12. Assume that a bucket split occurs whenever an overflow page is created.

1. What is the *maximum* number of data entries that can be inserted (given the best possible distribution of keys) before you have to split a bucket? Explain very briefly.
2. Show the file after inserting a *single* record whose insertion causes a bucket split.
3. (a) What is the *minimum* number of record insertions that will cause a split of all four buckets? Explain very briefly.

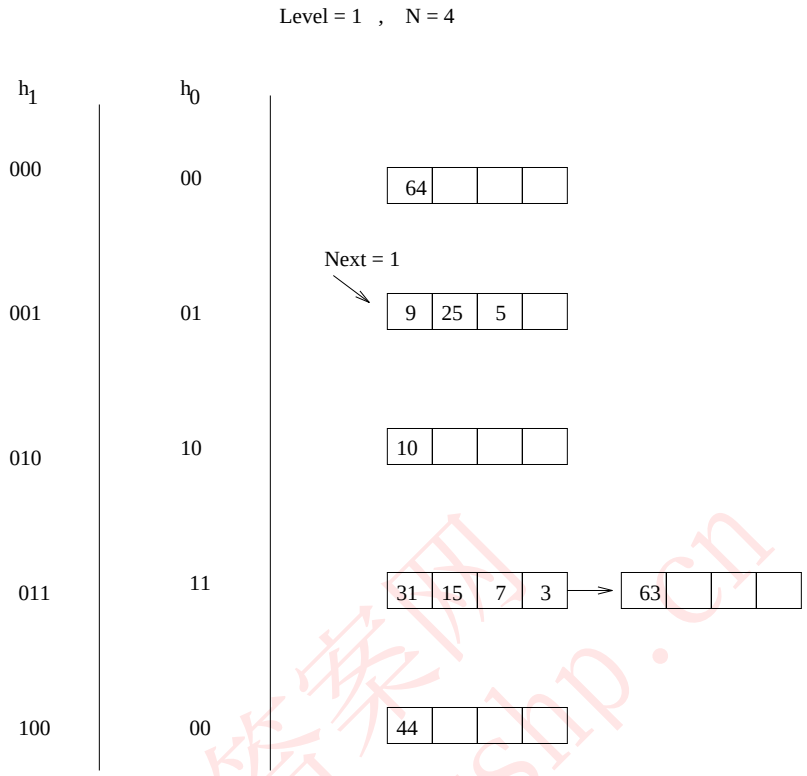


Figure 11.13

(b) What is the value of *Next* after making these insertions?

(c) What can you say about the number of pages in the fourth bucket shown after this series of record insertions?

Answer 11.9 The answer to each question is given below.

1. The maximum number of entries that can be inserted without causing a split is 6 because there is space for a total of 6 records in all the pages. A split is caused whenever an entry is inserted into a full page.

2. See Fig 11.13

3. (a) Consider the list of insertions $63, 41, 73, 137$ followed by 4 more entries which go into the same bucket, say $18, 34, 66, 130$ which go into the 3rd bucket. The insertion of 63 causes the first bucket to be split. Insertion of $41, 73$ causes the second bucket split leaving a full second bucket. Inserting 137 into it causes third bucket-split. At this point at least 4 more entries are required

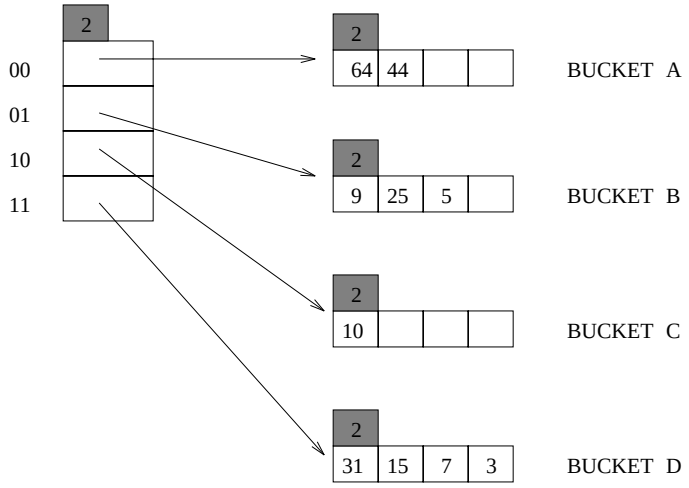


Figure 11.14

to split the fourth bucket. A minimum of 8 entries are required to cause the 4 splits.

(b) Since all four buckets would have been split, that particular round comes to an end and the next round begins. So $Next = 0$ again.

(c) There can be either one data page or two data pages in the fourth bucket after these insertions. If the 4 more elements inserted into the 2^{nd} bucket after 3^{rd} bucket-splitting, then 4^{th} bucket has 1 data page.

If the new 4 more elements inserted into the 4^{th} bucket after 3^{rd} bucket-spliting and all of them have 011 as its last three bits, then 4^{th} bucket has 2 data pages. Otherwise, if not all have 011 as its last three bits, then the 4^{th} bucket has 1 data page.

Exercise 11.10 Consider the data entries in the Linear Hashing index for Exercise 11.9.

1. Show an Extendible Hashing index with the same data entries.
2. Answer the questions in Exercise 11.9 with respect to this index.

Answer 11.10 An Extendible Hashing index with the same entries as Exercise 11.9 can be seen in Fig 11.14.

1. Six entries, for the same reason as in question 11.9.
2. See Fig 11.15

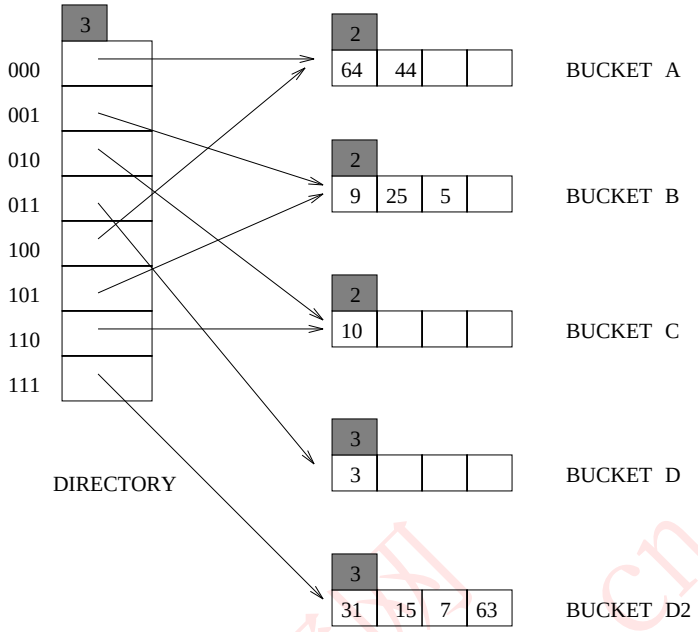


Figure 11.15

3. (a) 10. A bucket is split in extendible hashing only if it is full and a new entry is to be inserted into it.
   (b) The next pointer is not applicable in Extendible Hashing.
   (c) 1 page. Extendible hashing is not supposed to have overflow pages.

Exercise 11.11 In answering the following questions, assume that the full deletion algorithm is used. Assume that merging is done when a bucket becomes empty.

1. Give an example of Extendible Hashing where deleting an entry reduces global depth.
2. Give an example of Linear Hashing in which deleting an entry decrements *Next* but leaves *Level* unchanged. Show the file before and after the deletion.
3. Give an example of Linear Hashing in which deleting an entry decrements *Level*. Show the file before and after the deletion.
4. Give an example of Extendible Hashing and a list of entries e_1, e_2, e_3 such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

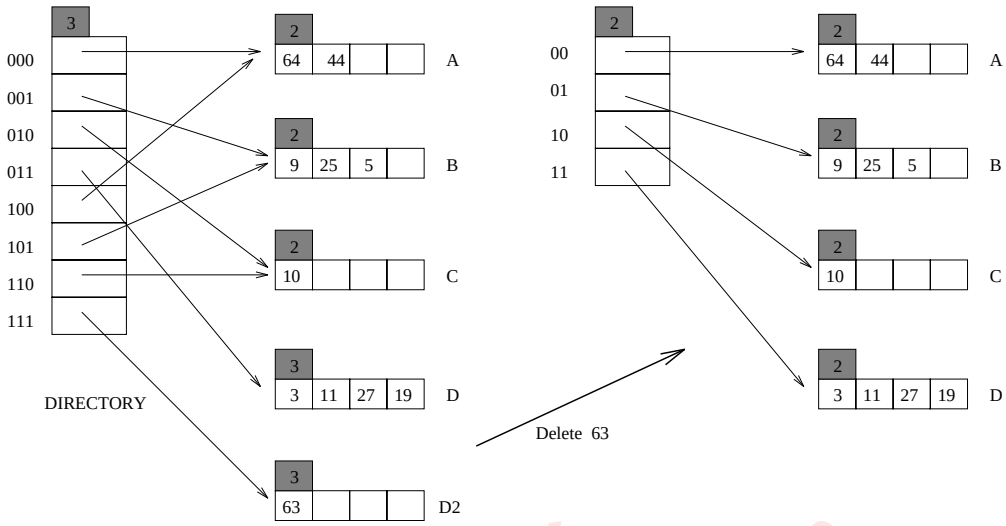


Figure 11.16

5. Give an example of a Linear Hashing index and a list of entries e_1, e_2, e_3 such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

Answer 11.11 The answers are as follows.

1. See Fig 11.16
2. See Fig 11.17
3. See Fig 11.18
4. Let us take the transition shown in Fig 11.19. Here we insert the data entries 4, 5 and 7. Each one of these insertions causes a split with the initial split also causing a directory split. But none of these insertions redistribute the already existing data entries into the new buckets. So when we delete these data entries in the reverse order (actually the order doesn't matter) and follow the full deletion algorithm we get back the original index.
5. This example is shown in Fig 11.20. Here the idea is similar to that used in the previous answer except that the bucket being split is the one into which the insertion being made. So bucket 2 has to be split and not bucket 3. Also the order of deletions should be exactly reversed because in the deletion algorithm Next is decremented only if the last bucket becomes empty.

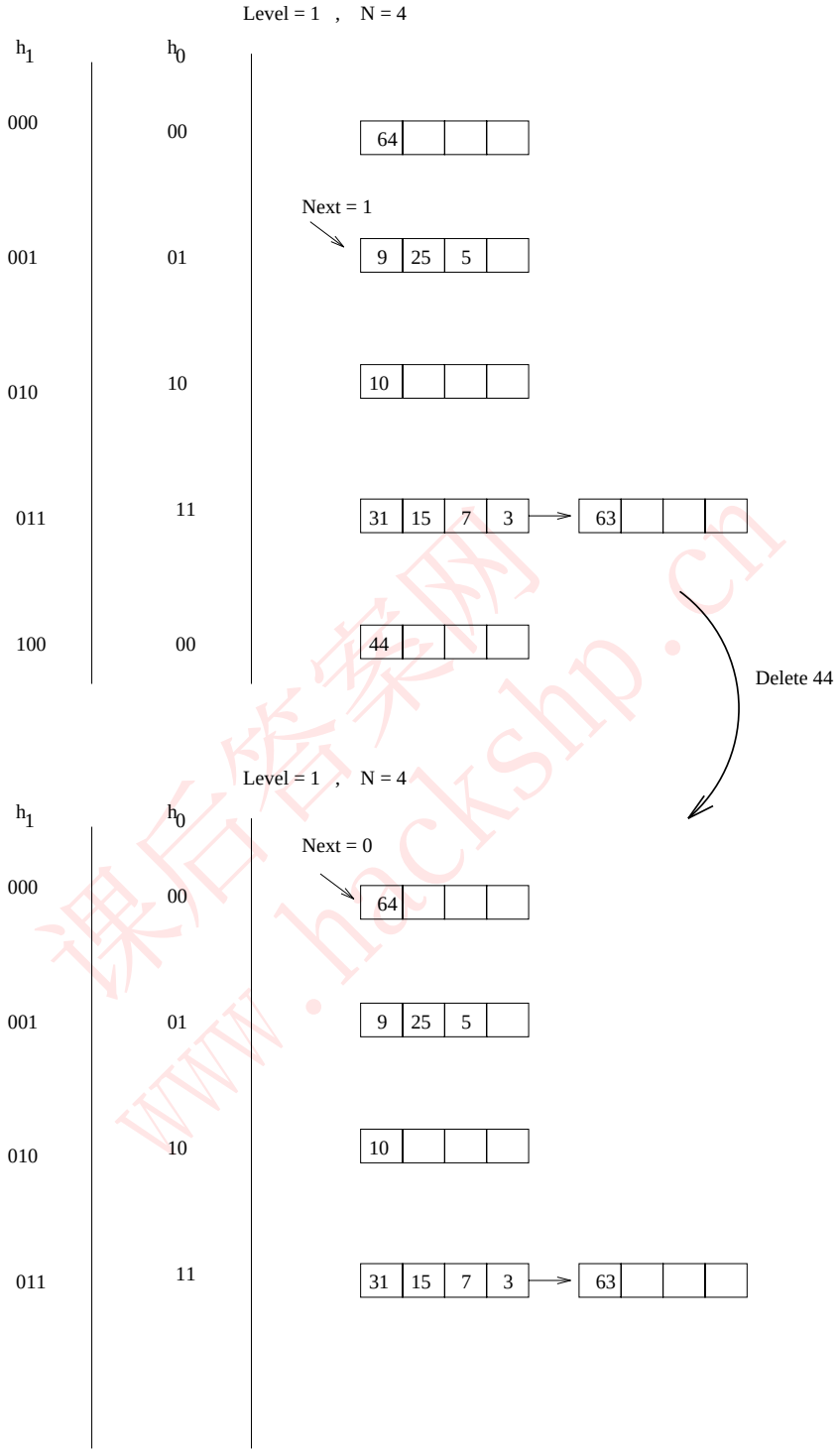


Figure 11.17

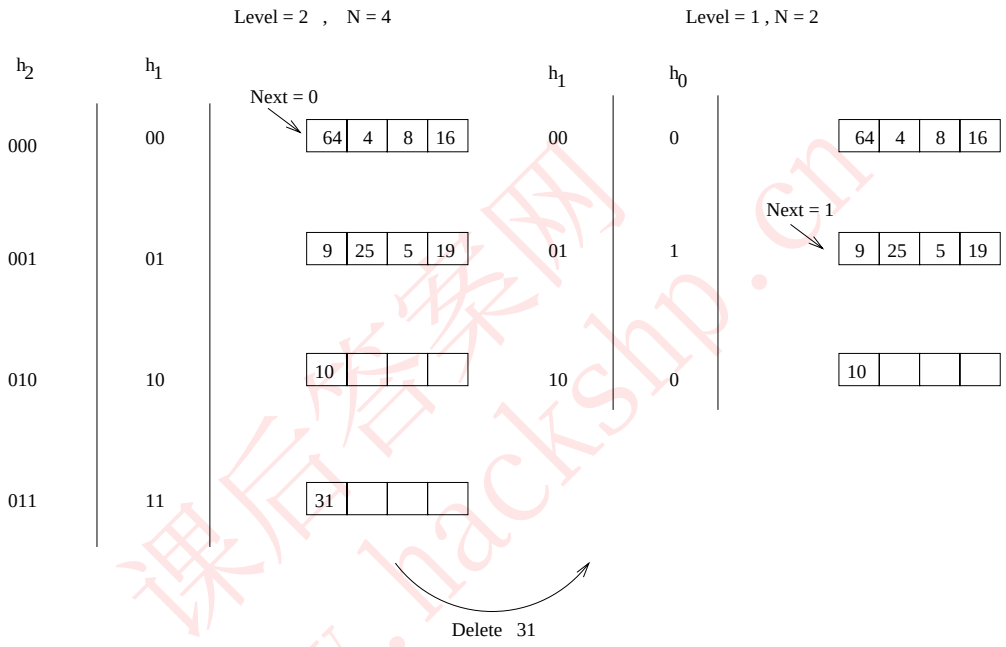


Figure 11.18

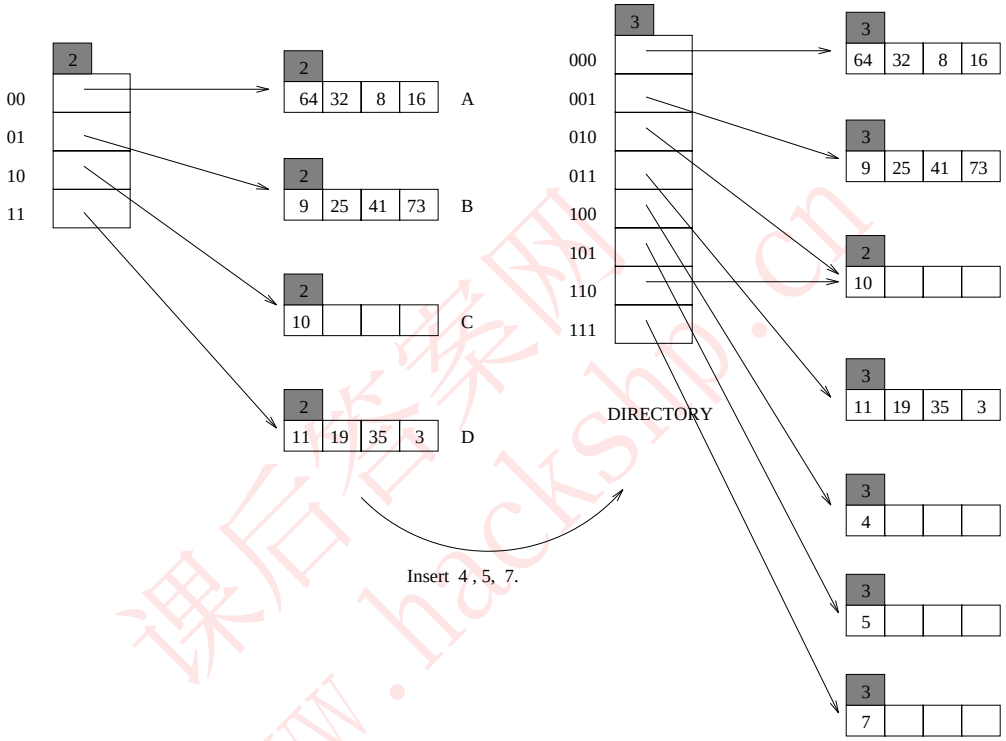


Figure 11.19

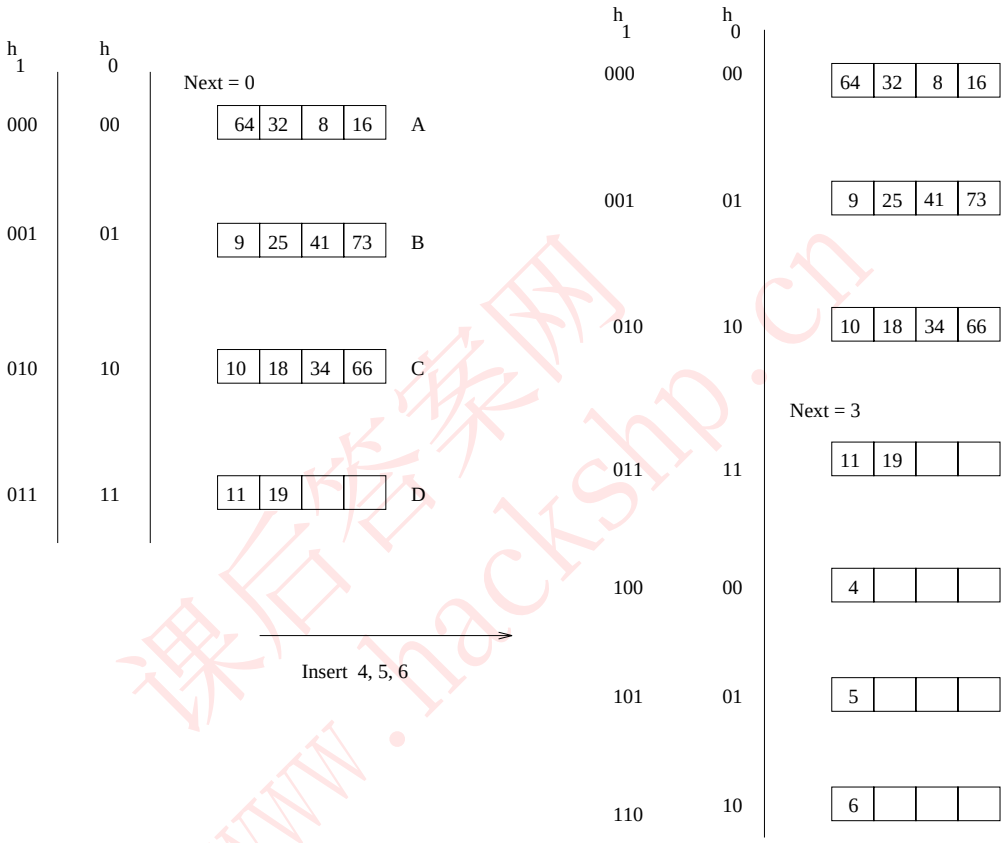


Figure 11.20

12

OVERVIEW OF QUERY EVALUATION

Exercise 12.1 Briefly answer the following questions:

1. Describe three techniques commonly used when developing algorithms for relational operators. Explain how these techniques can be used to design algorithms for the selection, projection, and join operators.
2. What is an access path? When does an index *match* an access path? What is a *primary conjunct*, and why is it important?
3. What information is stored in the system catalogs?
4. What are the benefits of storing the system catalogs as relations?
5. What is the goal of query optimization? Why is optimization important?
6. Describe *pipelining* and its advantages.
7. Give an example query and plan in which pipelining *cannot* be used.
8. Describe the *iterator* interface and explain its advantages.
9. What role do statistics gathered from the database play in query optimization?
10. What were the important design decisions made in the System R optimizer?
11. Why do query optimizers consider only left-deep join trees? Give an example of a query and a plan that would not be considered because of this restriction.

Answer 12.1 The answer to each question is given below.

1. The three techniques commonly used are indexing, iteration, and partitioning:
   - **Indexing:** If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition.

- **Iteration:** Examine all tuples in an input table, one after the other. If we need only a few fields from each tuple and there is an index whose key contains all these fields, instead of examining data tuples, we can scan all index data entries.
- **Partitioning:** By partitioing tuples on a sort key, we can often decompose an operation into a less expensive collection of operations on partitions. *Sorting* and *hashing* are two commonly used partitioning techniques.

They can be used to design algorithms for selection, projection, and join operators as follows:

- **Selection:** For a selection with more than one tuple matching the query (in general, at least 5%), indexing like B+ Trees are very useful. This comes into play often with range queries. It allows us to not only find the first qualifying tuple quickly, but also the other qualifying tuples soon after (especially if the index is clustered). For a selection condition with an equality query where there are only a few (usually 1) matching tuple, partitioning using hash indexing is often appropriate. It allows us to find the exact tuple we are searching for with a cost of only a few (typically one or two) I/Os.
- **Projection:** The projection operation requires us to drop certain fields of the input, which can result in duplicates appearing in the result set. If we do not need to remove these duplicates, then the iteration technique can efficiently handle this problem. On ther other hand, if we do need to eliminate duplicates, partitioning the data and applying a sort key is typically performed.

  In the case that there are appropriate indexes available, this can lead to less expensive plans for sorting the tuples during duplicate elimination since the data may all ready be sorted on the index (in that case we simply compare adjacent entries to check for duplicates)
- **Join:** The most expensive database operation, joins, can combinations of all three techniques. A join operation typically has multiple selection and projection elements built into it, so the importance of having appropriate indexes or of partitioning the data is just as above, if not more so. When possible, the individual selections and projections are applied to two relations before they are joined, so as to decrease the size of the intermediate table.

  As an example consider joining two relations with 100,000 tuples each and only 5 % of qualifying tuples in each table. Joining before applying the selection conditions, would result in a huge intermediate table size that would then have to be searched for matching selections. Alternatively, consider applying parts of the selection first. We can then perform a join of the 5,000 qualifying tuples found after applying the selection to each table, that can then be searched and handled significantly faster.

2. An *access path* is a way of retrieving tuples from a table and consists of either a file scan or an index plus a *matching* selection condition. An index *matches* a selection condition if the index can be used to retrieve just the tuples that satisfy the condition. An index can match some subset of conjunts in a selection condition even though it does not match the entire condition and we refer to the conjuct that the index matches as the *primary conjuncts* in the selection. Primary conjuncts are important because they allow us to quickly discard information we do not need and only focus in on searching/sorting the data that more closely matches the selection conditions.

3. Information about relations, indexes, and views is stored in the system catalogs. This includes file names, file sizes, and file structure, the attribute names and data types, lists of keys, and constraints.

   Some commonly stored statistical information includes:

   (a) Cardinality - the number of tuples for each relation

   (b) Size - the number of pages in each relation

   (c) Index Cardinality - the number of distinct key values for each index

   (d) Index Size - the number of pages for each index (or number of leaf pages)

   (e) Index Height - the number of nonleaf levels for each tree index

   (f) Index Range - the minimum present key value and the maximum present key value for each index.

4. There are several advantages to storing the system catatlogs as relations. Relational system catalogs take advantage of all of the implementation and management benefits of relational tables: effective information storage and rich querying capabilities. The choice of what system catalogs to maintain is left to the DBMS implementor.

5. The goal of query optimization is to avoid the worst plans and find a good plan. The goal is usually not to find the optimal plan. The difference in cost between a good plan and a bad plan can be several orders of magnitude: a good query plan can evaluate the query in seconds, whereas a bad query plan might take days!

6. Pipelining allows us to avoid creating and reading temporary relations; the I/O savings can be substantial.

7. Bushy query plans often cannot take advantage of pipelining because of limited buffer or CPU resources. Consider a bushy plan in which we are doing a selection on two relations, followed by a join. We cannot always use pipelining in this strategy becuase the result of the selection on the first selection may not fit in memory, and we must wait for the second relation's selection to complete before we can begin the join.

8. The iterator interface for an operator includes the functions *open*, *get_next*, and *close*; it hides the details of how the operator is implemented, and allows us to view all operator nodes in a query plan uniformly.

9. The query optimizer uses statistics to improve the chances of selecting an optimum query plan. The statistics are used to calculate reduction factors which determine the results the optimizer may expect given different indexes and inputs.

10. Some important design decisions in the System R optimizer are:

    (a) Using statistics about a database instance to estimate the cost of a query evaluation plan.

    (b) A decision to consider only plans with binary joins in which the inner plan is a base relation. This heuristic reduces the often significant number of alternative plans that must be considered.

    (c) A decision to focus optimization on the class of SQL queries without nesting and to treat nested queries in a relatively ad hoc way.

    (d) A decision not to perform duplicate elimination for projections (except as a final step in the query evaluation when required by a `DISTINCT` clause).

    (e) A model of cost that accounted for CPU costs as wellas I/O costs.

11. There are two main reasons for the decision to concentrate on left-deep plains only:

    (a) As the number of joins increases, the number of alternative plans increases rapidly and it becomes neccessary to prune the space of the alternative plans.

    (b) Left-deep trees allow us to generate all fully piplined plans; that is, plans in which all joins are evaluated using pipelining.

    Consider the join A $\bowtie$ B $\bowtie$ C $\bowtie$ D. The query plan (A $\bowtie$ B) $\bowtie$ (C $\bowtie$ D) would never be considered because it is a bushy tree.

Exercise 12.2 Consider a relation R(*a,b,c,d,e*) containing 5,000,000 records, where each data page of the relation holds 10 records. R is organized as a sorted file with secondary indexes. Assume that *R.a* is a candidate key for R, with values lying in the range 0 to 4,999,999, and that R is stored in *R.a* order. For each of the following relational algebra queries, state which of the following three approaches is most likely to be the cheapest:

- Access the sorted file for R directly.
- Use a (clustered) B+ tree index on attribute *R.a*.
- Use a linear hashed index on attribute *R.a*.

1. $\sigma_{a<50,000}(R)$
2. $\sigma_{a=50,000}(R)$
3. $\sigma_{a>50,000 \wedge a<50,010}(R)$
4. $\sigma_{a\neq 50,000}(R)$

Answer 12.2 The answer to each question is given below.

1. $\sigma_{a<50,000}(R)$ - For this selection, the choice of accessing the sorted file is slightly superior in cost to using the clused B+ tree index simply because of the lookup cost required on the B+ tree.
2. $\sigma_{a=50,000}(R)$ - A linear hashed index should be cheapest here.
3. $\sigma_{a>50,000 \wedge a<50,010}(R)$ - A B+ tree should be the cheapest of the three.
4. $\sigma_{a\neq 50,000}(R)$ - Since the selection will require a scan of the available entries, and we're starting at the beginning of the sorted index, accessing the sorted file should be slightly more cost-effective, again because of the lookup time.

Exercise 12.3 For each of the following SQL queries, for each relation involved, list the attributes that must be examined to compute the answer. All queries refer to the following relations:

Emp(*eid:* `integer`, *did:* `integer`, *sal:* `integer`, *hobby:* `char(20)`)
Dept(*did:* `integer`, *dname:* `char(20)`, *floor:* `integer`, *budget:* `real`)

1. `SELECT * FROM` Emp E
2. `SELECT * FROM` Emp E, Dept D
3. `SELECT * FROM` Emp E, Dept D `WHERE` E.did = D.did
4. `SELECT` E.eid, D.dname `FROM` Emp E, Dept D `WHERE` E.did = D.did

Answer 12.3 The answer to each question is given below.

1. E.eid, E.did, E.sal, E.hobby
2. E.eid, E.did, E.sal, E.hobby, D.did, D.dname, D.floor, D.budget
3. E.eid, E.did, E.sal, E.hobby, D.did, D.dname, D.floor, D.budget
4. E.eid, D.dname, E.did, D.did

Exercise 12.4 Consider the following schema with the Sailors relation:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)

For each of the following indexes, list whether the index matches the given selection conditions. If there is a match, list the primary conjuncts.

1. A B+-tree index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
2. A hash index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
3. A B+-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid<50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$
4. A hash-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$

Answer 12.4 The answer to each question is given below.

1. (a) Match. Primary conjuncts are: *Sailors.sid < 50,000*
   (b) Match. Primary conjuncts are: *Sailors.sid = 50,000*
2. (a) No Match. Range queries cannot be applied to hash indexes.
   (b) Match. Primary conjunct are: *Sailors.sid = 50,000*
3. (a) Match. Primary conjunct are: *Sailors.sid < 50,000* and *Sailors.sid < 50,000 ∧ Saliors.age = 21*

(b) Match. Primary conjunct are: *Sailors.sid = 50,000* and *Sailors.sid = 50,000 ∧ Saliors.age > 21*

(c) Match. Primary conjunct are: *Sailors.sid = 50,000*

(d) No Match. The index on ⟨ Sailors.sid, Sailors.age ⟩ is primarily sorted on Sailors.sid, therefore the entire relation would need to be searched to find those with a particular Sailors.age value.

4. (a) Match. Primary conjunct are: *Sailors.sid = 50,000* and *Sailors.sid = 50,000 ∧ Saliors.age = 21*

(b) Match. Primary conjunct are: *Sailors.sid = 50,000*

(c) Match. Primary conjunct are: *Sailors.sid = 50,000*

(d) No Match. The index on ⟨ Sailors.sid, Sailors.age ⟩ does not allow us to retrieve sets of sailors with age equal to 21.

Exercise 12.5 Consider again the schema with the Sailors relation:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)

Assume that each tuple of Sailors is 50 bytes long, that a page can hold 80 Sailors tuples, and that we have 500 pages of such tuples. For each of the following selection conditions, estimate the number of pages retrieved, given the catalog information in the question.

1. Assume that we have a B+-tree index T on the search key ⟨ Sailors.sid ⟩, and assume that $IHeight(T) = 4$, $INPages(T) = 50$, $Low(T) = 1$, and $High(T) = 100{,}000$.

   (a) $\sigma_{Sailors.sid<50{,}000}(Sailors)$

   (b) $\sigma_{Sailors.sid=50{,}000}(Sailors)$

2. Assume that we have a hash index T on the search key ⟨ Sailors.sid ⟩, and assume that $IHeight(T) = 2$, $INPages(T) = 50$, $Low(T) = 1$, and $High(T) = 100{,}000$.

   (a) $\sigma_{Sailors.sid<50{,}000}(Sailors)$

   (b) $\sigma_{Sailors.sid=50{,}000}(Sailors)$

Answer 12.5 The answer to each question is given below.

1. (a) Assuming uniform distribution, around half of the 40,000 tuples will match the selection condition. The total cost is then the cost of finding the first leaf node (4 I/O's) plus the cost of retrieving the matching tuples. If the index is clustered, then the total cost is 4 + 250 I/O's = 254 I/O's. If the index is

unclustered, then the cost of retrieving each matching tuple could be as high as 20,000 I/O's (one I/O for each matching tuple), leading to a total cost of 20,004 I/O's. If the index is unclustered, then doing a file scan is most likely the preferred method with a cost of 500 I/O's.

(b) Since *sid* is a primary key for the relation we expect only one matching tuple for the hash index, therefore the cost is just the height of the tree (4 I/O's) plus the cost of reading the qualifying tuple's page (1 I/O) which adds up to be 5 I/O's.

2. (a) Since a hash index on *sid* cannot help us for range queries, the index is useless, and therefore we must do a file scan at a cost of 500 pages I/O's (the cost of reading the entire relation).

(b) Since *sid* is a primary key for the relation we expect only one matching tuple for the hash index, therefore the cost is just the height of the tree (2 I/O's) plus the cost of reading the qualifying tuple's page (1 I/O) which adds up to be 3 I/O's.

Exercise 12.6 Consider the two join methods described in Section 12.3.3. Assume that we join two relations R and S, and that the systems catalog contains appropriate statistics about R and S. Write formulas for the cost estimates of the index nested loops join and sort-merge join using the appropriate variables from the systems catalog in Section 12.1. For index nested loops join, consider both a B+ tree index and a hash index. (For the hash index, you can assume that you can retrieve the index page containing the rid of the matching tuple with 1.2 I/Os on average.)

Answer 12.6 The answer to each part of the question is given below. Note that for the Index Nested Loop Joins, we only consider indexes on *Sailors.sid* since *sid* is a foriegn key reference in Reserves that matches up with exactly one Sailor tuple. In addition, we assume the typical cost of a lookup is 1.2 I/O's for hash-based indexes and 3 I/O's for B+ tree indexes.

- **Index Nested Loop Joins (Clustered B+ Tree Index on Sailors):**
  = Cost of Scanning R + Cost of Retrieving for each R an Index in S + Cost of Retrieving each matching tuple in S
  = NPages(R) + 3*NTuples(R) + NTuples(R) I/O's
  = NPages(R) + 4*NTuples(R) I/O's

- **Index Nested Loop Joins (Unclustered B+ Tree Index on Sailors):**
  = Cost of Scanning R + Cost of Retrieving for each R an Index in S + Cost of Retrieving each matching tuple in S
  = NPages(R) + 3*NTuples(R) + [NTuples(R)/NTuple(S)]*NTuples(R) I/O's
  = NPages(R) + [3 + [NTuples(R)/NTuple(S)]] * NTuples(R) I/O's

- **Index Nested Loop Joins (Hash Index on Sailors):**
  = Cost of Scanning R + Cost of Retrieving for each R an Index in S + Cost of Retrieving each matching tuple in S
  = NPages(R) + 1.2*NTuples(R) + NTuples(R) I/O's
  = NPages(R) + 2.2*NTuples(R) I/O's

- **Sort-Merge Joins:**
  = Cost of Sorting R + Cost of Sorting S + Cost of Scanning R and S
  = 4*NPages(R) + 4*NPages(S) + [NPages(R) + NPages(S)] I/O's
  = 5 * [NPages(R) + NPages(S)] I/O's

13

EXTERNAL SORTING

Exercise 13.1 Suppose you have a file with 10,000 pages and you have three buffer pages. Answer the following questions for each of these scenarios, assuming that our most general external sorting algorithm is used:

(a) A file with 10,000 pages and three available buffer pages.

(b) A file with 20,000 pages and five available buffer pages.

(c) A file with 2,000,000 pages and 17 available buffer pages.

1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

Answer 13.1 The answer to each question is given below.

1. In the first pass (Pass 0), $\lceil N/B \rceil$ runs of B pages each are produced, where N is the number of file pages and B is the number of available buffer pages:
   (a) $\lceil 10000/3 \rceil = 3334$ sorted runs.
   (b) $\lceil 20000/5 \rceil = 4000$ sorted runs.
   (c) $\lceil 2000000/17 \rceil = 117648$ sorted runs.
2. The number of passes required to sort the file completely, including the initial sorting pass, is $\lceil log_{B-1} N1 \rceil + 1$, where $N1 = \lceil N/B \rceil$ is the number of runs produced by Pass 0:
   (a) $\lceil log_2 3334 \rceil + 1 = 13$ passes.
   (b) $\lceil log_4 4000 \rceil + 1 = 7$ passes.
   (c) $\lceil log_{16} 117648 \rceil + 1 = 6$ passes.

3. Since each page is read and written once per pass, the total number of page I/Os for sorting the file is $2 * N * (\#passes)$:
   (a) 2*10000*13 = 260000.
   (b) 2*20000*7 = 280000.
   (c) 2*2000000*6 = 24000000.

4. In Pass 0, $\lceil N/B \rceil$ runs are produced. In Pass 1, we must be able to merge this many runs; i.e., $B - 1 \geq \lceil N/B \rceil$. This implies that B must at least be large enough to satisfy $B * (B - 1) \geq N$; this can be used to guess at B, and the guess must be validated by checking the first inequality. Thus:
   (a) With 10000 pages in the file, $B = 101$ satisfies both inequalities, $B = 100$ does not, so we need 101 buffer pages.
   (b) With 20000 pages in the file, $B = 142$ satisfies both inequalities, $B = 141$ does not, so we need 142 buffer pages.
   (c) With 2000000 pages in the file, $B = 1415$ satisfies both inequalities, $B = 1414$ does not, so we need 1415 buffer pages.

Exercise 13.2 Answer Exercise 13.1 assuming that a two-way external sort is used.

Answer 13.2 The answer to each question is given below.

1. In the first pass (Pass 0), N runs of 1 page each are produced, where N is the number of file pages:
   (a) 10000 sorted runs.
   (b) 20000 sorted runs.
   (c) 2000000 sorted runs.

2. The number of passes required to sort the file completely, including the initial sorting pass, is $\lceil log_2 N1 \rceil + 1$, where $N1 = N$ is the number of runs produced by Pass 0:
   (a) $\lceil log_2 10000 \rceil + 1 = 15$ passes.
   (b) $\lceil log_2 20000 \rceil + 1 = 16$ passes.
   (c) $\lceil log_2 2000000 \rceil + 1 = 22$ passes.

3. Since each page is read and written once per pass, the total number of page I/Os for sorting the file is $2 * N * (\#passes)$:
   (a) 2*10000*15 = 300000.
   (b) 2*20000*16 = 640000.
   (c) 2*2000000*22 = 88000000.

4. Using 2-way merge sort, it is impossible to sort these files in 2 passes. Additional buffer pages do not help, since the algorithm always uses just 3 buffer pages.

Exercise 13.3 Suppose that you just finished inserting several records into a heap file and now want to sort those records. Assume that the DBMS uses external sort and makes efficient use of the available buffer space when it sorts a file. Here is some potentially useful information about the newly loaded file and the DBMS software available to operate on it:

> The number of records in the file is 4500. The sort key for the file is 4 bytes long. You can assume that rids are 8 bytes long and page ids are 4 bytes long. Each record is a total of 48 bytes long. The page size is 512 bytes. Each page has 12 bytes of control information on it. Four buffer pages are available.

1. How many sorted subfiles will there be after the initial pass of the sort, and how long will each subfile be?
2. How many passes (including the initial pass just considered) are required to sort this file?
3. What is the total I/O cost for sorting this file?
4. What is the largest file, in terms of the number of records, you can sort with just four buffer pages in two passes? How would your answer change if you had 257 buffer pages?
5. Suppose that you have a B+ tree index with the search key being the same as the desired sort key. Find the cost of using the index to retrieve the records in sorted order for each of the following cases:
   - The index uses Alternative (1) for data entries.
   - The index uses Alternative (2) and is unclustered. (You can compute the worst-case cost in this case.)
   - How would the costs of using the index change if the file is the largest that you can sort in two passes of external sort with 257 buffer pages? Give your answer for both clustered and unclustered indexes.

Answer 13.3 The answer to each question is given below.

1. Assuming that the general external merge-sort algorithm is used, and that the available space for storing records in each page is $512 - 12 = 500$ bytes, each page can store up to 10 records of 48 bytes each. So 450 pages are needed in order to store all 4500 records, assuming that a record is not allowed to span more than one page.

   Given that 4 buffer pages are available, there will be $\lceil 450/4 \rceil = 113$ sorted runs (sub-files) of 4 pages each, except the last run, which is only 2 pages long.

2. The total number of passes will be equal to $log_3 113 + 1 = 6$ passes.

3. The total I/O cost for sorting this file is $2 * 450 * 6 = 5400$ I/Os.

4. As we saw in the previous exercise, in Pass 0, $\lceil N/B \rceil$ runs are produced. In Pass 1, we must be able to merge this many runs; i.e., $B - 1 \geq \lceil N/B \rceil$. When B is given to be 4, we get $N = 12$. The maximum number of records on 12 pages is $12 * 10 = 120$. When $B = 257$, we get $N = 65792$, and the number of records is $65792 * 10 = 657920$.

5. (a) If the index uses Alternative (1) for data entries, and it is clustered, the cost will be equal to the cost of traversing the tree from the root to the left-most leaf plus the cost of retrieving the pages in the sequence set. Assuming 67% occupancy, the number of leaf pages in the tree (the sequence set) is $450/0.67 = 600$.

   (b) If the index uses Alternative (2), and is not clustered, in the worst case, first we scan B+ tree's leaf pages, also each data entry will require fetching a data page. The number of data entries is equal to the number of data records, which is 4500. Since there is one data entry per record, each data entry requires 12 bytes, and each page holds 512 bytes, the number of B+ tree leaf pages is about $(4500 * 12)/(512 * 0.67))$, assuming 67% occupancy, which is about 150. Thus, about 4650 I/Os are required in a worst-case scenario.

   (c) The B+ tree in this case has $65792/0.67 = 98197$ leaf pages if Alternative (1) is used, assuming 67% occupancy. This is the number of I/Os required (plus the relatively minor cost of going from the root to the left-most leaf). If Alternative (2) is used, and the index is not clustered, the number of I/Os is approximately equal to the number of data entries in the worst case, that is 657920,plus the number of B+ tree leaf pages 2224. Thus, number of I/Os is 660144.

Exercise 13.4 Consider a disk with an average seek time of 10ms, average rotational delay of 5ms, and a transfer time of 1ms for a 4K page. Assume that the cost of reading/writing a page is the sum of these values (i.e., 16ms) unless a *sequence* of pages is read/written. In this case, the cost is the average seek time plus the average rotational delay (to find the first page in the sequence) plus 1ms per page (to transfer data). You are given 320 buffer pages and asked to sort a file with 10,000,000 pages.

1. Why is it a bad idea to use the 320 pages to support virtual memory, that is, to 'new' $10{,}000{,}000 \cdot 4$K bytes of memory, and to use an in-memory sorting algorithm such as Quicksort?

2. Assume that you begin by creating sorted runs of 320 pages each in the first pass. Evaluate the cost of the following approaches for the subsequent merging passes:

   (a) Do 319-way merges.

(b) Create 256 'input' buffers of 1 page each, create an 'output' buffer of 64 pages, and do 256-way merges.

(c) Create 16 'input' buffers of 16 pages each, create an 'output' buffer of 64 pages, and do 16-way merges.

(d) Create eight 'input' buffers of 32 pages each, create an 'output' buffer of 64 pages, and do eight-way merges.

(e) Create four 'input' buffers of 64 pages each, create an 'output' buffer of 64 pages, and do four-way merges.

Answer 13.4 In Pass 0, 31250 sorted runs of 320 pages each are created. For each run, we read and write 320 pages sequentially. The I/O cost per run is $2 * (10 + 5 + 1 * 320) = 670$ms. Thus, the I/O cost for Pass 0 is $31250 * 670 = 20937500$ms. For each of the cases discussed below, this cost must be added to the cost of the subsequent merging passes to get the total cost. Also, the calculations below are slightly simplified by neglecting the effect of a final read/written block that is slightly smaller than the earlier blocks.

1. For 319-way merges, only 2 more passes are needed. The first pass will produce

$$\lceil 31250/319 \rceil = 98$$

sorted runs; these can then be merged in the next pass. Every page is read and written individually, at a cost of 16ms per read or write, in each of these two passes. The cost of these merging passes is therefore $2*(2*16)*10000000 = 640000000$ms. (The formula can be read as 'number of passes times cost of read and write per page times number of pages in file'.)

2. With 256-way merges, only two additional merging passes are needed. Every page in the file is read and written in each pass, but the effect of blocking is different on reads and writes. For reading, each page is read individually at a cost of 16ms. Thus, the cost of reads (over both passes) is $2 * 16 * 10000000 = 320000000$ms. For writing, pages are written out in blocks of 64 pages. The I/O cost per block is $10 + 5 + 1 * 64 = 79$ms. The number of blocks written out per pass is $10000000/64 = 156250$, and the cost per pass is $156250 * 79 = 12343750$ms. The cost of writes over both merging passes is therefore $2 * 12343750 = 24687500$ms. The total cost of reads and writes for the two merging passes is $320000000 + 24687500 = 344687500$ms.

3. With 16-way merges, 4 additional merging passes are needed. For reading, pages are read in blocks of 16 pages, at a cost per block of $10 + 5 + 1 * 16 = 31$ms. In each pass, $10000000/16 = 625000$ blocks are read. The cost of reading over the 4 merging passes is therefore $4 * 625000 * 31 = 77500000$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 4 merging passes is $4 * 12343750 = 49375000$ms, and the total cost of the merging passes is $77500000 + 49375000 = 126875000$ms.

4. With 8-way merges, 5 merging passes are needed. For reading, pages are read in blocks of 32 pages, at a cost per block of $10 + 5 + 1 * 32 = 47$ms. In each pass, $10000000/32 = 312500$ blocks are read. The cost of reading over the 5 merging passes is therefore $5 * 312500 * 47 = 73437500$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 5 merging passes is $5 * 12343750 = 61718750$ms, and the total cost of the merging passes is $73437500 + 61718750 = 135156250$ms.

5. With 4-way merges, 8 merging passes are needed. For reading, pages are read in blocks of 64 pages, at a cost per block of $10 + 5 + 1 * 64 = 79$ms. In each pass, $10000000/64 = 156250$ blocks are read. The cost of reading over the 8 merging passes is therefore $8 * 156250 * 79 = 98750000$ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 8 merging passes is $8 * 12343750 = 98750000$ms, and the total cost of the merging passes is $98750000 + 98750000 = 197500000$ms.

There are several lessons to be drawn from this (rather tedious) exercise. First, the cost of the merging phase varies from a low of 126875000ms to a high of 640000000ms. Second, the highest cost is associated with the option of maximizing fanout, choosing a buffer size of 1 page! Thus, the effect of blocked I/O is significant. However, as the block size is increased, the number of passes increases slowly, and there is a trade-off to be considered: it does not pay to increase block size indefinitely. Finally, while this example uses a different block size for reads and writes, for the sake of illustration, in practice a single block size is used for both reads and writes.

Exercise 13.5 Consider the refinement to the external sort algorithm that produces runs of length $2B$ on average, where B is the number of buffer pages. This refinement was described in Section 11.2.1 under the assumption that all records are the same size. Explain why this assumption is required and extend the idea to cover the case of variable-length records.

Answer 13.5 The assumption that all records are of the same size is used when the algorithm moves the smallest entry with a key value large than k to the output buffer and replaces it with a value from the input buffer. This "replacement" will only work if the records of the same size.

If the entries are of variable size, then we must also keep track of the size of each entry, and replace the moved entry with a new entry that fits in the available memory location. Dynamic programming algorithms have been adapted to decide an optimal replacement strategy in these cases.

14

EVALUATION OF RELATIONAL OPERATORS

Exercise 14.1 Briefly answer the following questions:

1. Consider the three basic techniques, *iteration*, *indexing*, and *partitioning*, and the relational algebra operators *selection*, *projection*, and *join*. For each technique-operator pair, describe an algorithm based on the technique for evaluating the operator.
2. Define the term *most selective access path for a query*.
3. Describe *conjunctive normal form*, and explain why it is important in the context of relational query evaluation.
4. When does a general selection condition *match* an index? What is a *primary term* in a selection condition with respect to a given index?
5. How does hybrid hash join improve on the basic hash join algorithm?
6. Discuss the pros and cons of hash join, sort-merge join, and block nested loops join.
7. If the join condition is not equality, can you use sort-merge join? Can you use hash join? Can you use index nested loops join? Can you use block nested loops join?
8. Describe how to evaluate a grouping query with aggregation operator `MAX` using a sorting-based approach.
9. Suppose that you are building a DBMS and want to add a new aggregate operator called `SECOND LARGEST`, which is a variation of the `MAX` operator. Describe how you would implement it.
10. Give an example of how buffer replacement policies can affect the performance of a join algorithm.

Answer 14.1 The answer to each question is given below.

1. (a) *iteration–selection* Scan the entire collection, checking the condition on each tuple, and adding the tuple to the result if the condition is satisfied.
   (b) *indexing–selection* If the selection is equality and a B+ or hash index exists on the field condition, we can retrieve relevant tuples by finding them in the index and then locating them on disk.
   (c) *partitioning–selection* Do a binary search on sorted data to find the first tuple that mathes the condition. To retireve the remaining entries, we simple scan the collection starting at the first tuple we found.
   (d) *iteration–projection* Scan the entire relation, and eliminate unwanted attributes in the result.
   (e) *indexing–projection* If a multiattribute B+ tree index exists for all of the projection attributes, then one needs to only look at the leaves of the B+.
   (f) *partitioning–projection* To elimiate duplicates when doing a projection, one can simply project out the unwanted attributes and hash a combination of the remaining attributes so duplicates can be easily detected.
   (g) *iteration–join* To join two relations, one takes the first attribute in the first relation and scans the entire second relation to find tuples that match the join condition. Once the first attribute has compared to all tuples in the second relation, the second attribute from the first relation is compared to all tuples in the second relation, and so on.
   (h) *indexing–join* When an index is available, joining two relations can be more efficient. Say there are two relations A and B, and there is a secondary index on the join condition over relation A. The join works as follows: for each tuple in B, we lookup the join attribute in the index over relation A to see if there is a match. If there is a match, we store the tuple, otherwise we move to the next tuple in relation B.
   (i) *partitioning–join* One can join using partitioning by using hash join variant or a sort-merge join. For example, if there is a sort merge join, we sort both relations on the the join condition. Next, we scan both relations and identify matches. After sorting, this requires only a single scan over each relation.

2. The *most selective access path* is the query access path that retrieves the fewest pages during query evaluation. This is the most efficient way to gather the query's results.

3. *Conjuntive normal form* is important in query evaluation because often indexes exist over some subset of conjucts in a CNF expression. Since conjuct order does not matter in CNF expressions, often indexes can be used to increase the selectivity of operators by doing a selection over two, three, or more conjucts using a single multiattribute index.

4. An index *matches* a selection condition if the index can be used to retrieve just the tuples that satisfy the condition. A *primary term* in a selection condition is a conjuct that matches an index (i.e. can be used by the index).

5. Hybrid hash join improves performance by comparing the first hash buckets during the partitioning phase rather than saving it for the probing phase. This saves us teh cost of writing and reading the first partition to disk.

6. Hash join provides excellent performance for equality joins, and can be tuned to require very few extra disk accesses beyond a one-time scan (provided enough memory is available). However, hash join is worthless for non-equality joins.

   Sort-merge joins are suitable when there is either an equality or non-equality based join condition. Sort-merge also leaves the results sorted which is often a desired property. Sort-merge join has extra costs when you have to use external sorting (there is not enough memory to do the sort in-memory).

   Block nested loops is efficient when one of the relations will fit in memory and you are using an MRU replacement strategy. However, if an index is available, there are better strategies available (but often indexes are not available).

7. If the join condition is not equality, you can use sort-merge join, index nested loops (if you have a range style index such as a B+ tree index or ISAM index), or block nested loops join. Hash joining works best for equality joins and is not suitable otherwise.

8. First we sort all of the tuples based on the `GROUP BY` attribute. Next we re-sort each group by sorting all elements on the `MAX` attribute, taking case not to re-sort beyond the group boundaries.

9. The operator `SECOND LARGEST` can be implemented using sorting. For each group (if there is a `GROUP BY` clause), we sort the tuples and return the second largest value for the desired attribute. The cost here is the cost of sorting.

10. One example where the buffer replacement stratagy effects join performance is the use of LRU and MRU in an simple nested loops join. If the relations don't fit in main memory, then the buffer strategy is critical. Say there are M buffer pages and N are filled by the first relation, and the second relation is of size M-N+P, meaning all of the second relation will fit in the the buffer except P pages. Since we must do repeated scans of the second relation, the replacement policy comes into play. With LRU, whenever we need to find a page it will have been paged out so every page request requires a disk IO. On the other hand, with MRU, we will only need to reread P-1 of the pages in the second relation, since the others will remain in memory.

Exercise 14.2 Consider a relation R(*a,b,c,d,e*) containing 5,000,000 records, where each data page of the relation holds 10 records. R is organized as a sorted file with secondary indexes. Assume that *R.a* is a candidate key for R, with values lying in the range 0 to 4,999,999, and that R is stored in *R.a* order. For each of the following relational algebra queries, state which of the following approaches (or combination thereof) is most likely to be the cheapest:

- Access the sorted file for R directly.
- Use a clustered B+ tree index on attribute $R.a$.
- Use a linear hashed index on attribute $R.a$.
- Use a clustered B+ tree index on attributes $(R.a, R.b)$.
- Use a linear hashed index on attributes $(R.a, R.b)$.
- Use an unclustered B+ tree index on attribute $R.b$.

1. $\sigma_{a<50,000 \wedge b<50,000}(R)$
2. $\sigma_{a=50,000 \wedge b<50,000}(R)$
3. $\sigma_{a>50,000 \wedge b=50,000}(R)$
4. $\sigma_{a=50,000 \wedge a=50,010}(R)$
5. $\sigma_{a\neq 50,000 \wedge b=50,000}(R)$
6. $\sigma_{a<50,000 \vee b=50,000}(R)$

Answer 14.2 Answer Omitted.

Exercise 14.3 Consider processing the following SQL projection query:

```
SELECT DISTINCT E.title, E.ename FROM Executives E
```

You are given the following information:

Executives has attributes *ename*, *title*, *dname*, and *address*; all are string fields of the same length.
The *ename* attribute is a candidate key.
The relation contains 10,000 pages.
There are 10 buffer pages.

Consider the optimized version of the sorting-based projection algorithm: The initial sorting pass reads the input relation and creates sorted runs of tuples containing only attributes *ename* and *title*. Subsequent merging passes eliminate duplicates while merging the initial runs to obtain a single sorted result (as opposed to doing a separate pass to eliminate duplicates from a sorted result containing duplicates).

1. How many sorted runs are produced in the first pass? What is the average length of these runs? (Assume that memory is utilized well and any available optimization to increase run size is used.) What is the I/O cost of this sorting pass?

2. How many additional merge passes are required to compute the final result of the projection query? What is the I/O cost of these additional passes?

3. (a) Suppose that a clustered B+ tree index on *title* is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (b) Suppose that a clustered B+ tree index on *ename* is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (c) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

4. Suppose that the query is as follows:

   ```
   SELECT E.title, E.ename FROM Executives E
   ```

   That is, you are not required to do duplicate elimination. How would your answers to the previous questions change?

Answer 14.3 The answer to each question is given below.

1. The first pass will produce 250 sorted runs of 20 pages each, costing 15000 I/Os.

2. Using the ten buffer pages provided, on average we can write 2*10 internally sorted pages per pass, instead of 10. Then, three more passes are required to merge the 5000/20 runs, costing 2*3*5000 = 30000 I/Os.

3. (a) Using a clustered B+ tree index on *title* would reduce the cost to single scan, or 12,500 I/Os. An unclustered index could potentially cost more than 2500+100,000 (2500 from scanning the B+ tree, and 10000 * tuples per page, which I just assumed to be 10). Thus, an unclustered index would not be cheaper. Whether or not to use a hash index would depend on whether the index is clustered. If so, the hash index would probably be cheaper.

   (b) Using the clustered B+ tree on *ename* would be cheaper than sorting, in that the cost of using the B+ tree would be 12,500 I/Os. Since *ename* is a candidate key, no duplicate checking need be done for $< title, ename >$ pairs. An unclustered index would require 2500 (scan of index) + 10000 * tuples per page I/Os and thus probably be more expensive than sorting.

   (c) Using a clustered B+ tree index on $< ename, title >$ would also be more cost-effective than sorting. An unclustered B+ tree over the same attributes

would allow an index-only scan, and would thus be just as economical as the clustered index. This method (both by clustered and unclustered) would cost around 5000 I/O's.

4. Knowing that duplicate elimination is not required, we can simply scan the relation and discard unwanted fields for each tuple. This is the best strategy except in the case that an index (clustered or unclustered) on $< ename, title >$ is available; in this case, we can do an index-only scan. (Note that even with `DISTINCT` specified, no duplicates are actually present int he answer because *ename* is a candidate key. However, a typical optimizer is not likely to recognize this and omit the duplicate elimination step.)

Exercise 14.4 Consider the join $R \bowtie_{R.a=S.b} S$, given the following information about the relations to be joined. The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

Relation R contains 10,000 tuples and has 10 tuples per page.
Relation S contains 2000 tuples and also has 10 tuples per page.
Attribute b of relation S is the primary key for S.
Both relations are stored as simple heap files.
Neither relation has any indexes built on it.
52 buffer pages are available.

1. What is the cost of joining R and S using a page-oriented simple nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
2. What is the cost of joining R and S using a block nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
3. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?
4. What is the cost of joining R and S using a hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?
5. What would be the lowest possible I/O cost for joining R and S using *any* join algorithm, and how much buffer space would be needed to achieve this cost? Explain briefly.
6. How many tuples does the join of R and S produce, at most, and how many pages are required to store the result of the join back on disk?
7. Would your answers to any of the previous questions in this exercise change if you were told that $R.a$ is a foreign key that refers to $S.b$?

Answer 14.4 Let $M = 1000$ be the number of pages in R, $N = 200$ be the number of pages in S, and $B = 52$ be the number of buffer pages available.

1. Basic idea is to read each page of the outer relation, and for each page scan the inner relation for matching tuples. Total cost would be

   $$\#pagesinouter + (\#pagesinouter * \#pagesininner)$$

   which is minimized by having the smaller relation be the outer relation.

   $$TotalCost = N + (N * M) = 200,200$$

   The minimum number of buffer pages for this cost is 3.

2. This time read the outer relation in *blocks*, and for each block scan the inner relation for matching tuples. So the outer relation is still read once, but the inner relation is scanned only once for each outer block, of which there are $\lceil \frac{\#pagesinouter}{B-2} \rceil = \lceil 200/50 \rceil = 4$.

   $$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 4,200$$

   If the number of buffer pages is less than 52, the number of scans of the inner would be more than 4 since $= \lceil 200/49 \rceil$ is 5. The minimum number of buffer pages for this cost is therefore 52.

3. Since $B > \sqrt{M} > \sqrt{N}$ we can use the refinement to Sort-Merge discussed on pages 254-255 in the text.

   $$TotalCost = 3 * (M + N) = 3,600$$

   NOTE: if $S.b$ were not a key, then the merging phase could require more than one pass over one of the relations, making the cost of merging $M * N$ I/Os in the worst case.

   The minimum number of buffer pages required is 25. With 25 buffer pages, the initial sorting pass will split R into 20 runs of size 50 and split S into 4 runs of size 50 (approximately). These 24 runs can then be merged in one pass, with one page left over to be used as an output buffer. With fewer than 25 buffer pages the number of runs produced by the first pass over both relations would exceed the number of available pages, making a one-pass merge impossible.

4. The cost of Hash Join is $3*(M+N)$ if $B > \sqrt{f * N}$ where f is a 'fudge factor' used to capture the small increase in size involved in building a hash table, and N is the number of pages in the smaller relation, S (see page 258). Since $\sqrt{N} \approx 14$, we can assume that this condition is met. We will also assume uniform partitioning from our hash function.

   $$TotalCost = 3 * (M + N) = 3,600$$

Without knowing f we can only approximate the minimum number of buffer pages required, and a good guess is that we need $B > \sqrt{f * N}$.

5. The optimal cost would be achieved if each relation was only read once. We could do such a join by storing the entire smaller relation in memory, reading in the larger relation page-by-page, and for each tuple in the larger relation we search the smaller relation (which exists entirely in memory) for matching tuples. The buffer pool would have to hold the entire smaller relation, one page for reading in the larger relation, and one page to serve as an output buffer.

$$TotalCost = M + N = 1,200$$

The minimum number of buffer pages for this cost is $N + 1 + 1 = 202$.

6. Any tuple in R can match at most one tuple in S because $S.b$ is a primary key (which means the $S.b$ field contains no duplicates). So the maximum number of tuples in the result is equal to the number of tuples in R, which is 10,000.

The size of a tuple in the result could be as large as the size of an R tuple plus the size of an S tuple (minus the size of the shared attribute). This may allow only 5 tuples to be stored on a page. Storing 10,000 tuples at 5 per page would require 2000 pages in the result.

7. The foreign key constraint tells us that for every R tuple there is *exactly* one matching S tuple (because $S.b$ is a key). The Sort-Merge and Hash Joins would not be affected, but we could reduce the cost of the two Nested Loops joins. If we make R the outer relation then for each tuple of R we only have to scan S until a match is found. This will require scanning only 50% of S on average.

For Page-Oriented Nested Loops, the new cost would be

$$TotalCost = M + (M * \frac{N}{2}) = 101,000$$

and 3 buffer pages are still required.

For Block Nested Loops, the new cost would be

$$TotalCost = M + (\frac{N}{2}) * \lceil \frac{M}{B-2} \rceil = 3,000$$

and again this cost can only be achieved with 52 available buffer pages.

Exercise 14.5 Consider the join of R and S described in Exercise 14.1.

1. With 52 buffer pages, if unclustered B+ indexes existed on $R.a$ and $S.b$, would either provide a cheaper alternative for performing the join (using an index nested loops join) than a block nested loops join? Explain.

(a) Would your answer change if only five buffer pages were available?

(b) Would your answer change if S contained only 10 tuples instead of 2000 tuples?

2. With 52 buffer pages, if *clustered* B+ indexes existed on $R.a$ and $S.b$, would either provide a cheaper alternative for performing the join (using the *index nested loops* algorithm) than a block nested loops join? Explain.

   (a) Would your answer change if only five buffer pages were available?

   (b) Would your answer change if S contained only 10 tuples instead of 2000 tuples?

3. If only 15 buffers were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?

4. If the size of S were increased to also be 10,000 tuples, but only 15 buffer pages were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?

5. If the size of S were increased to also be 10,000 tuples, and 52 buffer pages were available, what would be the cost of sort-merge join? What would be the cost of hash join?

Answer 14.5 Assume that it takes 3 I/Os to access a leaf in R, and 2 I/Os to access a leaf in S. And since S.b is a primary key, we will assume that every tuple in S matches 5 tuples in R.

1. The Index Nested Loops join involves probing an index on the inner relation for each tuple in the outer relation. The cost of the probe is the cost of accessing a leaf page plus the cost of retrieving any matching data records. The cost of retrieving data records could be as high as one I/O per record for an unclustered index.

   With R as the outer relation, the cost of the Index Nested Loops join will be the cost of reading R plus the cost of 10,000 probes on S.

   $$TotalCost = 1,000 + 10,000 * (2 + 1) = 31,000$$

   With S as the outer relation, the cost of the Index Nested Loops join will be the cost of reading S plus the cost of 2000 probes on R.

   $$TotalCost = 200 + 2,000 * (3 + 5) = 16,200$$

   Neither of these solutions is cheaper than Block Nested Loops join which required 4,200 I/Os.

(a) With 5 buffer pages, the cost of the Index Nested Loops joins remains the same, but the cost of the Block Nested Loops join will increase. The new cost of the Block Nested Loops join is

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 67,200$$

And now the cheapest solution is the Index Nested Loops join with S as the outer relation.

(b) If S contains 10 tuples then we'll need to change some of our initial assumptions. Now all of the S tuples fit on a single page, and it will only require a single I/O to access the (single) leaf in the index. Also, each tuple in S will match 1,000 tuples in R.

Block Nested Loops:

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 1,001$$

Index Nested Loops with R as the outer relation:

$$TotalCost = 1,000 + 10,000 * (1 + 1) = 21,000$$

Index Nested Loops with S as the outer relation:

$$TotalCost = 1 + 10 * (3 + 1,000) = 10,031$$

Block Nested Loops is still the best solution.

2. With a clustered index the cost of accessing data records becomes one I/O for every 10 data records.

With R as the outer relation, the cost of the Index Nested Loops join will be the cost of reading R plus the cost of 10,000 probes on S.

$$TotalCost = 1,000 + 10,000 * (2 + 1) = 31,000$$

With S as the outer relation, the cost of the Index Nested Loops join will be the cost of reading S plus the cost of 2000 probes on R.

$$TotalCost = 200 + 2,000 * (3 + 1) = 8,200$$

Neither of these solutions is cheaper than Block Nested Loops join which required 4,200 I/Os.

(a) With 5 buffer pages, the cost of the Index Nested Loops joins remains the same, but the cost of the Block Nested Loops join will increase. The new cost of the Block Nested Loops join is

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 67,200$$

And now the cheapest solution is the Index Nested Loops join with S as the outer relation.

(b) If S contains 10 tuples then we'll need to change some of our initial assumptions. Now all of the S tuples fit on a single page, and it will only require a single I/O to access the (single) leaf in the index. Also, each tuple in S will match 1,000 tuples in R.
Block Nested Loops:

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 1,001$$

Index Nested Loops with R as the outer relation:

$$TotalCost = 1,000 + 10,000 * (1 + 1) = 21,000$$

Index Nested Loops with S as the outer relation:

$$TotalCost = 1 + 10 * (3 + 100) = 1,031$$

Block Nested Loops is still the best solution.

3. SORT-MERGE: With 15 buffer pages we can sort R in three passes and S in two passes. The cost of sorting R is $2 * 3 * M = 6,000$, the cost of sorting S is $2 * 2 * N = 800$, and the cost of the merging phase is $M + N = 1,200$.

$$TotalCost = 6,000 + 800 + 1,200 = 8,000$$

HASH JOIN: With 15 buffer pages the first scan of S (the smaller relation) splits it into 14 buckets, each containing about 15 pages. To store one of these buckets (and its hash table) in memory will require $f * 15$ pages, which is more than we have available. We must apply the Hash Join technique again to all partitions of R and S that were created by the first partitioning phase. Then we can fit an entire partition of S in memory. The total cost will be the cost of two partioning phases plus the cost of one matching phase.

$$TotalCost = 2 * (2 * (M + N)) + (M + N) = 6,000$$

4. SORT-MERGE: With 15 buffer pages we can sort R in three passes and S in three passes. The cost of sorting R is $2 * 3 * M = 6,000$, the cost of sorting S is $2 * 3 * N = 6,000$, and the cost of the merging phase is $M + N = 2,000$.

$$TotalCost = 6,000 + 6,000 + 2,000 = 14,000$$

HASH JOIN: Now both relations are the same size, so we can treat either one as the smaller relation. With 15 buffer pages the first scan of S splits it into 14 buckets, each containing about 72 pages, so again we have to deal with partition overflow. We must apply the Hash Join technique again to all partitions of R and S that were created by the first partitioning phase. Then we can fit an entire partition of S in memory. The total cost will be the cost of two partioning phases plus the cost of one matching phase.

$$TotalCost = 2 * (2 * (M + N)) + (M + N) = 10,000$$

5. SORT-MERGE: With 52 buffer pages we have $B > \sqrt{M}$ so we can use the "merge-on-the-fly" refinement which costs $3 * (M + N)$.

$$TotalCost = 3 * (1,000 + 1,000) = 6,000$$

HASH JOIN: Now both relations are the same size, so we can treat either one as the smaller relation. With 52 buffer pages the first scan of S splits it into 51 buckets, each containing about 20 pages. This time we do not have to deal with partition overflow. The total cost will be the cost of one partioning phase plus the cost of one matching phase.

$$TotalCost = (2 * (M + N)) + (M + N) = 6,000$$

Exercise 14.6 Answer each of the questions—if some question is inapplicable, explain why—in Exercise 14.4 again but using the following information about R and S:

Relation R contains 200,000 tuples and has 20 tuples per page.
Relation S contains 4,000,000 tuples and also has 20 tuples per page.
Attribute a of relation R is the primary key for R.
Each tuple of R joins with exactly 20 tuples of S.
1,002 buffer pages are available.

Answer 14.6 Let M = 10,000 be the number of pages in R, N = 200,000 be the number of pages in S, and B = 1002 be the number of buffer pages available.

1. Basic idea is to read each page of the outer relation, and for each page scan the inner relation for matching tuples. Total cost would be

$$\#pagesinouter + (\#pagesinouter * \#pagesininner)$$

which is minimized by having the smaller relation be the outer relation.

$$TotalCost = M + (M * N) = 2,000,010,000$$

The minimum number of buffer pages for this cost is 3.

2. This time read the outer relation in *blocks*, and for each block scan the inner relation for matching tuples. So the outer relation is still read once, but the inner relation is scanned only once for each outer block, of which there are $\lceil \frac{\#pagesinouter}{B-2} \rceil$.

$$TotalCost = M + N * \lceil \frac{M}{B-2} \rceil = 2,010,000$$

The minimum number of buffer pages for this cost is 1002.

3. Since $B > \sqrt{N} > \sqrt{M}$ we can use the refinement to Sort-Merge discussed on pages 254-255 in the text.

$$TotalCost = 3 * (M + N) = 630,000$$

NOTE: if $R.a$ were not a key, then the merging phase could require more than one pass over one of the relations, making the cost of merging $M * N$ I/Os in the worst case.

The minimum number of buffer pages required is 325. With 325 buffer pages, the initial sorting pass will split R into 16 runs of size 650 and split S into 308 runs of size 650 (approximately). These 324 runs can then be merged in one pass, with one page left over to be used as an output buffer. With fewer than 325 buffer pages the number of runs produced by the first pass over both relations would exceed the number of available pages, making a one-pass merge impossible.

4. The cost of Hash Join is $3*(M+N)$ if $B > \sqrt{f * M}$ where f is a 'fudge factor' used to capture the small increase in size involved in building a hash table, and M is the number of pages in the smaller relation, S (see page 258). Since $\sqrt{M} = 100$, we can assume that this condition is met. We will also assume uniform partitioning from our hash function.

$$TotalCost = 3 * (M + N) = 630,000$$

Without knowing f we can only approximate the minimum number of buffer pages required, and a good guess is that we need $B > \sqrt{f * M}$.

5. The optimal cost would be achieved if each relation was only read once. We could do such a join by storing the entire smaller relation in memory, reading in the larger relation page-by-page, and for each tuple in the larger relation we search the smaller relation (which exists entirely in memory) for matching tuples. The buffer pool would have to hold the entire smaller relation, one page for reading in the larger relation, and one page to serve as an output buffer.

$$TotalCost = M + N = 210,000$$

The minimum number of buffer pages for this cost is $M + 1 + 1 = 10,002$.

6. Any tuple in S can match at most one tuple in R because $R.a$ is a primary key (which means the $R.a$ field contains no duplicates). So the maximum number of tuples in the result is equal to the number of tuples in S, which is 4,000,000.

The size of a tuple in the result could be as large as the size of an R tuple plus the size of an S tuple (minus the size of the shared attribute). This may allow only 10 tuples to be stored on a page. Storing 4,000,000 tuples at 10 per page would require 400,000 pages in the result.

7. If R.b is a foreign key referring to S.a, this contradicts the statement that each R tuple joins with exactly 20 S tuples.

Exercise 14.7 We described variations of the join operation called *outer joins* in Section 5.6.4. One approach to implementing an outer join operation is to first evaluate the corresponding (inner) join and then add additional tuples padded with *null* values to the result in accordance with the semantics of the given outer join operator. However, this requires us to compare the result of the inner join with the input relations to determine the additional tuples to be added. The cost of this comparison can be avoided by modifying the join algorithm to add these extra tuples to the result while input tuples are processed during the join. Consider the following join algorithms: *block nested loops join, index nested loops join, sort-merge join,* and *hash join.* Describe how you would modify each of these algorithms to compute the following operations on the Sailors and Reserves tables discussed in this chapter:

1. Sailors `NATURAL LEFT OUTER JOIN` Reserves
2. Sailors `NATURAL RIGHT OUTER JOIN` Reserves
3. Sailors `NATURAL FULL OUTER JOIN` Reserves

Answer 14.7 Each join method is considered in turn.

1. Sailors (S) `NATURAL LEFT OUTER JOIN` Reserves (R)
   In this `LEFT OUTER JOIN`, Sailor rows without a matching Reserves row will appear in the result with a *null* value for the Reserves value.

   (a) *block nested loops join*
   In the *block nested loops join* algorithm, we place as large a partition of the Sailors relation in memory as possibly, leaving 2 extra buffer pages (one for input pages of R, the other for output pages plus enough pages for a single bit for each record of the block of S. These 'bit pages' are initially set to zero; when a tuple of R matches a tuple in S, the bit is set to 1 meaning that this page has already met the join condition. Once all of R has been compared to the block of S, any tuple with its bit still set to zero is added to the ouput with a *null* value for the R tuple. This process is then repeated for the remaining blocks of S.

   (b) *index nested loops join*
   An *index nested loops join* requires an index for Reserves on all attributes that Sailors and Reserves have in common. For each tuple in Sailors, if it matches a tuple in the R index, it is added to the output, otherwise the S tuple is added to the output with a *null* value.

   (c) *sort-merge join*
   When the two relations are merged, Sailors is scanned in sorted order and if there is no match in Reserves, the Sailors tuple is added to the output with a *null* value.

(d) *hash join*
We hash so that partitions of Reserves will fit in memory with enough leftover space to hold a page of the corresponding Sailors partition. When we compare a Sailors tuple to all of the tuples in the Reserves partition, if there is a match it is added to the output, otherwise we add the S tuple and a *null* value to the output.

2. Sailors `NATURAL RIGHT OUTER JOIN` Reserves
In this `RIGHT OUTER JOIN`, Reserves rows without a matching Sailors row will appear in the result with a *null* value for the Sailors value.

(a) *block nested loops join*
In the *block nested loops join* algorithm, we place as large a partition of the Reserves relation in memory as possibly, leaving 2 extra buffer pages (one for input pages of Sailors, the other for output pages plus enough pages for a single bit for each record of the block of R. These 'bit pages' are initially set to zero; when a tuple of S matches a tuple in R, the bit is set to 1 meaning that this page has already met the join condition. Once all of S has been compared to the block of R, any tuple with its bit still set to zero is added to the ouput with a *null* value for the S tuple. This process is then repeated for the remaining blocks of R.

(b) *index nested loops join*
An *index nested loops join* requires an index for Sailors on all attributes that Reserves and Sailors have in common. For each tuple in Reserves, if it matches a tuple in the S index, it is added to the output, otherwise the R tuple is added to the output with a *null* value.

(c) *sort-merge join*
When the two relations are merged, Reserves is scanned in sorted order and if there is no match in Sailors, the Reserves tuple is added to the output with a *null* value.

(d) *hash join*
We hash so that partitions of Sailors will fit in memory with enough leftover space to hold a page of the corresponding Reserves partition. When we compare a Reserves tuple to all of the tuples in the Sailors partition, if there is a match it is added to the output, otherwise we add the Reserves tuple and a *null* value to the output.

3. Sailors `NATURAL FULL OUTER JOIN` Reserves
In this `FULL OUTER JOIN`, Sailor rows without a matching Reserves row will appear in the result with a *null* value for the Reserves value, and Reserves rows without a matching Sailors row will appear in teh result with a *null* value.

(a) *block nested loops join*
For this algorithm to work properly, we need a bit for each tuple in both

relations. If after completing the join there are any bits still set to zero, these tuples are joined with *null* values.

(b) *index nested loops join*
If there is only an index on one relation, we can use that index to find half of the full outer join in a similar fashion as in the `LEFT` and `RIGHT OUTER` joins. To find the non-matches of the relation with the index, we can use the same trick as in the *block nested loops join* and keep bit flags for each block of scans.

(c) *sort-merge join*
During the merge phase, we scan both relations alternating to the relation with the lower value. If that tuple has no match, it is added to the output with a *null* value.

(d) *hash join*
When we hash both relations, we should choose a hash function that will hash the larger relation into partitions that will fit in half of memory. This way we can fit both relations' partitions into main memory and we can scan both relations for matches. If no match is found (we must scan for both relations), then we add that tuple to the output with a *null* value.

15

A TYPICAL QUERY OPTIMIZER

Exercise 15.1 Briefly answer the following questions:

1. In the context of query optimization, what is an *SQL query block*?
2. Define the term *reduction factor*.
3. Describe a situation in which projection should precede selection in processing a project-select query, and describe a situation where the opposite processing order is better. (Assume that duplicate elimination for projection is done via sorting.)
4. If there are unclustered (secondary) B+ tree indexes on both $R.a$ and $S.b$, the join $R \bowtie_{a=b} S$ could be processed by doing a sort-merge type of join—without doing any sorting—by using these indexes.
   (a) Would this be a good idea if R and S each has only one tuple per page or would it be better to ignore the indexes and sort R and S? Explain.
   (b) What if R and S each have many tuples per page? Again, explain.
5. Explain the role of *interesting orders* in the System R optimizer.

Answer 15.1 The answer to each question is given below.

1. An SQL query block is an SQL query without nesting, and serves as a unit of optimization. Blocks have one `SELECT` statement, one `FROM` statement, and at most one `WHERE`, one `GROUP BY`, and one `HAVING` statements. Queries with nesting can be broken up into a collection of query blocks whose evaluation must be coordinated at runtime.
2. The *reduction factor* for a term, is the ratio between the expected result size to the input size, considering only the slection represented by the term.

3. If the selection is to be done on the inner relation of a simple nested loop, and the projection will reduce the number of pages occupied significantly, then the projection should be done first.

   The opposite is true in the case of an index-only join. The projections should be done on the fly after the join.

4. (a) Using the indexes is a good idea when R and S each have only one tuple per page. Each data page is read exactly once and the cost of scanning the B+ tree is likely to be very small.

   (b) Doing an actual data sort on appropriate keys may be a good idea when R and S have many tuples per page. Given that the indexes are unclustered, without sorting there is potential for many reads of a single page. After sorting, there will only be one read per matching page. The choice may be determined by number of potential matches and number of tuples per page.

5. The System R optimizer implements a multiple pass algorithm. In each pass, it must consider adding a join to those retained in previous passes. Each level retains the cheapest plan for each interesting order for result tuples. An ordering of tuples is interesting if it is sorted on some combination of fields.

Exercise 15.2 Consider a relation with this schema:

Employees(*eid:* `integer`, *ename:* `string`, *sal:* `integer`, *title:* `string`, *age:* `integer`)

Suppose that the following indexes, all using Alternative (2) for data entries, exist: a hash index on *eid*, a B+ tree index on *sal*, a hash index on *age*, and a clustered B+ tree index on ⟨*age, sal*⟩. Each Employees record is 100 bytes long, and you can assume that each index data entry is 20 bytes long. The Employees relation contains 10,000 pages.

1. Consider each of the following selection conditions and, assuming that the reduction factor (RF) for each term that matches an index is 0.1, compute the cost of the most selective access path for retrieving all Employees tuples that satisfy the condition:

   (a) $sal > 100$

   (b) $age = 25$

   (c) $age > 20$

   (d) $eid = 1,000$

   (e) $sal > 200 \wedge age > 30$

   (f) $sal > 200 \wedge age = 20$

   (g) $sal > 200 \wedge title = 'CFO'$

(h) $sal > 200 \wedge age > 30 \wedge title = 'CFO'$

2. Suppose that, for each of the preceding selection conditions, you want to retrieve the average salary of qualifying tuples. For each selection condition, describe the least expensive evaluation method and state its cost.

3. Suppose that, for each of the preceding selection conditions, you want to compute the average salary for each *age* group. For each selection condition, describe the least expensive evaluation method and state its cost.

4. Suppose that, for each of the preceding selection conditions, you want to compute the average age for each *sal* level (i.e., group by *sal*). For each selection condition, describe the least expensive evaluation method and state its cost.

5. For each of the following selection conditions, describe the best evaluation method:

   (a) $sal > 200 \vee age = 20$

   (b) $sal > 200 \vee title = 'CFO'$

   (c) $title = 'CFO' \wedge ename = 'Joe'$

Answer 15.2 The answer to each question is given below.

1. For this problem, it will be assumed that each data page contains 20 relations per page.

   (a) $sal > 100$ For this condition, a filescan would probably be best, since a clustered index does not exist on *sal*. Using the unclustered index would accrue a cost of 10,000 pages * $\frac{20bytes}{100bytes}$ * 0.1 for the B+ index scan plus 10,000 pages * 20 tuples per page * 0.1 for the lookup = 22000, and would be inferior to the filescan cost of 10000.

   (b) $age = 25$ The clustered B+ tree index would be the best option here, with a cost of 2 (lookup) + 10000 pages * 0.1 (selectivity) + 10,000 * 0.2 (reduction) * 0.1 = 1202. Although the hash index has a lesser lookup time, the potential number of record lookups (10000 pages * 0.1 * 20 tuples per page = 20000) renders the clustered index more efficient.

   (c) $age > 20$ Again the clustered B+ tree index is the best of the options presented; the cost of this is 2 (lookup) + 10000 pages * 0.1 (selectivity)+ 200 = 1202.

   (d) $eid = 1000$ Since *eid* is a candiate key, one can assume that only one record will be in each bucket. Thus, the total cost is roughly 1.2 (lookup) + 1 (record access) which is 2 or 3.

   (e) $sal > 200 \wedge age > 30$ This query is similar to the $age > 20$ case if the $age > 30$ clause is examined first. Then, the cost is again 1202.

(f) $sal > 200 \wedge age = 20$ Similar to the previous part, the cost for this case using the clustered B+ index on $< age, sal >$ is smaller, since only 10 % of all relations fulfill $sal > 200$. Assuming a linear distribution of values for sal for age, one can assume a cost of 2 (lookup) + 10000 pages * 0.1 (selectivity for age) * 0.1 (selectivity for sal) + 10,000 * 0.4 * 0.1 * 0.1 = 142.

(g) $sal > 200 \wedge title = "CFO"$ In this case, the filescan is the best available method to use, with a cost of 10000.

(h) $sal > 200 \wedge age > 30 \wedge title = "CFO"$ Here an age condition is present, so the clustered B+ tree index on $< age, sal >$ can be used. Here, the cost is 2 (lookup) + 10000 pages * 0.1 (selectivity) = 1002.

2. (a) $sal > 100$ Since the result desired is only the average salary, an index-only scan can be performed using the unclusterd B+ tree on sal for a cost of 2 (lookup) + 10000 * 0.1 * 0.2 (due to smaller index tuples) = 202.

(b) $age = 25$ For this case, the best option is to use the clustered index on $< age, sal >$, since it will avoid a relational lookup. The cost of this operation is 2 (B+ tree lookup) + 10000 * 0.1 * 0.4 (due to smaller index tuple sizes) = 402.

(c) $age > 20$ Similar to the $age = 25$ case, this will cost 402 using the clustered index.

(d) $eid = 1000$ Being a candidate key, only one relation matching this should exist. Thus, using the hash index again is the best option, for a cost of 1.2 (hash lookup) + 1 (relation retrieval) = 2.2.

(e) $sal > 200 \wedge age > 30$ Using the clustered B+ tree again as above is the best option, with a cost of 402.

(f) $sal > 200 \wedge age = 20$ Similarly to the $sal > 200 \wedge age = 20$ case in the previous problem, this selection should use the clustered B+ index for an index only scan, costing 2 (B+ lookup) + 10000 * 0.1 (selectivity for age) * 0.1 (selectivity for sal) * 0.4 (smaller tuple sizes, index-only scan) = 42.

(g) $sal > 200 \wedge title = "CFO"$ In this case, an index-only scan may not be used, and individual relations must be retrieved from the data pages. The cheapest method available is a simple filescan, with a cost of 10000 I/Os.

(h) $sal > 200 \wedge age > 30 \wedge title = "CFO"$ Since this query includes an age restriction, the clustered B+ index over $< age, sal >$ can be used; however, the inclusion of the title field precludes an index-only query. Thus, the cost will be 2 (B+ tree lookup) + 10000 * 0.1 (selectivity on age)+ 10,000 * 0.1 * 0.4 = 1402 I/Os.

3. (a) $sal > 100$ The best method in terms of I/O cost requires usage of the clustered B+ index over $< age, sal >$ in an index-only scan. Also, this assumes the ablility to keep a running average for each age category. The total cost of this plan is 2 (lookup on B+ tree, find min entry) + 10000 * 0.4 (index-only

scan) = 4002. Note that although *sal* is part of the key, since it is not a *prefix* of the key, the entire list of pages must be scanned.

(b) $age = 25$ Again, the best method is to use the clustered B+ index in an index-only scan. For this selection condition, this will cost 2 (*age* lookup in B+ tree) + 10000 pages * 0.1 (selectivity on *age*) * 0.4 (index-only scan, smaller tuples, more per page, etc.) = 2 + 400 = 402.

(c) $age > 20$ This selection uses the same method as the previous condition, the clustered B+ tree index over $< age, sal >$ in an index-only scan, for a total cost of 402.

(d) $eid = 1000$ As in previous questions, *eid* is a candidate field, and as such should have only one match for each equality condition. Thus, the hash index over *eid* should be the most cost effective method for selecting over this condition, costing 1.2 (hash lookup) + 1 (relation retrieval) = 2.2.

(e) $sal > 200 \wedge age > 30$ This can be done with the clustered B+ index and an index-only scan over the $< age, sal >$ fields. The total estimated cost is 2 (B+ lookup) + 10000 pages * 0.1 (selectivity on *age*) * 0.4 (index-only scan) = 402.

(f) $sal > 200 \wedge age = 20$ This is similar to the previous selection conditions, but even cheaper. Using the same index-only scan as before (the clustered B+ index over $< age, sal >$), the cost should be 2 + 10000 * 0.4 * 0.1 (*age* selectivity) * 0.1 (*sal* selectivity) = 42.

(g) $sal > 200 \wedge title = "CFO"$ Since the results must be grouped by age, a scan of the clustered $< age, sal >$ index, getting each result from the relation pages, should be the cheapest. This should cost 2 + 10000 * .4 + 10000 * tuples per page * 0.1 + 5000 * 0.1 (index scan cost) = 2 + 1000(4 + tuples per page). Assuming the previous number of tuples per page (20), the total cost would be 24002. Sorting the filescan alone, would cost 40000 I/Os. However, if the tuples per page is greater than 36, then sorting the filescan would be the best, with a cost of 40000 + 6000 (secondary scan, with the assumption that unneeded attributes of the relation have been discarded).

(h) $sal > 200 \wedge age > 30 \wedge title = "CFO"$ Using the clustered B+ tree over $< age, sal >$ would accrue a cost of 2 + 10000 * 0.1 (selectivity of *age*) + 5000 * 0.1 = 1502 lookups.

4. (a) $sal > 100$ The best operation involves an external merge sort over $< sal, age >$, discarding unimportant attributes, followed by a binary search to locate minimum $sal < 100$ and a scan of the remainder of the sort. This costs a total of 16000 (sort) + 12 (binary search) + 10000 * 0.4 (smaller tuples) * 0.1 (selectivity of *sal*) + 2 = 16000 + 4000 + 12 + 400 + 2= 16414.

(b) $age = 25$ The most cost effective technique here employs sorting the clustered B+ index over $< age, sal >$, as the grouping requires that the output be sorted. An external merge sort with 11 buffer pages would require 16000. Totalled, the cost equals 16000 (sort) + 10000 * 0.4 = 20000.

(c) $age > 20$ This selection criterion works similarly to the previous one, in that an external merge over $< age, sal >$ is required, using the clustered index provided as the pages to sort. The final cost is the same, 20000.

(d) $eid = 1000$ Being a candidate key, only one relation should match with a given *eid* value. Thus, the estimated cost should be 1.2 (hash lookup) + 1 (relation retrieval).

(e) $sal > 200 \wedge age > 30$ This case is similar to the $sal > 100$ case above, cost = 16412.

(f) $sal > 200 \wedge age = 20$ Again, this case is also similar to the $sal > 100$ case, cost = 16412.

(g) $sal > 200 \wedge title = "CFO"$ The solution to this case greatly depends of the number of tuples per page. Assuming a small number of tuples per page, the cheapest route is to use the B+ tree index over *sal*, getting each index. The total cost for this is 2 (lookup, $sal > 200$) + 10000 * .2 (smaller size) * .1 (selectivity) + 10000 * .1 (selectivity) * tuples per page. The solution to this case is similar to that of the other requiring sorts, but at a higher cost. Since the sort can't be preformed over the clustered B+ tree in this case, the sort costs 40000 I/Os. Thus, for tuples per page ¡ 40, the B+ index method is superior, otherwise, the sort solution is cheaper.

(h) $sal > 200 \wedge age > 30 \wedge title = "CFO"$ This solution is the same as the previous, since either the index over *sal* or an external sort must be used. The cost is the cheaper of 2 + 1000 * (.2 + tuples per page) [index method] and 40000 [sort method].

5. (a) $sal > 200 \vee age = 20$ In this case, a filescan would be the most cost effective, because the most cost effective method for satisfying $sal > 200$ alone is a filescan.

(b) $sal > 200 \vee title = "CFO"$ Again a filescan is the better alternative here, since no index at all exists for *title*.

(c) $title = "CFO" \wedge ename = "Joe"$ Even though this condition is a conjunction, the filescan is still the best method, since no indexes exist on either *title* or *ename*.

Exercise 15.3 For each of the following SQL queries, for each relation involved, list the attributes that must be examined to compute the answer. All queries refer to the following relations:

Emp(*eid:* `integer`, *did:* `integer`, *sal:* `integer`, *hobby:* `char(20)`)
Dept(*did:* `integer`, *dname:* `char(20)`, *floor:* `integer`, *budget:* `real`)

1. `SELECT COUNT(*) FROM` Emp E, Dept D `WHERE` E.did = D.did

2. `SELECT MAX`(E.sal) `FROM` Emp E, Dept D `WHERE` E.did = D.did
3. `SELECT MAX`(E.sal) `FROM` Emp E, Dept D `WHERE` E.did = D.did `AND` D.floor = 5
4. `SELECT` E.did, `COUNT`(*) `FROM` Emp E, Dept D `WHERE` E.did = D.did `GROUP BY` D.did
5. `SELECT` D.floor, `AVG`(D.budget) `FROM` Dept D `GROUP BY` D.floor `HAVING COUNT`(*) > 2
6. `SELECT` D.floor, `AVG`(D.budget) `FROM` Dept D `GROUP BY` D.floor `ORDER BY` D.floor

Answer 15.3 The answer to each question is given below.

1. E.did, D.did
2. E.sal, E.did, D.did
3. E.sal, E.did, D.did, D.floor
4. E.did, D.did
5. D.floor, D.budget
6. D.floor, D.budget

Exercise 15.4 You are given the following information:

Executives has attributes *ename*, *title*, *dname*, and *address*; all are string fields of the same length.
The *ename* attribute is a candidate key.
The relation contains 10,000 pages.
There are 10 buffer pages.

1. Consider the following query:

   `SELECT` E.title, E.ename `FROM` Executives E `WHERE` E.title='CFO'

   Assume that only 10% of Executives tuples meet the selection condition.

   (a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan? (In this and subsequent questions, be sure to describe the plan you have in mind.)

   (b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on *ename* is (the only index) available. What is the cost of the best plan?

(d) Suppose that a clustered B+ tree index on *address* is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ is (the only index) available. What is the cost of the best plan?

2. Suppose that the query is as follows:

```
SELECT E.ename FROM Executives E WHERE E.title='CFO' AND E.dname='Toy'
```

Assume that only 10% of Executives tuples meet the condition $E.title = 'CFO'$, only 10% meet $E.dname = 'Toy'$, and that only 5% meet both conditions.

(a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(b) Suppose that a clustered B+ tree index on *dname* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on $\langle title, dname \rangle$ is (the only index) available. What is the cost of the best plan?

(d) Suppose that a clustered B+ tree index on $\langle title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle dname, title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

(f) Suppose that a clustered B+ tree index on $\langle ename, title, dname \rangle$ is (the only index) available. What is the cost of the best plan?

3. Suppose that the query is as follows:

```
SELECT E.title, COUNT(*) FROM Executives E GROUP BY E.title
```

(a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on *ename* is (the only index) available. What is the cost of the best plan?

(d) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

4. Suppose that the query is as follows:

```
SELECT E.title, COUNT(*) FROM Executives E
WHERE E.dname > 'W%' GROUP BY E.title
```

Assume that only 10% of Executives tuples meet the selection condition.

(a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan? If an additional index (on any search key you want) is available, would it help produce a better plan?

(b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on *dname* is (the only index) available. What is the cost of the best plan? If an additional index (on any search key you want) is available, would it help to produce a better plan?

(d) Suppose that a clustered B+ tree index on $\langle dname, title\rangle$ is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle title, dname\rangle$ is (the only index) available. What is the cost of the best plan?

Answer 15.4 1. (a) The best plan, a B+ tree search, would involve using the B+ tree to find the first *title* index such that *title*="CFO", cost = 2. Then, due to the clustering of the index, the relation pages can be scanned from that index's reference, cost = 10000 * 10% + 2500 * 10% (Scanning the index) = 1000 + 250 + 2 = 1252 (total cost) .

(b) An unclustered index would preclude the low cost of the previous plan and necessitate the choice of a simple filescan, cost = 10000, as the best.

(c) Due to the `WHERE` clause, the clustered B+ index on ename doesn't help at all. The best alternative is to use a filescan, cost = 10000.

(d) Again, as in the previous answer, the best choice is a filescan, cost = 10000.

(e) Although the order of the B+ index key makes the tree much less useful, the leaves can still be scanned in an index-only scan, and the increased number of tuples per page lowers the I/O cost. Cost = 10000 * .5 = 5000.

2. (a) A clustered index on *title* would allow scanning of only the 10% of the tuples desired. Thus the total cost is 2 (lookup) + 10000 * 10% + 2500 * 10%= 1252.

(b) A clustered index on *dname* works functionally in the same manner as that in the previous question, for a cost 1002 + 250 = 1252 . The *ename* field still must be retrieved from the relation data pages.

(c) In this case, using the index lowers the cost of the query slightly, due to the greater selectivity of the combined query and to the search key taking advantage of it. The total cost = 2 (look up) + 10000 * 5% + 5000 * 5% =752.

(d) Although this index does contain the output field, the *dname* still must be retrieved from the relational data pages, for a cost of 2 (lookup) + 10000 * 10% + 5000 * 10%= 1502.

(e) Since this index contains all three indexes needed for an index-only scan, the cost drops to 2 (look up) + 10000 * 5% * .75 (smaller size) = 402.

(f) Finally, in this case, the prefix cannot be matched with the equality information in the `WHERE` clause, and thus a scan would be the superior method of retrieval. However, as the clustered B+ tree's index contains all the indexes needed for the query and has a smaller tuple, scanning the leaves of the B+ tree is the best plan, costing 10000 * .75 = 7500 I/Os.

3. (a) Since *title* is the only attribute required, an index-only scan could be performed, with a running counter. This would cost 10000 * .25 (index-only scan, smaller tuples) = 2500.

(b) Again, as the index contains the only attribute of import, an index-only scan could again be performed, for a cost of 2500.

(c) This index is useless for the given query, and thus requires a sorting of the file, costing 10000 + 3 * 2 * (2500). Finally, a scan of this sorted result will allow us to answer the query, for a cost of 27500.

(d) This is similar to the previous part, except that the initial scan requires fewer I/Os if the leaves of the B+ tree are scanned instead of the data file. Cost = 5000 + 3 * 2 * (2500) = 22500.

(e) The clustered B+ index given contains all the information required to perform an index-only scan, at a cost of 10000 * .5 (tuple size) = 5000.

4. (a) Using a clustered B+ tree index on *title*, the cost of the given query is 10000 I/Os. The addition of another index would not lower the cost of any evaluation strategy that also utilizes the given index. However, the cost of the query is significantly cheaper if a clustered index on *dname*, *title* is available and is used by itself, and if added would reduce the cost of the best plan to 1500. (See below.)

(b) The cheapest plan here involves simply sorting the file, at a cost of 10000 + 2 * 2 * (10000 *.25 (size reduction due to elimination of unwanted attributes; the selection can be checked on the fly and we only need to retail the *title* field)) = 20000.

(c) The optimal plan with the indexes given involves scanning the *dname* index and sorting the (records consisting of the) *title* field of records that satisfy the `WHERE` condition. This would cost 2500 * 10 % [scanning relevant portion of index] + 10000 * 10% [retrieving qualifying records] + 10000 * 10% * .25 (reduction in size) [writing out *title* records] + 3 * 250 [sorting *title* records; result is not written out]. This is a total of 2250.

(d) We can simply scan the relevant portion of the index, discard tuples that don't satisfy the `WHERE` condition, and write out the *title* fields of qualifying records. The *title* records must then be sorted. Cost = 5000 * 10% + 10000 * 10% * .25 + 3 * 250 = 1500.

(e) A clustered index on *title*, *dname* supports an index-only scan costing 10000 * .5 = 5000.

Exercise 15.5 Consider the query $\pi_{A,B,C,D}(R \bowtie_{A=C} S)$. Suppose that the projection routine is based on sorting and is smart enough to eliminate all but the desired attributes during the initial pass of the sort and also to toss out duplicate tuples on-the-fly while sorting, thus eliminating two potential extra passes. Finally, assume that you know the following:

R is 10 pages long, and R tuples are 300 bytes long.
S is 100 pages long, and S tuples are 500 bytes long.
C is a key for S, and A is a key for R.
The page size is 1024 bytes.
Each S tuple joins with exactly one R tuple.
The combined size of attributes A, B, C, and D is 450 bytes.
A and B are in R and have a combined size of 200 bytes; C and D are in S.

1. What is the cost of writing out the final result? (As usual, you should ignore this cost in answering subsequent questions.)
2. Suppose that three buffer pages are available, and the only join method that is implemented is simple (page-oriented) nested loops.
   (a) Compute the cost of doing the projection followed by the join.
   (b) Compute the cost of doing the join followed by the projection.
   (c) Compute the cost of doing the join first and then the projection on-the-fly.
   (d) Would your answers change if 11 buffer pages were available?

Answer 15.5 The answer to each question is given below.

1. From the given information, we know that R has 30 tuples (10 pages of 3 records each), and S has 200 tuples (100 pages of 2 records each). Since every S tuple joins with exaclty one R tuple, there can be at most 200 tuples after the join. Since the size of the result is 450 bytes/record, 2 records will fit on a page. This means 200 / 2 = 100 page writes are needed to write the result to disk.
2. (a) Cost of projection followed by join: The projection is sort-based, so we must sort relation S, which contains attributes C and D. Relation S has 100 pages,

and we have 3 buffer pages, so the sort cost is $200 * \lceil log2(100) \rceil = 200 * 7 = 1400$.

Assume that 1/10 of the tuples are removed as duplicates, so that there are 180 remaining tuples of S, each of size 150 bytes (combined size of attributes C, D). Therefore, 6 tuples fit on a page, so the resulting size of the inner relation is 30 pages.

The projection on R is calculated similarly: R has 10 pages, so the sort will cost $30 * \lceil log2(10) \rceil = 30 * 4 = 120$. If 1/10 are not duplicates, then there are 27 tuples remaining, each of size 200 bytes. Therefore, 5 tuples fit on a page so the resulting size of the outer relation is 6 pages.

The cost using SNL is (6 + 6*30) = 186 I/Os, for a total cost of 1586 I/Os.

(b) Cost of join followed by projection:

SNL join is (10 + 10*100) = 1010 I/Os, and results in 200 tuples, each of size 800 bytes. Thus, only one result tuple fits on a page, and we have 200 pages.

The projection is a sort using 3 buffer pages, and in the first pass unwanted attributes are eliminated on-the-fly to produce tuples of size 450 bytes, i.e., 2 tuples per page. Thus, 200 pages are scanned and 100 pages written in the first pass in 33 runs of 3 pages each and 1 run of a page. These runs are merged pairwise in 6 additional passes for a total projection cost of 200+100+2*6*100=1500 I/Os. This includes the cost of writing out the result of 100 pages; removing this cost and adding the cost of the join step, we obtain a total cost of 2410 I/Os.

(c) Cost of join and projection on the fly:

This means that the projection cost is 0, so the only cost is the join, which we know from above is 1010 I/Os.

(d) If we had 11 buffer pages, then the projection sort could be done log10 instead of log2.

Exercise 15.6 Briefly answer the following questions:

1. Explain the role of relational algebra equivalences in the System R optimizer.
2. Consider a relational algebra expression of the form $\sigma_c(\pi_l(R \times S))$. Suppose that the equivalent expression with selections and projections pushed as much as possible, taking into account only relational algebra equivalences, is in one of the following forms. In each case give an illustrative example of the selection conditions and the projection lists (c, l, $c1$, $l1$, etc.).
   (a) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times S)$.
   (b) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times \sigma_{c2}(S))$.
   (c) *Equivalent maximally pushed form:* $\sigma_c(\pi_{l1}(\pi_{l2}(R) \times S))$.

(d) *Equivalent maximally pushed form:* $\sigma_{c1}(\pi_{l1}(\sigma_{c2}(\pi_{l2}(R)) \times S))$.

(e) *Equivalent maximally pushed form:* $\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times S))$.

(f) *Equivalent maximally pushed form:* $\pi_l(\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times S)))$.

Answer 15.6 The answer to each question is given below.

1. Relational algebra equivalences are used to modify the query in hope of finding an optimal plan.

2. (a) $\sigma_{A=1}(\pi_{ABCD}(R \times S))$
   $= \pi_{ABCD}(\sigma_{A=1}(R) \times S)$

   (b) $\sigma_{A=1,C=2}(\pi_{ABCD}(R \times S))$
   $= \pi_{ABCD}(\sigma_{A=1,C=2}(R \times S))$
   $= \pi_{ABCD}(\sigma_{A=1}(R) \times \sigma_{C=2}(S))$

   (c) $\sigma_{C=5}(\pi_{BC}(R \times S))$
   $= \sigma_{C=5}(\pi_C(\pi_B(R) \times S))$

   (d) $\sigma_{B=1,C=3}(\pi_{BC}(R \times S))$
   $= \sigma_{B=1,C=3}(\pi_C(\pi_B(R) \times S))$
   $= \sigma_{B=1}(\sigma_{C=3}(\pi_C(\pi_B(R) \times S)))$
   $= \sigma_{B=1}(\pi_C(\sigma_{C=3}(\pi_B(R) \times S)))$

   (e) $\sigma_{B=1,C=3}(\pi_{BC}(R \times S))$
   $= \sigma_{B=1,C=3}(\pi_C(\pi_B(R) \times S))$
   $= \sigma_{B=1}(\sigma_{C=3}(\pi_C(\pi_B(R) \times S)))$
   $= \sigma_{C=3}(\pi_C(\pi_B(\sigma_{B=1}(R)) \times S))$

   (f) $\sigma_{A=1,B=D}(\pi_{BC}(R \times S))$
   $= \pi_{BC}(\sigma_{B=D}(\sigma_{A=1}(R) \times S)) = \pi_{BC}(\sigma_{B=D}(\pi_{BCD}(\sigma_{A=1}(R) \times S))$

Exercise 15.7 Consider the following relational schema and SQL query. The schema captures information about employees, departments, and company finances (organized on a per department basis).

Emp(*eid:* `integer`, *did:* `integer`, *sal:* `integer`, *hobby:* `char(20)`)
Dept(*did:* `integer`, *dname:* `char(20)`, *floor:* `integer`, *phone:* `char(10)`)
Finance(*did:* `integer`, *budget:* `real`, *sales:* `real`, *expenses:* `real`)

Consider the following query:

```
SELECT D.dname, F.budget
FROM   Emp E, Dept D, Finance F
WHERE  E.did=D.did AND D.did=F.did AND D.floor=1
       AND E.sal ≥ 59000 AND E.hobby = 'yodeling'
```

1. Identify a relational algebra tree (or a relational algebra expression if you prefer) that reflects the order of operations a decent query optimizer would choose.

2. List the join orders (i.e., orders in which pairs of relations can be joined to compute the query result) that a relational query optimizer will consider. (Assume that the optimizer follows the heuristic of never considering plans that require the computation of cross-products.) Briefly explain how you arrived at your list.

3. Suppose that the following additional information is available: Unclustered B+ tree indexes exist on *Emp.did*, *Emp.sal*, *Dept.floor*, *Dept.did*, and *Finance.did*. The system's statistics indicate that employee salaries range from 10,000 to 60,000, employees enjoy 200 different hobbies, and the company owns two floors in the building. There are a total of 50,000 employees and 5,000 departments (each with corresponding financial information) in the database. The DBMS used by the company has just one join method available, index nested loops.

   (a) For each of the query's base relations (Emp, Dept, and Finance) estimate the number of tuples that would be initially selected from that relation if all of the non-join predicates on that relation were applied to it before any join processing begins.

   (b) Given your answer to the preceding question, which of the join orders considered by the optimizer has the lowest estimated cost?

Answer 15.7 The answer to each question is given below.

1. $$\pi_{D.dname,F.budget}(((\pi_{E.did}(\sigma_{E.sal>=59000,E.hobby="yodelling"}(E))) \bowtie \pi_{D.did,D.dname}(\sigma_{D.floor=1}(D))) \bowtie \pi_{F.budget,F.did}(F))$$

2. There are 2 join orders considered, assuming that the optimizer only consider left-deep joins and ignores cross-products: (D,E,F) and (D,F,E)

3. (a) The answer to each relation is given below.
   - Emp: card = 50,000, E.sal $\geq$ 59,000, E.hobby = "yodelling" resulting card = 50000 * 1/50 * 1/200 = 5
   - Dept: card = 5000, D.floor = 1
     resulting card = 5000 * 1/2 = 2500
   - Finance: card = 5000, there are no non-join predicates
     resulting card = 5000

   (b) Consider the following join methods on the following left-deep tree: $(E \bowtie D) \bowtie F)$.

   The tuples from E will be pipelined, no temporary relations are created.

First, retrieve the tuples from E with salary $\geq$ 59,000 using the B-tree index on salary; we estimate 1000 such tuples will be found, with a cost of 1 tree traversal + the cost of retrieving the 1000 tuples (since the index is unclustered) = 3+1000 = 1003. Note, we ignore the cost of scanning the leaves.

Of these 1000 retrieved tuples, on the fly select only those that have hobby = "yodelling", we estimate there will be 5 such tuples.

Pipeline these 5 tuples one at a time to D, and using the B-tree index on D.did and the fact the D.did is a key, we can find the matching tuples for the join by searching the Btree and retrieving at most 1 matching tuple, for a total cost of 5(3 + 1) = 20. The resulting cardinality of this join is at most 5.

Pipeline the estimated 3 tuples of these 5 that have D.floor=1 1 up to F, and use the Btree index on F.did and the fact that F.did is a key to retrieve at most 1 F tuple for each of the 3 pipelined tuples. This costs at most 3(3+1) = 12.

Ignoring the cost of writing out the final result, we get a total cost of 1003+20+12 = 1035.

Exercise 15.8 Consider the following relational schema and SQL query:

Suppliers(*sid:* `integer`, *sname:* `char(20)`, *city:* `char(20)`)
Supply(*sid:* `integer`, *pid:* `integer`)
Parts(*pid:* `integer`, *pname:* `char(20)`, *price:* `real`)

```
SELECT S.sname, P.pname
FROM   Suppliers S, Parts P, Supply Y
WHERE  S.sid = Y.sid AND Y.pid = P.pid AND
       S.city = 'Madison' AND P.price ≤ 1,000
```

1. What information about these relations does the query optimizer need to select a good query execution plan for the given query?
2. How many different join orders, assuming that cross-products are disallowed, does a System R style query optimizer consider when deciding how to process the given query? List each of these join orders.
3. What indexes might be of help in processing this query? Explain briefly.
4. How does adding `DISTINCT` to the `SELECT` clause affect the plans produced?
5. How does adding `ORDER BY` *sname* to the query affect the plans produced?
6. How does adding `GROUP BY` *sname* to the query affect the plans produced?

Answer 15.8 The answer to each question is given below.

1. The query optimizer will need information such as what indexes exist (and what type) on: S.sid, Y.sid, Y.pid, P.pid, S.city, P.price It will also need statistics about the database such as low/high index values and distribution between fields.
2. Only left-deep plans are allowed: $(S \bowtie Y) \bowtie P)$ and $((Y \bowtie P) \bowtie S$.
3. A sorted, clustered index on P.price would be useful for range retrieval. A B+ Tree index on S.sid, Y.sid, Y.pid, P.pid could be used in an index-only sort-merge.
4. To support the `DISTINCT` selection, we must sort the results (unless they already are in sorted order) and scan for multiple occurences. Different sorted orders are known as "interestin orders" in the System R optimizer, and these orders are considered when determining the plan.
5. The `ORDER BY` *sname* selection would have effects similar to question 4. The optimizer would consider plans which left *sname* ordered as a side effect and plans which ordered *sname* directly.
6. The `GROUP BY` *sname* clause requires us to sort the results of the earlier steps on *sname*, and to compute some aggregate (e.g., `SUM`) for each group (i.e., set of tuples with the same *sname* value).

Exercise 15.9 Consider the following scenario:

Emp(*eid:* `integer`, *sal:* `integer`, *age:* `real`, *did:* `integer`)
Dept(*did:* `integer`, *projid:* `integer`, *budget:* `real`, *status:* `char(10)`)
Proj(*projid:* `integer`, *code:* `integer`, *report:* `varchar`)

Assume that each Emp record is 20 bytes long, each Dept record is 40 bytes long, and each Proj record is 2000 bytes long on average. There are 20,000 tuples in Emp, 5000 tuples in Dept (note that *did* is not a key), and 1000 tuples in Proj. Each department, identified by *did*, has 10 projects on average. The file system supports 4000 byte pages, and 12 buffer pages are available. All following questions are based on this information. You can assume uniform distribution of values. State any additional assumptions. The cost metric to use is *the number of page I/Os*. Ignore the cost of writing out the final result.

1. Consider the following two queries: "Find all employees with $age = 30$" and "Find all projects with $code = 20$." Assume that the number of qualifying tuples is the same in each case. If you are building indexes on the selected attributes to speed up these queries, for which query is a *clustered* index (in comparison to an *unclustered* index) more important?

2. Consider the following query: "Find all employees with $age > 30$." Assume that there is an unclustered index on *age*. Let the number of qualifying tuples be N. For what values of N is a sequential scan cheaper than using the index?

3. Consider the following query:

```
SELECT *
FROM   Emp E, Dept D
WHERE  E.did=D.did
```

   (a) Suppose that there is a clustered hash index on *did* on Emp. List all the plans that are considered and identify the plan with the lowest estimated cost.

   (b) Assume that both relations are sorted on the join column. List all the plans that are considered and show the plan with the lowest estimated cost.

   (c) Suppose that there is a clustered B+ tree index on *did* on Emp and Dept is sorted on *did*. List all the plans that are considered and identify the plan with the lowest estimated cost.

4. Consider the following query:

```
SELECT      D.did, COUNT(*)
FROM        Dept D, Proj P
WHERE       D.projid=P.projid
GROUP BY    D.did
```

   (a) Suppose that no indexes are available. Show the plan with the lowest estimated cost.

   (b) If there is a hash index on *P.projid* what is the plan with lowest estimated cost?

   (c) If there is a hash index on *D.projid* what is the plan with lowest estimated cost?

   (d) If there is a hash index on *D.projid* and *P.projid* what is the plan with lowest estimated cost?

   (e) Suppose that there is a clustered B+ tree index on *D.did* and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

   (f) Suppose that there is a clustered B+ tree index on *D.did*, a hash index on *D.projid*, and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

   (g) Suppose that there is a clustered B+ tree index on ⟨*D.did, D.projid*⟩ and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

   (h) Suppose that there is a clustered B+ tree index on ⟨*D.projid, D.did*⟩ and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

5. Consider the following query:

```
SELECT      D.did, COUNT(*)
FROM        Dept D, Proj P
WHERE       D.projid=P.projid AND D.budget>99000
GROUP BY    D.did
```

Assume that department budgets are uniformly distributed in the range 0 to 100,000.

(a) Show the plan with lowest estimated cost if no indexes are available.

(b) If there is a hash index on *P.projid* show the plan with lowest estimated cost.

(c) If there is a hash index on *D.budget* show the plan with lowest estimated cost.

(d) If there is a hash index on *D.projid* and *D.budget* show the plan with lowest estimated cost.

(e) Suppose that there is a clustered B+ tree index on ⟨*D.did,D.budget*⟩ and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

(f) Suppose there is a clustered B+ tree index on *D.did*, a hash index on *D.budget*, and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

(g) Suppose there is a clustered B+ tree index on ⟨*D.did, D.budget, D.projid*⟩ and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

(h) Suppose there is a clustered B+ tree index on ⟨*D.did, D.projid, D.budget*⟩ and a hash index on *P.projid*. Show the plan with the lowest estimated cost.

6. Consider the following query:

```
SELECT E.eid, D.did, P.projid
FROM   Emp E, Dept D, Proj P
WHERE  E.sal=50,000 AND D.budget>20,000
       E.did=D.did AND D.projid=P.projid
```

Assume that employee salaries are uniformly distributed in the range 10,009 to 110,008 and that project budgets are uniformly distributed in the range 10,000 to 30,000. There is a clustered index on *sal* for Emp, a clustered index on *did* for Dept, and a clustered index on *projid* for Proj.

(a) List all the one-relation, two-relation, and three-relation subplans considered in optimizing this query.

(b) Show the plan with the lowest estimated cost for this query.

(c) If the index on Proj were unclustered, would the cost of the preceding plan change substantially? What if the index on Emp or on Dept were unclustered?

Answer 15.9 The reader should calculate actual costs of all alternative plans; in the answers below, we just outline the best plans without detailed cost calculations to prove that these are indeed the best plans.

1. The question specifies that the *number*, rather than the *fraction*, of qualifying tuples is identical for the two queries. Since *Emp* tuples are small, many will fit on a single page; conversely, few (just 2) of the large *Proj* tuples will fit on a page.

   Since we wish to minimize the number of page I/Os, it will be an advantage if the *Emp* tuples are clustered with respect to the *age* index (all matching tuples will be retrieved in a few page I/Os). Clustering is not as important for the *Proj* tuples since almost every matching tuple will require a page I/O, even with clustering.

2. The *Emp* relation occupies 100 pages. For an unclustered index retrieving N tuples requires N page I/Os. If more than 100 tuples match, the cost of fetching *Emp* tuples by following pointers in the index data entries exceeds the cost of sequential scan. Using the index also involves about 2 I/Os to get to the right leaf page, and the cost of fetching leaf pages that contain qualifying data entries; this makes scan better than the index with fewer than 100 matches.)

3. (a) One plan is to use (simple or blocked) NL join with E as the outer. Another plan is SM or Hash join. A third plan is to use D as the outer and to use INL; given the clustered hash index on E, this plan will likely be the cheapest.

   (b) The same plans are considered as before, but now, SM join is the best strategy because both relations are sorted on the join column (and all tuples of *Emp* are likely to join with some tuple of *Dept*, and must therefore be fetched at least once, even if INL is used).

   (c) The same plans are considered as before. As in the previous case, SM join is the best: the clustered B+ tree index on *Emp* can be used to efficiently retrieve *Emp* tuples in sorted order.

4. (a) BNL with *Proj* as the outer, followed by sorting on *did* to implement the aggregation. All attributes except *did* can be eliminated during the join but duplicates should not be eliminated!

   (b) Sort *Dept* on *did* first (all other attributes except *projid* can be projected out), then scan while probing *Proj* and counting tuples in each *did* group on-the-fly.

   (c) INL with *Dept* as inner, followed by sorting on *did* to implement the aggregation. Again, all attributes except *did* can be eliminated during the join but duplicates should not be eliminated!

   (d) As in the previous case, INL with *Dept* as inner, followed by sorting on *did* to implement the aggregation. Again, all attributes except *did* can be eliminated during the join but duplicates should not be eliminated!

(e) Scan *Dept* in *did* order using the clustered B+ tree index while probing *Proj* and counting tuples in each *did* group on-the-fly.

(f) Same as above.

(g) Scan the clustered B+ tree index using an index-only scan while probing *Proj* and counting tuples in each *did* group on-the-fly.

(h) Sort the data entries in the clustered B+ tree index on *Dept*, then scan while probing *Proj* and counting tuples in each *did* group on-the-fly.

5. (a) BNL with *Proj* as the outer with the selection applied on-the-fly, followed by sorting on *did* to implement the aggregation. All attributes except *did* can be eliminated during the join but duplicates should not be eliminated!

(b) Sort *Dept* on *did* first (while applying the selection and projecting out all other attributes except *projid* in the initial scan), then scan while probing *Proj* and counting tuples in each *did* group on-the-fly.

(c) Select *Dept* tuples using the index on *budget*, join using INL with *Proj* as inner, projecting out all attributes except *did*. Then sort to implement the aggregation.

(d) Same as the case with no index; this index does not help.

(e) Retrieve *Dept* tuples that satisfy the condition on *budget* in *did* order by using the clustered B+ tree index while probing *Proj* and counting tuples in each *did* group on-the-fly.

(f) Since the condition on *budget* is very selective, even though the index on *budget* is unclustered we retrieve *Dept* tuples using this index, project out the *did* and *projid* fields and sort them by *did*. Then we scan while probing *Proj* and counting tuple sin each *did* gorup on-the-fly.

(g) Use an index-only scan on the B+ tree and apply the condition on *budget*, while probing *Proj* and counting tuples in each *did* group on-the-fly. Notice that this plan is applicable even if the B+ tree index is *not* clustered. (Within each *did* group, can optimize search for data entries in the index that satisfy the *budget* condition, but this is a minor gain.)

(h) Use an index-only scan on the B+ tree and apply the condition on *budget*, while probing *Proj* and counting tuples in each *did* group on-the-fly.

6. (a) 1-relation subplans: Clustered index on *E.sal*; Scan *Dept*; and Scan *Proj*.
2-relation subplans: (i) Clustered index on *E.sal*, probe *Dept* using the index on *did*, apply predicate on *D.budget* and join. (ii) Scan *Dept*, apply predicate on *D.budget* and probe *Proj*. (iii) Scan *Proj*, probe *Dept* and apply predicate on *D.budget* and join.
3-relation subplans: Join *Emp* and *Dept* and probe *Proj*; Join *Dept* and *Proj* and probe *Emp*.

(b) The least cost plan is to use the index on $E.sal$ to eliminate most tuples, probe $Dept$ using the index on $D.did$, apply the predicate on $D.budget$, probe and join on $Proj.projid$.

(c) Unclustering the index on $Proj$ would increase the number of I/Os but not substantially since the total number of matching $Proj$ tuples to be retrieved is small.

16

OVERVIEW OF TRANSACTION MANAGEMENT

Exercise 16.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity, consistency, isolation, durability, schedule, blind write, dirty read, unrepeatable read, serializable schedule, recoverable schedule, avoids-cascading-aborts schedule.*
3. Describe Strict 2PL.
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?

Answer 16.1 The answer to each question is given below.

1. A *transaction* is an execution of a user program, and is seen by the DBMS as a series or list of actions. The actions that can be executed by a transaction include reads and writes of database objects, whereas actions in an ordinary program could involve user input, access to network devices, user interface drawing, etc.
2. Each term is described below.
   (a) *Atomicity* means a transaction executes when all actions of the transaction are completed fully, or none are. This means there are no partial transactions (such as when half the actions complete and the other half do not).
   (b) *Consistency* involves beginning a transaction with a 'consistent' database, and finishing with a 'consistent' database. For example, in a bank database, money should never be "created" or "deleted" without an appropriate deposit or withdrawal. Every transaction should see a consistent database.

(c) *Isolation* ensures that a transaction can run independently, without considering any side effects that other concurrently running transactions might have. When a database interleaves transaction actions for performance reasons, the database protects each transaction from the effects of other transactions.

(d) *Durability* defines the persistence of committed data: once a transaction commits, the data should persist in the database even if the system crashes before the data is written to non-volatile storage.

(e) A *schedule* is a series of (possibly overlapping) transactions.

(f) A *blind write* is when a transaction writes to an object without ever reading the object.

(g) A *dirty read* occurs when a transaction reads a database object that has been modified by another not-yet-committed transaction.

(h) An *unrepeatable read* occurs when a transaction is unable to read the same object value more than once, even though the transaction has not modified the value. Suppose a transaction T2 changes the value of an object A that has been read by a transaction T1 while T1 is still in progress. If T1 tries to read the value of A again, it will get a different result, even though it has not modified A.

(i) A *serializable schedule* over a set S of transactions is a schedule whose effect on any consistent database instance is identical to that of some complete serial schedule over the set of committed transactions in S.

(j) A *recoverable schedule* is one in which a transaction can commit only after all other transactions whose changes it has read have committed.

(k) A schedule that *avoids-cascading-aborts* is one in which transactions only read the changes of committed transactions. Such a schedule is not only recoverable, aborting a transaction can be accomplished without cascading the abort to other transactions.

3. *Strict 2PL* is the most widely used locking protocol where 1) A transaction requests a shared/exclusive lock on the object before it reads/modifies the object. 2) All locks held by a transaction are released when the transaction is completed.

4. The *phantom problem* is a situation where a transaction retrieves a collection of objects twice but sees different results, even though it does not modify any of these objects itself and follows the strict 2PL protocol. This problem usually arises in dynamic databases where a transaction cannot assume it has locked all objects of a given type (such as all sailors with rank 1; new sailors of rank 1 can be added by a second transaction after one transaction has locked all of the original ones).

   If the set of database objects is fixed and only the values of objects can be changed, the phantom problem cannot occur since one cannot insert new objects into the database.

Exercise 16.2 Consider the following actions taken by transaction $T1$ on database objects X and Y:

R(X), W(X), R(Y), W(Y)

1. Give an example of another transaction $T2$ that, if run concurrently to transaction T without some form of concurrency control, could interfere with $T1$.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Answer 16.2 The answer to each question is given below.

1. If the transaction T2 performed $W(Y)$ before T1 performed $R(Y)$, and then T2 aborted, the value read by T1 would be invalid and the abort would be cascaded to T1 (i.e. T1 would also have to abort).
2. Strict 2PL would require T2 to obtain an exclusive lock on Y before writing to it. This lock would have to be held until T2 committed or aborted; this would block T1 from reading Y until T2 was finished, thus there would be no interference.
3. Strict 2PL is popular for many reasons. One reason is that it ensures only 'safe' interleaving of transactions so that transactions are recoverable, avoid cascading aborts, etc. Another reason is that strict 2PL is very simple and easy to implement. The lock manager only needs to provide a lookup for exclusive locks and an atomic locking mechanism (such as with a semaphore).

Exercise 16.3 Consider a database with objects X and Y and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects X and Y and then writes object X. Transaction $T2$ reads objects X and Y and then writes objects X and Y.

1. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-read conflict.
2. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a read-write conflict.
3. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Answer 16.3 The answer to each question is given below.

1. The following schedule results in a write-read conflict:
   T2:R(X), T2:R(Y), T2:W(X), T1:R(X) ...
   T1:R(X) is a dirty read here.

2. The following schedule results in a read-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X) ...
   Now, T2 will get an unrepeatable read on X.

3. The following schedule results in a write-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X), T2:W(X) ...
   Now, T2 has overwritten uncommitted data.

4. Strict 2PL resolves these conflicts as follows:

   (a) In S2PL, T1 could not get a shared lock on X because T2 would be holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.

   (b) Here T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.

   (c) Same as above.

Exercise 16.4 We call a transaction that only reads database object a **read-only** transaction, otherwise the transaction is called a **read-write** transaction. Give brief answers to the following questions:

1. What is lock thrashing and when does it occur?

2. What happens to the database system throughput if the number of read-write transactions is increased?

3. What happens to the database system throughput if the number of read-only transactions is increased?

4. Describe three ways of tuning your system to increase transaction throughput.

Answer 16.4 The answer to each question is given below.

1. *Locking thrashing* occurs when the database system reaches to a point where adding another new active transaction actually reduces throughput due to competition for locking among all active transactions. Empirically, locking thrashing is seen to occur when 30% of active transactions are blocked.

2. If the number of read-write transaction is increased, the database system throughput will increase until it reaches the thrashing point; then it will decrease since read-write transactions require exclusive locks, thus resulting in less concurrent execution.

3. If the number of read-only transaction is increased, the database system throughput will also increase since read-only transactions require only shared locks. So we are able to have more concurrency and execute more transactions in a given time.

4. Throughput can be increased in three ways:

   (a) By locking the smallest sized objects possible.

   (b) By reducing the time that transaction hold locks.

   (c) By reducing hot spots, a database object that is frequently accessed and modified.

Exercise 16.5 Suppose that a DBMS recognizes *increment*, which increments an integer-valued object by 1, and *decrement* as actions, in addition to reads and writes. A transaction that increments an object need not know the value of the object; increment and decrement are versions of blind writes. In addition to shared and exclusive locks, two special locks are supported: An object must be locked in I mode before incrementing it and locked in D mode before decrementing it. An I lock is compatible with another I or D lock on the same object, but not with S and X locks.

1. Illustrate how the use of I and D locks can increase concurrency. (Show a schedule allowed by Strict 2PL that only uses S and X locks. Explain how the use of I and D locks can allow more actions to be interleaved, while continuing to follow Strict 2PL.)

2. Informally explain how Strict 2PL guarantees serializability even in the presence of I and D locks. (Identify which pairs of actions conflict, in the sense that their relative order can affect the result, and show that the use of S, X, I, and D locks according to Strict 2PL orders all conflicting pairs of actions to be the same as the order in some serial schedule.)

Answer 16.5 The answer to each question is given below.

1. Take the following two transactions as example:

   T1: Increment A, Decrement B, Read C;
   T2: Increment B, Decrement A, Read C

   If using only strict 2PL, all actions are versions of blind writes, they have to obtain exclusive locks on objects. Following strict 2PL, T1 gets an exclusive lock on A, if T2 now gets an exclusive lock on B, there will be a deadlock. Even if T1 is fast enough to have grabbed an exclusive lock on B first, T2 will now be blocked until T1 finishes. This has little concurrency. If I and D locks are used, since I and

D are compatible, T1 obtains an I-Lock on A, and a D-Lock on B; T2 can still obtain an I-Lock on B, a D-Lock on A; both transactions can be interleaved to allow maximum concurrency.

2. The pairs of actions which conflicts are:

   RW, WW, WR, IR, IW, DR, DW

   We know that strict 2PL orders the first 3 conflicts pairs of actions to be the same as the order in some serial schedule. We can also show that even in the presence of I and D locks, strict 2PL also orders the latter 4 pairs of actions to be the same as the order in some serial schedule. Think of an I (or D)lock under these circumstances as an exclusive lock, since an I(D) lock is not compatible with S and X locks anyway (ie. can't get a S or X lock if another transaction has an I or D lock). So serializability is guaranteed.

Exercise 16.6 Answer the following questions: SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. Consider the four SQL isolation levels. Describe which of the phenomena can occur at each of these isolation levels: *dirty read, unrepeatable read, phantom problem*.
2. For each of the four isolation levels, give examples of transactions that could be run safely at that level.
3. Why does the access mode of a transaction matter?

Answer 16.6 The answer to each question is given below.

1.

Level	Dirty Read	Unrepeatable read	phantom problem
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

2. (a) A SERIALIZABLE transaction achieves the highest degree of isolation from the effects of other transactions. It obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged and hold them until the end. Thus it is immune to all three phenomena above. We can safely run transactions like (assuming T2 inserts new objects):
   T1: R(X), R(Y), W(X). W(Y) Commit
   T2: W(X), W(Y) Commit

(b) A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it locks only objects, not sets of objects. We can safely run transactions like (assuming T1 or T2 does not insert any new objects):
T1: R(X), R(Y), W(X), W(Y), Commit
T2: R(X), W(X), R(Y), W(Y), Commit

(c) A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared lock before reading, but it releases it immediately. Thus it is only immune to dirty read. So we can safely run transactions like (assuming T1 or T2 does not insert any new objects):
T1:R(X), W(X), Commit
T2:R(X), W(X), Commit

(d) A READ UNCOMMITTED transaction can never make any lock requests, thus it is vulnerable to dirty read, unrepeatable read and phantom problem. Transactions which run safely at this level can only read objects from the database:
T1:R(X), R(Y), Commit
T2:R(Y), R(X), Commit

3. Access mode of a transaction tells what kind of lock is needed by the transaction. If the transaction is with READ ONLY access mode, only shared locks need to be obtained, thereby increases concurrency.

Exercise 16.7 Consider the university enrollment database schema:

Student(*snum:* `integer`, *sname:* `string`, *major:* `string`, *level:* `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid:* `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

For each of the following transactions, state the SQL isolation level you would use and explain why you chose it.

1. Enroll a student identified by her *snum* into the class named 'Introduction to Database Systems'.

2. Change enrollment for a student identified by her *snum* from one class to another class.
3. Assign a new faculty member identified by his *fid* to the class with the least number of students.
4. For each class, show the number of students enrolled in the class.

Answer 16.7 The answer to each question is given below.

1. Because we are inserting a new row in the table *Enrolled*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.
2. Because we are updating one existing row in the table *Enrolled*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.
3. To prevent other transactions from inserting or updating the table *Enrolled* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.
4. same as above.

Exercise 16.8 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it.

1. A transaction that adds a new part to a supplier's catalog.
2. A transaction that increases the price that a supplier charges for a part.
3. A transaction that determines the total number of items for a given supplier.
4. A transaction that shows, for each part, the supplier that supplies the part at the lowest price.

Answer 16.8 The answer to each question is given below.

1. Because we are inserting a new row in the table *Catalog*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.

2. Because we are updating one existing row in the table *Catalog*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.

3. To prevent other transactions from inserting or updating the table *Catalog* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.

4. same as above.

Exercise 16.9 Consider a database with the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

Consider the transactions $T1$ and $T2$. $T1$ always has SQL isolation level `SERIALIZABLE`. We first run $T1$ concurrently with $T2$ and then we run $T1$ concurrently with $T2$ but we change the isolation level of $T2$ as specified below. Give a database instance and SQL statements for $T1$ and $T2$ such that result of running $T2$ with the first SQL isolation level is different from running $T2$ with the second SQL isolation level. Also specify the common schedule of $T1$ and $T2$ and explain why the results are different.

1. `SERIALIZABLE` versus `REPEATABLE READ`.

2. `REPEATABLE READ` versus `READ COMMITTED`.

3. `READ COMMITTED` versus `READ UNCOMMITTED`.

Answer 16.9 The answer to each question is given below.

1. Suppose a database instance of table Catalog and SQL statements shown below:

sid	pid	cost
18	45	$7.05
22	98	$89.35
31	52	$357.65
31	53	$26.22
58	15	$37.50
58	94	$26.22

```
SELECT *
FROM   Catalog C
WHERE  C.cost < 100
EXCEPT
(SELECT *
FROM   Catalog C
WHERE  C.cost < 100 )

INSERT INTO catalog (sid, pid, cost)
VALUES (99, 38, 75.25)
```

When we use SERIALIZABLE, we would expect that the first SQL statement return nothing. But if we instead use REPEATABLE READ, then it is possible that the phantom problem could occur from inserting a new tuple into the table, resulting in the first SQL statement incorrectly returning a tuple (in this case the one inserted by the second SQL statement).

2. Suppose the same database instance as above and SQL statements shown below:

```
UPDATE Catalog
SET    cost = cost * 0.95
WHERE  sid = 31

UPDATE Catalog
SET    cost = cost * 0.95
WHERE  sid = 31
```

When we use READ COMMITTED on the SQL statements above, an unrepeatable read could occur resulting in an incorrect value being assigned to cost. But this problem cannot occur when we use REPEATABLE READ.

3. Suppose the same database instance as above and SQL statements shown below (assuming READ UNCOMMITTED can write to the database):

```
UPDATE Catalog
SET    cost = cost * 0.95

SELECT C.sid, C.pid
FROM   Catalog C
WHERE  C.cost = 36.22
```

When we use READ UNCOMMITTED on the SQL statements above, dirty read of the value of cost could occur because the first SQL statement might not be finished while the second one is reading. But this problem cannot occur when we use READ UNCOMMITTED.

17

CONCURRENCY CONTROL

Exercise 17.1 Answer the following questions:

1. Describe how a typical lock manager is implemented. Why must lock and unlock be atomic operations? What is the difference between a lock and a *latch*? What are *convoys* and how should a lock manager handle them?
2. Compare *lock downgrades* with upgrades. Explain why downgrades violate 2PL but are nonetheless acceptable. Discuss the use of *update* locks in conjunction with lock downgrades.
3. Contrast the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
4. State and justify the Thomas Write Rule.
5. Show that, if two schedules are conflict equivalent, then they are view equivalent.
6. Give an example of a serializable schedule that is not strict.
7. Give an example of a strict schedule that is not serializable.
8. Motivate and describe the use of locks for improved conflict resolution in Optimistic Concurrency Control.

Answer 17.1 The answer to each question is given below.

1. A typical lock manager is implemented with a hash table, also called lock table, with the data object identifier as the key. A lock table entry contains the following information: the number of transactions currently holding a lock on the object, the nature of the lock, and a pointer to a queue of lock requests.

Lock and unlock must be atomic operations because otherwise it may be possible for two transactions to obtain an exclusive lock on the same object, thereby destroying the principles of 2PL.

A lock is held over a long duration, and a latch is released immediately after the physical read or write operation is completed.

Convoy is a queue of waiting transactions. It occurs when a transaction holding a heavily used lock is suspended by the operating system, and every other transactions that needs this lock is queued.

2. A *lock upgrade* is to grant a transaction an exclusive lock of an object for which it already holds a shared lock. A *lock downgrade* happens when an exclusive lock is obtained by a transaction initially, but downgrades to a shared lock once it's clear that this is sufficient.

   Lock downgrade violates the 2PL requirement because it reduces the locking privileges held by a transaction, and the transaction may go on to acquire other locks. But the transaction did nothing but read the object that it downgraded. So it is nonetheless acceptable, provided that the transaction has not modified the object.

   The downgrade approach reduces concurrency, so we introduce a new kind of lock, called an update lock, that is compatible with shared locks but not other updates and exclusive lock. By setting an update lock initially, rather than a exclusive lock as in the case of lock downgrade, we prevent conflicts with other read operations and increase concurrency.

3. When timestamps are used for deadlock prevention, a transaction that is aborted and re-started it is given the same timestamp that it had originally. When timestamps are used for concurrency control, a transaction that is aborted and restarted is given a new, larger timestamp.

4. The Thomas Write Rule says that if an transaction T with timestamp TS(T) acts on a database object O with a write timestamp of WTS(O) such that TS(T) $<$ WTS(O), we can safely ignore writes by T and continue.

   To understand and justify the Thomas Write Rule fully, we need to give the complete context when it arises.

   To implement timestamp-based concurrency control scheme, the following regulations are made when transaction T wants to write object O:

   (a) If $TS(T) < RTS(O)$, the write action conflicts with the most recent read action of O, and T is therefore aborted and restarted.
   (b) If $TS(T) < WTS(O)$, a naive approach would be to abort T as well because its write action conflicts with the most recent write of O, and is out of timestamp order. But it turns out that we can safely ignore such previous write and process with this new write; this is called $Thomas' Write Rule$.

(c) Otherwise, T writes O and WTS(O) is set to TS(T).

The justification is as follows: had $TS(T) < RTS(O)$, T would have been aborted and we would not have bothered to check the WTS(O). So to decide whether to abort T based on WTS(O), we can assume that $TS(T) >= RTS(O)$. If $TS(T) >= RTS(O)$ and $TS(T) < WTS(O)$, then $RTS(O) < WTS(O)$, which means the previous write occurred immediately before this planned-new-write of O and was never read by anyone, therefore the previous write can be safely ignored.

5. If two schedules over the same set of actions of the transactions are conflict equivalent, they must order every pair of conflicting actions of two committed transactions in the same way. Let's assume that two schedules are conflict equivalent, but they are not view equivalent, then one of the three conditions held under view equivalency must be violated. But as we can see if every pair of conflicting actions is ordered in the same way, this cannot happen. Thus we can conclude that if two schedules are conflict equivalent, they are also view equivalent.

6. The following example is a serializable schedule, but it's not strict.
T1:R(X), T2:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit

7. The following example is a strict schedule, but it's not serializable.
T1:R(X), T2:R(X), T1:W(X), T1:Commit, T2:W(X), T2:Commit

8. In Optimistic Concurrency Control, we have no way to tell when Ti wrote the object at the time we validate Tj, since all we have is the list of objects written by Ti and the list read by Tj. To solve such conflict, we use mechanisms very similar to locking. The basic idea is that each transaction in the Read phase tells the DBMS about items it is reading, and when a transaction Ti is committed and its writes are accepted, the DBMS checks whether any of the items written by Ti are being rad by any (yet to be validated) transaction Tj. If so, we know that Tj's validation must eventually fail. Then we can pick either the die or kill policy to resolve the conflict.

Exercise 17.2 Consider the following classes of schedules: *serializable, conflict-serializable, view-serializable, recoverable, avoids-cascading-aborts,* and *strict*. For each of the following schedules, state which of the preceding classes it belongs to. If you cannot decide whether a schedule belongs in a certain class based on the listed actions, explain briefly.

The actions are listed in the order they are scheduled and prefixed with the transaction name. If a commit or abort is not shown, the schedule is incomplete; assume that abort or commit must follow all the listed actions.

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)

2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)
3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)
4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)
5. T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
7. T1:W(X), T2:R(X), T1:W(X), T2:Abort, T1:Commit
8. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. T2: R(X), T3:W(X), T3:Commit, T1:W(Y), T1:Commit, T2:R(Y), T2:W(Z), T2:Commit
11. T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit

Answer 17.2 For simplicity, we assume the listed transactions are the only ones active currently in the database and if a commit or abort is not shown for a transaction, we'll assume a commit will follow all the listed actions.

1. Not serializable, not conflict-serializable, not view-serializable;
   It is recoverable and avoid cascading aborts; not strict.
2. It is serializable, conflict-serializable, and view-serializable;
   It does NOT avoid cascading aborts, is not strict;
   We can not decide whether it's recoverable or not, since the abort/commit sequence of these two transactions are not specified.
3. It is the same with number 2 above.
4. It is NOT serializable, NOT conflict-serializable, NOT view-serializable;
   It is NOT avoid cascading aborts, not strict;
   We can not decide whether it's recoverable or not, since the abort/commit sequence of these transactions are not specified.
5. It is serializable, conflict-serializable, and view-serializable;
   It is recoverable and avoid cascading aborts;
   It is not strict.
6. It is serializable and view-serializable, not conflict-serializable;
   It is recoverable and avoid cascading aborts;
   It is not strict.

7. It is not serializable, not view-serializable, not conflict-serializable;
   It is not recoverable, therefore not avoid cascading aborts, not strict.
8. It is not serializable, not view-serializable, not conflict-serializable;
   It is not recoverable, therefore not avoid cascading aborts, not strict.
9. It is serializable, view-serializable, and conflict-serializable;
   It is not recoverable, therefore not avoid cascading aborts, not strict.
10. It belongs to all above classes.
11. (assume the 2nd T2:Commit is instead T1:Commit).
    It is serializable and view-serializable, not conflict-serializable;
    It is recoverable, avoid cascading aborts and strict.
12. It is serializable and view-serializable, not conflict-serializable;
    It is recoverable, but not avoid cascading aborts, not strict.

Exercise 17.3 Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule, and Multiversion. For each of the schedules in Exercise 17.2, state which of these protocols allows it, that is, allows the actions to occur in exactly the order shown.

For the timestamp-based protocols, assume that the timestamp for transaction Ti is i and that a version of the protocol that ensures recoverability is used. Further, if the Thomas Write Rule is used, show the equivalent serial schedule.

Answer 17.3 See the table 17.1.

Note the following abbreviations.
S-2PL: Strict 2PL; C-2PL: Conservative 2PL; Opt cc: Optimistic; TS W/O THR: Timestamp without Thomas Write Rule; TS With THR: Timestamp without Thomas Write Rule.

Thomas Write Rule is used in the following schedules, and the equivalent serial schedules are shown below:
5. T1:R(X), T1:W(X), T2:Abort, T1:Commit

6. T1:R(X), T1:W(X), T2:Commit, T1:Commit
11. T1:R(X), T2:Commit, T1:W(X), T2:Commit, T3:R(X), T3:Commit

Exercise 17.4 Consider the following sequences of actions, listed in the order they are submitted to the DBMS:

	2PL	S-2PL	C-2PL	Opt CC	TS w/o TWR	TS w/ TWR	Multiv.
1	N	N	N	N	N	N	N
2	Y	N	N	Y	Y	Y	Y
3	N	N	N	Y	N	N	Y
4	N	N	N	Y	N	N	Y
5	N	N	N	Y	N	Y	Y
6	N	N	N	N	N	Y	Y
7	N	N	N	Y	N	N	N
8	N	N	N	N	N	N	N
9	N	N	N	Y	N	N	N
10	N	N	N	N	Y	Y	Y
11	N	N	N	N	N	Y	N
12	N	N	N	N	N	Y	Y

Table 17.1

- **Sequence S1:** T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit
- **Sequence S2:** T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control mechanisms, describe how the concurrency control mechanism handles the sequence.

Assume that the timestamp of transaction Ti is i. For lock-based concurrency control mechanisms, add lock and unlock requests to the previous sequence of actions as per the locking protocol. The DBMS processes actions in the order shown. If a transaction is blocked, assume that all its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

1. Strict 2PL with timestamps used for deadlock prevention.
2. Strict 2PL with deadlock detection. (Show the waits-for graph in case of deadlock.)
3. Conservative (and Strict, i.e., with locks held until end-of-transaction) 2PL.
4. Optimistic concurrency control.
5. Timestamp concurrency control with buffering of reads and writes (to ensure recoverability) and the Thomas Write Rule.
6. Multiversion concurrency control.

Answer 17.4 The answer to each question is given below.

1. Assume we use Wait-Die policy.
   Sequence S1: T1 acquires shared-lock on X;
   When T2 asks for an exclusive lock on X, since T2 has a lower priority, it will be aborted;
   T3 now gets exclusive-lock on Y;
   When T1 also asks for an exclusive-lock on Y which is still held by T3, since T1 has higher priority, T1 will be blocked waiting;
   T3 now finishes write, commits and releases all the lock;
   T1 wakes up, acquires the lock, proceeds and finishes;
   T2 now can be restarted successfully.

   Sequence S2: The sequence and consequence are the same with Sequence S1, except T2 was able to advance a little more before it gets aborted.

2. In deadlock detection, transactions are allowed to wait, they are not aborted until a deadlock has been detected. (Compared to prevention schema, some transactions may have been aborted prematurely.)

   Sequence S1: T1 gets a shared-lock on X;
   T2 blocks waiting for an exclusive-lock on X;
   T3 gets an exclusive-lock on Y;
   T1 blocks waiting for an exclusive-lock on Y;
   T3 finishes, commits and releases locks;
   T1 wakes up, gets an exclusive-lock on Y, finishes up and releases lock on X and Y;
   T2 now gets both an exclusive-lock on X and Y, and proceeds to finish.
   No deadlock.

   Sequence S2: There is a deadlock. T1 waits for T2, while T2 waits for T1.

3. **Sequence S1:** With conservative and strict 2PL, the sequence is easy. T1 acquires lock on both X and Y, commits, releases locks; then T2; then T3.

   Sequence S2: Same as Sequence S1.

4. Optimistic concurrency control:
   For both S1 and S2: each transaction will execute, read values from the database and write to a private workspace; they then acquire a timestamp to enter the validation phase. The timestamp of transaction Ti is i.

   Sequence S1: Since T1 gets the earliest timestamp, it will commit without problem; but when validating T2 against T1, none of the three conditions hold, so T2 will be aborted and restarted later; so is T3 (same as T2).

   Sequence S2: The fate is the same as in Sequence S1.

	Serializable	Conflict-serializable	Recoverable	Avoid cascading aborts
1	No	No	No	No
2	No	No	Yes	Yes
3	Yes	Yes	Yes	Yes
4	Yes	Yes	Yes	Yes

Table 17.2

5. Timestamp concurrency control with buffering of reads and writes and TWR.

   Sequence S1: This sequence will be allowed the way it is.

   Sequence S2: Same as above.

6. Multiversion concurrency control

   Sequence S1: T1 reads X, so RTS(X) = 1;
   T2 is able to write X, since TS(T2) ¿ RTS(X); and RTS(X) and WTS(X) are set to 2;
   T2 writes Y, RTS(Y) and WTS(Y) are set to 2;
   T3 is able to write Y as well, so RTS(Y) and WTS(Y) are set to 3;
   Now when T1 tries to write Y, since TS(T1) ¡ RTS(Y), T1 needs to be aborted and restarted later.

   Sequence S2: The fate is similar to the one in Sequence S1.

Exercise 17.5 For each of the following locking protocols, assuming that every transaction follows that locking protocol, state which of these desirable properties are ensured: serializability, conflict-serializability, recoverability, avoidance of cascading aborts.

1. Always obtain an exclusive lock before writing; hold exclusive locks until end-of-transaction. No shared locks are ever obtained.
2. In addition to (1), obtain a shared lock before reading; shared locks can be released at any time.
3. As in (2), and in addition, locking is two-phase.
4. As in (2), and in addition, all locks held until end-of-transaction.

Answer 17.5 See the table 17.2.

Exercise 17.6 The Venn diagram (from [76]) in Figure 17.1 shows the inclusions between several classes of schedules. Give one example schedule for each of the regions $S1$ through $S12$ in the diagram.

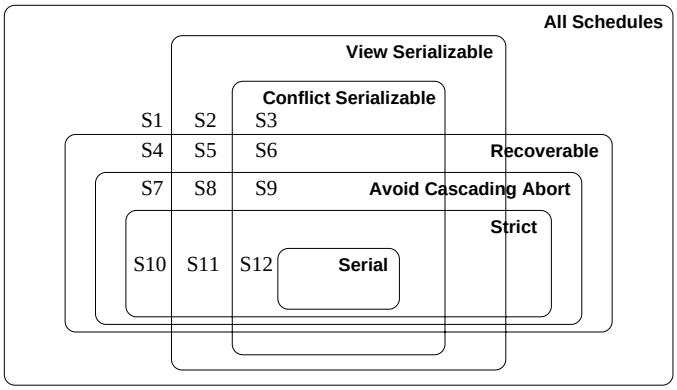


Figure 17.1 Venn Diagram for Classes of Schedules

Answer 17.6 Each section is described below.

- S1
  T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
- S2
  T1:R(X), T2:W(X), T1:W(X), T3:R(X), T3:Commit, T1:Commit, T2:Commit
- S3
  T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
- S4
  T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y), T3:Commit, T2:Commit, T1:Commit
- S5
  T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit
- S6
  T1:W(X), T2:R(Y), T1:R(Y), T2:R(X), T1:Commit, T2:Commit
- S7
  T1:R(X), T2:R(X), T1:W(X), T2:W(X), T1:Commit, T2:Commit
- S8
  T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
- S9
  T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
- S10
  T1:R(X), T2:R(X), T1:W(X), T1:Commit, T2:W(X), T2:Commit

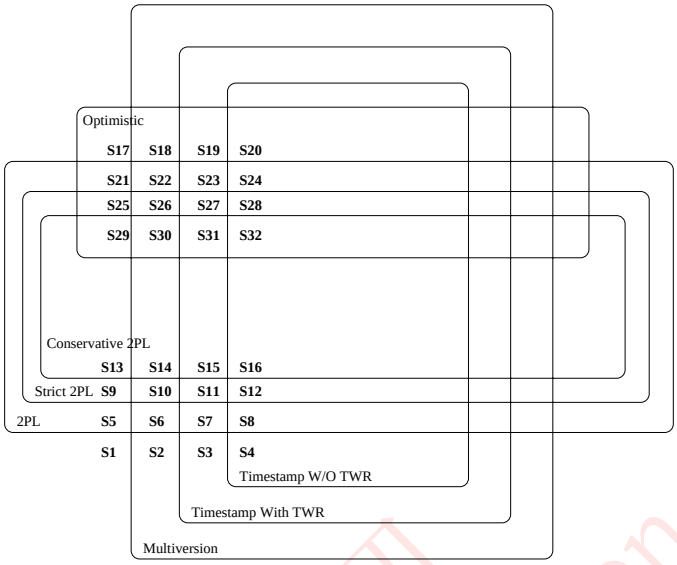


Figure 17.2

- S11
  T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
- S12
  T1:R(X), T2:R(X), T1:Commit, T2:W(X), T2:Commit

Exercise 17.7 Briefly answer the following questions:

1. Draw a Venn diagram that shows the inclusions between the classes of schedules permitted by the following concurrency control protocols: *2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule,* and *Multiversion.*
2. Give one example schedule for each region in the diagram.
3. Extend the Venn diagram to include serializable and conflict-serializable schedules.

Answer 17.7 The answer to each question is given below.

1. See figure 17.2.
2. (a) Here we define the following schedule first:
   i. C1: T0:R(O),T0:Commit.
   ii. C2: T1:Begin,T2:Begin,T1:W(A),T1:Commit,T2:R(A),T2:Commit.

iii. C3: T4:Begin,T3:Begin,T3:W(B),T3:Commit,T4:W(B),T4:Abort.
iv. C4: T4:Begin,T3:Begin,T3:W(B),T3:Commit,T4:R(B),T4:Abort.
v. C5: T3:Begin,T4:Begin,T4:R(B),T4:Commit,T3:W(B),T3:Commit.
vi. C6: T5:Begin,T6:Begin,T6:R(D),T5:R(C),T5:Commit, T6:W(C),T6:Commit.
vii. C7: T5:Begin,T6:Begin,T6:R(D),T5:R(C),T6:W(C), T5:Commit,T6:Commit.
viii. C8: T5:Begin,T6:Begin,T5:R(C),T6:W(C),T5:R(D), T5:Commit,T6:Commit.

Then we have the following schedule for each region in the diagram.(Please note, S1: C2,C5,C8 means that S1 is the combination of schedule C2,C5,C8.)

i. S1: C2,C5,C8
ii. S2: C2,C4,C8
iii. S3: C2,C3,C8
iv. S4: C2,C8
v. S5: C2,C5,C7
vi. S6: C2,C4,C7
vii. S7: C2,C3,C7
viii. S8: C2,C7
ix. S9: C2,C5,C6
x. S10: C2,C4,C6
xi. S11: C2,C3,C6
xii. S12: C2,C6
xiii. S13: C2,C5
xiv. S14: C2,C4
xv. S15: C2,C3
xvi. S16: C2,C1

And for the rest of 16 schedules, just remove the C2 from the corresponding schedule.(eg, S17: C5,C8, which is made by removing C2 from S1.)

3. See figure 17.3.

Exercise 17.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`, *did:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *floor:* `integer`)

and on the following update command:

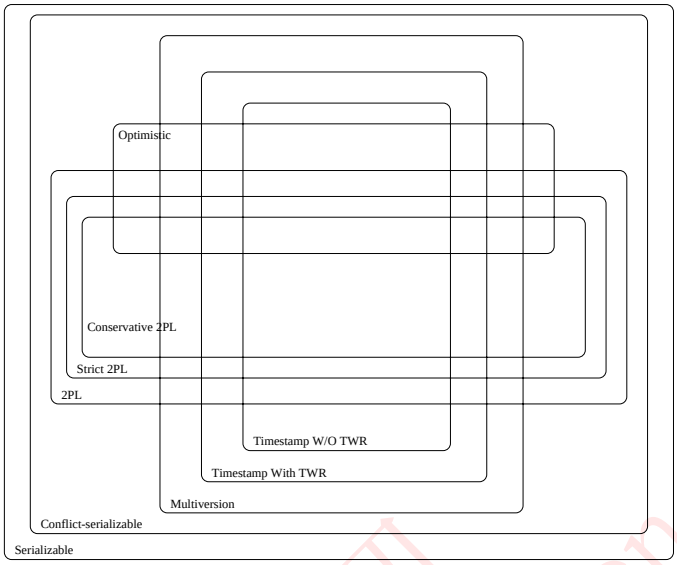


Figure 17.3

replace (salary = 1.1 * EMP.salary) where EMP.ename = 'Santa'

1. Give an example of a query that would conflict with this command (in a concurrency control sense) if both were run at the same time. Explain what could go wrong, and how locking tuples would solve the problem.
2. Give an example of a query or a command that would conflict with this command, such that the conflict could not be resolved by just locking individual tuples or pages but requires index locking.
3. Explain what index locking is and how it resolves the preceding conflict.

Answer 17.8 The answer to each question is given below.

1. One example that would conflict with the above query is:

   Select eid From Emp Where salary = 10,000

   Suppose there are two employees who both have salary of 10,000. If the update command is carried out first, neither of them will be selected; if the select command is done first, both of them should have been selected. However, if these two were run at the same time, one tuple might be updated first causing one to be selected while the other is not. By locking tuples, it is ensured that either all the tuples are updated first or all the tuples go through the selection query first.

2. One example that would conflict with the above query is:

   Insert (EID, "Santa", AGE, SALARY, DID) Into Emp

3. If there is an index on a particular field of a relation, a transaction can obtain a lock on one or more index page(s). This will effectively lock all the existing records with the field having a certain range of values, it will also prevent insertion of new records with the field value in that particular range. This will prevent the so-called phantom problem. This technique is called **index locking.**

   It is clear to see that index locking could solve the problem mentioned in 2.

Exercise 17.9 SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. For each of the eight classes, describe a locking protocol that allows only transactions in this class. Does the locking protocol for a given class make any assumptions about the locking protocols used for other classes? Explain briefly.
2. Consider a schedule generated by the execution of several SQL transactions. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
3. Consider a schedule generated by the execution of several SQL transactions, each of which has `READ ONLY` access-mode. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
4. Consider a schedule generated by the execution of several SQL transactions, each of which has `SERIALIZABLE` isolation-level. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
5. Can you think of a timestamp-based concurrency control scheme that can support the eight classes of SQL transactions?

Answer 17.9 1. The classes `SERIALIZABLE`, `REPEATABLE READ` and `READ COMMITTED` rely on the assumption that other classes obtain exclusive locks before writing objects and hold exclusive locks until the end of the transaction.

   (a) `SERIALIZABLE` + `READ ONLY`: Strict 2PL including locks on a set of objects that it requires to be unchanged. No exclusive locks are granted.

   (b) `SERIALIZABLE` + `READ WRITE`: Strict 2PL including locks on a set of objects that it requires to be unchanged.

   (c) `REPEATABLE READ` + `READ ONLY`: Strict 2PL, only locks individual objects, not sets of objects. No exclusive locks are granted.

   (d) `REPEATABLE READ` + `READ WRITE`: Strict 2PL, only locks individual objects, not sets of objects.

(e) `READ COMMITTED` + `READ ONLY`: Obtains shared locks before reading objects, but these locks are released immediately.

(f) `READ COMMITTED` + `READ WRITE`: Obtains exclusive locks before writing objects, and hold these locks until the end. Obtains shared locks before reading objects, but these locks are released immediately.

(g) `READ UNCOMMITTED` + `READ ONLY`: Do not obtain shared locks before reading objects.

(h) `READ UNCOMMITTED` + `READ WRITE`: Obtains exclusive locks before writing objects, and hold these locks until the end. Does not obtain shared locks before reading objects.

2. Suppose we do not have any requirements for the access-mode and isolation-level of the transaction, then they are not guaranteed to be conflict-serializable, serializable, or recoverable.

3. A schedule generated by the execution of several SQL transactions, each of which having `READ ONLY` access-mode, would be guaranteed to be conflict-serializable, serializable, and recoverable. This is because the only actions are reads so there are no WW, RW, or WR conflicts.

4. A schedule generated by the execution of several SQL transactions, each of which having `SERIALIZABLE` isolation-level, would be guaranteed to be conflict-serializable, serializable, and recoverable. This is because `SERIALIZABLE` isolation level follows strict 2PL.

5. Timestamp locking with wait-die or would wait would be suitable for any `SERIALIZABLE` or `REPEATABLE READ` transaction because these follow strict 2PL. This could be modified to allow `READ COMMITTED` by allowing other transactions with a higher priority to read values changed by this transaction, as long as they didn't need to overwrite the changes. `READ UNCOMMITTED` transactions can only be in the `READ ONLY` access mode, so they can read from any timestamp.

Exercise 17.10 Consider the tree shown in Figure 19.5. Describe the steps involved in executing each of the following operations according to the tree-index concurrency control algorithm discussed in Section 19.3.2, in terms of the order in which nodes are locked, unlocked, read, and written. Be specific about the kind of lock obtained and answer each part independently of the others, always starting with the tree shown in Figure 19.5.

1. Search for data entry 40*.

2. Search for all data entries $k*$ with $k \leq 40$.

3. Insert data entry 62*.

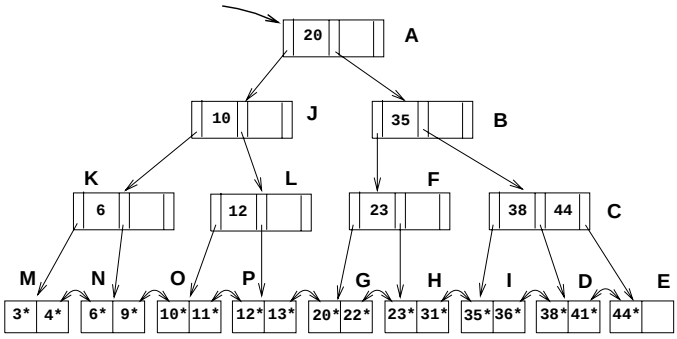


Figure 17.4

4. Insert data entry 40*.

5. Insert data entries 62* and 75*.

Answer 17.10 Please refer to Figure 17.4, and note the abbreviation used:
S(A): Obtains shared lock on A.
X(A): Obtains exclusive lock on A.
RS(A): Release shared lock on A.
RX(A): Release exclusive lock on A.
R(A): Read node A.
W(A): Write node A.

1. S(A),R(A),S(B),RS(A),R(B),S(C),RS(B),R(C),S(D),RS(C),R(D),...(obtain other locks needed for further operation), then release lock on D when the transaction is going to commit(Strict 2PL).

2. S(A), R(A), S(J), S(B), R(J), R(B), RS(A), S(K), S(L), RS(J), S(F), S(C), RS(B), R(K), R(L),
R(F), R(C), S(M), S(N), S(O), S(P), S(G), S(H), S(I), S(D), R(M), R(N), R(O), R(P), R(G),
R(H), R(I), R(D), ..., then release locks on M, N, O, P, G, H, I, D when the transaction is going to commit.

3. X(A), R(A), X(B), R(B), RX(A), X(C), R(C), X(E), R(E), W(E), ..., then release lock on D when the transaction is going to commit.

4. X(A), R(A), X(B), R(B), RX(A), X(F), R(F), RX(B), X(H), R(H), New Node Z, X(Z), X(G), R(G), W(G), W(Z), W(H), W(F), RX(F), ..., then release locks on G, Z, H when the transaction is going to commit.

5. X(A), R(A), X(B), R(B), RX(A), X(C), R(C), X(E), R(E), New Node Z, X(Z), W(E), W(Z), New Node Y, X(Y), W(C), W(Y), W(B), RX(B), RX(C), RX(Y), ..., then release locks on E, Z when the transaction is going to commit.

Exercise 17.11 Consider a database organized in terms of the following hierarchy of objects: The database itself is an object (D), and it contains two files ($F1$ and $F2$), each of which contains 1000 pages ($P1 \ldots P1000$ and $P1001 \ldots P2000$, respectively). Each page contains 100 records, and records are identified as $p : i$, where p is the page identifier and i is the slot of the record on that page.

Multiple-granularity locking is used, with S, X, IS, IX and SIX locks, and database-level, file-level, page-level and record-level locking. For each of the following operations, indicate the sequence of lock requests that must be generated by a transaction that wants to carry out (just) these operations:

1. Read record $P1200 : 5$.
2. Read records $P1200 : 98$ through $P1205 : 2$.
3. Read all (records on all) pages in file $F1$.
4. Read pages $P500$ through $P520$.
5. Read pages $P10$ through $P980$.
6. Read all pages in $F1$ and (based on the values read) modify 10 pages.
7. Delete record $P1200 : 98$. (This is a blind write.)
8. Delete the first record from each page. (Again, these are blind writes.)
9. Delete all records.

Answer 17.11 The answer to each question is given below.

1. IS on D; IS on F2; IS on P1200; S on P1200:5.
2. IS on D; IS on F2; IS on P1200, S on 1201 through 1204, IS on P1205; S on P1200:98/99/100, S on P1205:1/2.
3. IS on D; S on F1
4. IS on D; IS on F1; S on P500 through P520.
5. IS on D; S on F1 (performance hit of locking 970 pages is likely to be higher than other blocked transactions).

6. IS and IX on D; SIX on F1.

7. IX on D; IX on F2; X on P1200.
(Locking the whole page is not necessary, but it would require some reorganization or compaction.)

8. IX on D; X on F1 and F2.
(There are many ways to do this, there is a tradeoff between overhead and concurrency.)

9. IX on D; X on F1 and F2.

Exercise 17.12 Suppose that we have only two types of transactions, $T1$ and $T2$. Transactions preserve database consistency when run individually. We have defined several *integrity constraints* such that the DBMS never executes any SQL statement that brings the database into an inconsistent state. Assume that the DBMS does not perform *any* concurrency control. Give an example schedule of two transactions $T1$ and $T2$ that satisfies all these conditions, yet produces a database instance that is not the result of any serial execution of $T1$ and $T2$.

Answer 17.12 We cannot guarantee the correctness of the database just by defining integrity constraints. Consider the case where transaction $T1$ increments all employee's salaries by \$500 and transaction $T2$ decrements all employee's salaries by \$500. If there's no concurrency control when executing these two transactions together, it could possibly overwrite some uncommitted data and increment some of employess's salaries by \$500, decrement some by \$500, and leave no changes on the rest. Conflicts caused by such as interleaving transactions cannot be simply resolved by defining integrity constraints on the database. Thus we need concurrency control.

18

CRASH RECOVERY

Exercise 18.1 Briefly answer the following questions:

1. How does the recovery manager ensure atomicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a system crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

Answer 18.1 The answer to each question is given below.

1. The Recovery Manager ensures atomicity of transactions by undoing the actions of transactions that do not commit. It ensures durability by making sure that all actions of committed transactions survive system crashes and media failures.
2. Stable storage is guaranteed (with very high probability) to survive crashes and media failures. A disk might get corrupted or fail but the stable storage is still expected to retain whatever is stored in it. One of the ways of achieving stable storage is to store the information in a set of disks rather than in a single disk with some information duplicated so that the information is available even if one or two of the disks fail.
3. A system crash happens when the system stops functioning in a normal way or stops altogether. The Recovery Manager and other parts of the DBMS stop functioning (e.g. a core dump caused by a bus error) as opposed to media failure.

   In a media failure, the system is up and running but a particular entity of the system is not functioning. In this case, the Recovery Manager is still functioning and can start recovering from the failure while the system is still running (e.g., a disk is corrupted).

4. WAL Protocol: Whenever a change is made to a database object, the change is first recorded in the log and the log is written to stable storage before the change is written to disk.

5. If a steal policy is in effect, the changes made to an object in the buffer pool by a transaction can be written to disk before the transaction commits. This might be because some other transaction might "steal" the buffer page presently occupied by an uncommitted transaction.

   A no-force policy is in effect if, when a transaction commits, we need not ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk.

Exercise 18.2 Briefly answer the following questions:

1. What are the properties required of LSNs?
2. What are the fields in an update log record? Explain the use of each field.
3. What are redoable log records?
4. What are the differences between update log records and CLRs?

Answer 18.2 The answer to each question is given below.

1. As with any record id, it should be possible to fetch a log record with one disk access given the log sequence numbers, LSNs. Further, LSNs should be assigned in monotonically increasing order; this property is required by the ARIES recovery algorithm.

2. An update log record consists of two sets of fields: a) Fields common to all log records – prevLSN, transID and type. b) Fields unique for update log records – pageID, length, offset, before-image and after-image.

   - prevLSN - the previous LSN of a given transaction.
   - transID - the id of the transaction generating the log record.
   - type - indicates the type of the log record.
   - pageID - the pageID of the modified page.
   - length - length in bytes of the change.
   - offset - offset into the page of the change.
   - before-image - value of the changed bytes before the change.
   - after-image - value of the changed bytes after the change.

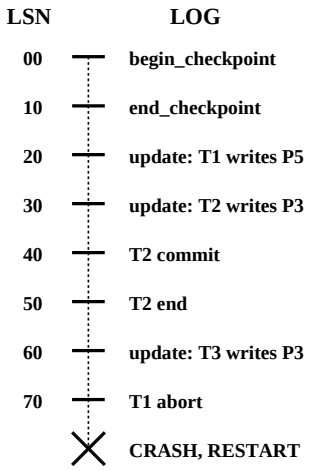


Figure 18.1 Execution with a Crash

3. Redoable log records are update log records and compensation log records; executing the actions indicated by these records several times is equivalent to executing them once.

4. A compensation log record (CLR) C describes the action taken to undo the actions recorded in the corresponding update log record U. (This can happen during normal system execution when a transaction is aborted, or during recovery from a crash.) The compensation log record C also contains a field called undonextLSN which is the LSN of the next log record that is to be undone for the transaction that wrote update record U; this field in C is set to the value of prevLSN in U.

   Unlike an update log record, a CLR describes an action that will never be undone. An aborted transaction will never be revived, therefore once a CLR has properly returned the data its previous state, both transactions can be forgotten.

Exercise 18.3 Briefly answer the following questions:

1. What are the roles of the Analysis, Redo, and Undo phases in ARIES?

2. Consider the execution shown in Figure 18.1.

   (a) What is done during Analysis? (Be precise about the points at which Analysis begins and ends and describe the contents of any tables constructed in this phase.)

   (b) What is done during Redo? (Be precise about the points at which Redo begins and ends.)

   (c) What is done during Undo? (Be precise about the points at which Undo begins and ends.)

Answer 18.3 The answer to each question is given below.

1. The Analysis phase starts with the most recent begin_checkpoint record and proceeds forward in the log until the last log record. It determines

   (a) The point in the log at which to start the Redo pass

   (b) The dirty pages in the buffer pool at the time of the crash.

   (c) Transactions that were active at the time of the crash which need to be undone.

   The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash. The Undo phase follows Redo and undoes the changes of all transactions that were active at the time of the crash.

2. (a) For this example, we will assume that the Dirty Page Table and Transaction Table were empty before the start of the log. Analysis determines that the last begin_checkpoint was at LSN 00 and starts at the corresponding end_checkpoint (LSN 10).

   We will denote Transaction Table records as (transID, lastLSN) and Dirty Page Table records as (pageID, recLSN) sets.

   Then Analysis phase runs until LSN 70, and does the following:

LSN 20	Adds (T1, 20) to TT and (P5, 20) to DPT
LSN 30	Adds (T2, 30) to TT and (P3, 30) to DPT
LSN 40	Changes status of T2 to "C" from "U"
LSN 50	Deletes entry for T2 from Transaction Table
LSN 60	Adds (T3, 60) to TT. Does not change P3 entry in DPT
LSN 70	Changes (T1, 20) to (T1, 70)

   The final Transaction Table has two entries: (T1, 70), and (T3, 60). The final Dirty Page Table has two entries: (P5, 20), and (P3, 30).

   (b) Redo Phase: Redo starts at LSN 20 (smallest recLSN in DPT).

LSN 20	Changes to P5 are redone.
LSN 30	P3 is retrieved and its pageLSN is checked. If the page had been written to disk before the crash (i.e. if $pageLSN >= 30$), nothing is re-done otherwise the changes are re-done.
LSN 40,50	No action
LSN 60	Changes to P3 are redone
LSN 70	No action

   (c) Undo Phase: Undo starts at LSN 70 (highest lastLSN in TT). The Loser Set consists of LSNs 70 and 60. LSN 70: Adds LSN 20 to the Loser Set. Loser Set = (60, 20). LSN 60: Undoes the change on P3 and adds a CLR indicating this

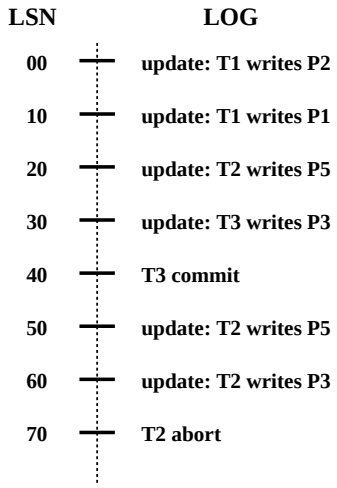


Figure 18.2 Aborting a Transaction

Undo. Loser Set = (20). LSN 20: Undoes the change on P5 and adds a CLR indicating this Undo.

Exercise 18.4 Consider the execution shown in Figure 18.2.

1. Extend the figure to show prevLSN and undonextLSN values.
2. Describe the actions taken to rollback transaction $T2$.
3. Show the log after $T2$ is rolled back, including all prevLSN and undonextLSN values in log records.

Answer 18.4 The answer to each question is given below.

1. The extended figure is shown below:

LSN	prevLSN	undonextLSN(of a CLR corresponds to the ULR)
00	–	–
10	00	00
20	–	–
30	–	–
40	30	– (not an update log record)
50	20	20
60	50	50
70	60	– (not an update log record)

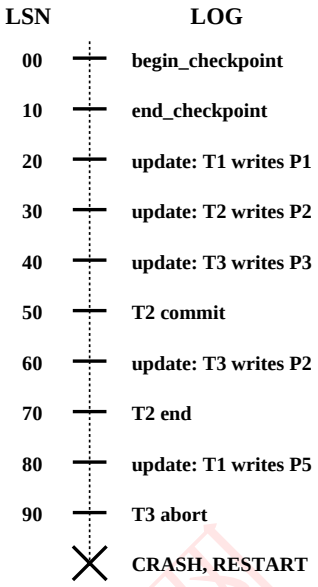


Figure 18.3 Execution with Multiple Crashes

2. Step i) Restore P3 to the before-image stored in LSN 60.
   Step ii) Restore P5 to the before-image stored in LSN 50.
   Step iii) Restore P5 to the before-image stored in LSN 20.

3. The log tail should look something like this:

LSN	prevLSN	transID	type	pageID	undonextLSN
80	70	T2	CLR	P3	50
90	80	T2	CLR	P5	20
100	90	T2	CLR	P5	–
110	100	T2	END	–	–

Exercise 18.5 Consider the execution shown in Figure 18.3. In addition, the system crashes during recovery after writing two log records to stable storage and again after writing another two log records.

1. What is the value of the LSN stored in the master log record?
2. What is done during Analysis?
3. What is done during Redo?
4. What is done during Undo?

5. Show the log when recovery is complete, including all non-null prevLSN and undonextLSN values in log records.

Answer 18.5 The answer to each question is given below.

1. LSN 00 is stored in the master log record as it is the LSN of the begin_checkpoint record.

2. During analysis the following happens:

LSN 20	Add (T1,20) to TT and (P1,20) to DPT
LSN 30	Add (T2,30) to TT and (P2,30) to DPT
LSN 40	Add (T3,40) to TT and (P3,40) to DPT
LSN 50	Change status of T2 to C
LSN 60	Change (T3,40) to (T3,60)
LSN 70	Remove T2 from TT
LSN 80	Change (T1,20) to (T1,70) and add (P5,70) to DPT
LSN 90	No action

   At the end of analysis, the transaction table contains the following entries: (T1,80), and (T3,60). The Dirty Page Table has the following entries: (P1,20), (P2,30), (P3,40), and (P5,80).

3. Redo starts from LSN20 (minimum recLSN in DPT).

LSN 20	Check whether P1 has pageLSN more than 10 or not. Since it is a committed transaction, we probably need not redo this update.
LSN 30	Redo the change in P2
LSN 40	Redo the change in P3
LSN 50	No action
LSN 60	Redo the changes on P2
LSN 70	No action
LSN 80	Redo the changes on P5
LSN 90	No action

4. ToUndo consists of (80, 60).

LSN 80	Undo the changes in P5. Append a CLR: Undo T1 LSN 80, set undonextLSN = 20. Add 20 to ToUndo.

   ToUndo consists of (60, 20).

LSN 60	Undo the changes on P2. Append a CLR: Undo T3 LSN 60, set undonextLSN = 40. Add 40 to ToUndo.

ToUndo consists of (40, 20).

LSN 40	Undo the changes on P3. Append a CLR: Undo T3 LSN 40, T3 end

ToUndo consists of (20).

LSN 20	Undo the changes on P1. Append a CLR: Undo T1 LSN 20, T1 end

5. The log looks like the following after recovery:

LSN 00	begin_checkpoint	
LSN 10	end_checkpoint	
LSN 20	update: T1 writes P1	
LSN 30	update: T2 writes P2	
LSN 40	update: T3 writes P3	
LSN 50	T2 commit	prevLSN = 30
LSN 60	update: T3 writes P2	prevLSN = 40
LSN 70	T2 end	prevLSN = 50
LSN 80	update: T1 writes P5	prevLSN = 20
LSN 90	T3 abort	prevLSN = 60
LSN 100	CLR: Undo T1 LSN 80	undonextLSN= 20
LSN 110	CLR: Undo T3 LSN 60	undonextLSN= 40
LSN 120,125	CLR: Undo T3 LSN 40	T3 end.
LSN 130,135	CLR: Undo T1 LSN 20	T1 end.

Exercise 18.6 Briefly answer the following questions:

1. How is checkpointing done in ARIES?
2. Checkpointing can also be done as follows: Quiesce the system so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and transaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointing approach of ARIES?
3. What happens if a second begin_checkpoint record is encountered during the Analysis phase?
4. Can a second end_checkpoint record be encountered during the Analysis phase?
5. Why is the use of CLRs important for the use of undo actions that are not the physical inverse of the original update?
6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

Answer 18.6 The answer to each question is given below.

1. Checkpointing in ARIES consists of three steps. First, to indicate the instant at which the checkpoint starts, a begin_checkpoint record is written. Second, an end_checkpoint record is constructed, which includes the current contents of the transaction table and the dirty page table, and is appended to the log. Note that in ARIES, dirty pages in the buffer pool are not written out. Third, when the end_checkpoint record is written to stable storage, a special master record containing the LSN of the begin_checkpoint log record is written to a known place on stable storage.

2. Checkpointing in ARIES is inexpensive because it does not require quiescing the system or writing out pages in the buffer pool.

   But there are two catches here:
   i) The effectiveness of this checkpointing technique is limited by the earliest recLSN of pages in the dirty pages table, since during restart we must redo changes starting from the log record whose LSN is equal to this recLSN. However, having a background process that periodically writes dirty pages to disk helps to limit this problem.

   ii) If there are additional log records between the begin_checkpoint and end_checkpoint records, the dirty page table and transaction table must be adjusted to reflect the information in these records in the Analysis phase.

3. If a second begin_checkpoint record is encountered during the Analysis phase, it is ignored since without its corresponding end_checkpoint record, no information is provided by this begin_checkpoint record.

4. Yes, this could happen if the system crashes after this end_checkpoint record is written but before the master log record is updated to reflect it.

5. It may happen that a CLR is written to stable storage (following WAL, of course) but that the undo action that it describes is not yet written to disk when the system crashes again. In this case, the undo action described in the CLR is lost and thus needs to be re-applied. The information needed is stored in the CLRs.

6. Consider a transaction T that inserts a data entry 15* into a B+ tree index. Between the time this insert is done and the time that T is eventually aborted, other transactions may also insert and delete entries from the tree. If record-level locks are set, rather than page-level locks, it is possible that the entry 15* is on a different physical level page when T aborts from the one that T inserted it into. In this case, the undo operation for the insert of 15* must be recorded in logical terms, since the physical (byte-level) actions involved in undoing this operation are not the inverse of the physical actions involved in inserting the entry. This results in considerably higher concurrency.

Exercise 18.7 Briefly answer the following questions:

1. If the system fails repeatedly during recovery, what is the maximum number of log records that can be written (as a function of the number of update and other log records written before the crash) before restart completes successfully?
2. What is the oldest log record we need to retain?
3. If a bounded amount of stable storage is used for the log, how can we always ensure enough stable storage to hold all log records written during restart?

Answer 18.7 The answer to each question is given below.

1. Let us take the case where each log record is an update record of an uncommitted transaction and each record belongs to a different transaction. This means there are n records of n different transactions, each of which has to be undone. During recovery, we have to add a CLR record of the undone action and an end_transaction record after each CLR. Thus, we can write a maximum of $2n$ log records before restart completes.
2. The oldest begin_checkpoint referenced in any fuzzy dump or master log record.
3. One needs to ensure that there is enough to hold twice as many records as the current number of log records. If necessary, do a fuzzy dump to free up some log records whenever the number of log records goes above one third of the available space.

Exercise 18.8 Consider the three conditions under which a redo is unnecessary (Section 18.6.2).

1. Why is it cheaper to test the first two conditions?
2. Describe an execution that illustrates the use of the first condition.
3. Describe an execution that illustrates the use of the second condition.

Answer 18.8 The answer to each question is given below.

1. The first two conditions allow us to recognize that a redo is unnecessary without fetching the page.
2. The first condition means all changes to that page have been written to disk. Let say P100 is modified by T2000, then the buffer frame holding P100 is chosen for replacement so that P100 is flushed onto disk. The entry P100 is then removed from the dirty page table. After that, the system crashed. This is the scenario for condition 1.

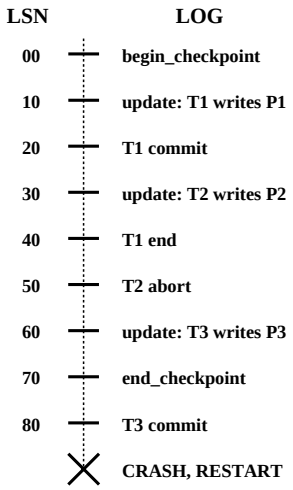


Figure 18.4 Log Records between Checkpoint Records

3. The second condition means that the update being checked was indeed propagated to disk, since the recLSN of a page in the dirty page table is the first update to that page that may not have been written to disk. Let's say P100 is modified by T2000 with recLSN=10, then the buffer frame holding P100 is chosen for replacement so that P100 is flushed onto disk. After that, P100 is modified by T2000 again with recLSN=30. Now in the dirty page table, there is an entry with pageID P100 and recLSN 30. Now, if there is a system crash, in the redo phase while the log record with LSN 10 is being examined, the update does not need to be redone because in the dirty page table, the recLSN for the entry with pageID P100 is greater than the log record being checked.

Exercise 18.9 The description in Section 18.6.1 of the Analysis phase made the simplifying assumption that no log records appeared between the begin_checkpoint and end_checkpoint records for the most recent complete checkpoint. The following questions explore how such records should be handled.

1. Explain why log records could be written between the begin_checkpoint and end_checkpoint records.
2. Describe how the Analysis phase could be modified to handle such records.
3. Consider the execution shown in Figure 18.4. Show the contents of the end_checkpoint record.
4. Illustrate your modified Analysis phase on the execution shown in Figure 18.4.

Answer 18.9 The answer to each question is given below.

1. In ARIES, first a begin_checkpoint record is written and then, after some time, an end_checkpoint record is written. While the end_checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records. So, we could have log records between the begin_checkpoint and the end_checkpoint records. The only guarantee we have is that the transaction table and the dirty page table are accurate as of the time of the begin_checkpoint record.

2. The Analysis phase begins by examining the most recent begin_checkpoint log record and then searches for the next end_checkpoint record. Then the Dirty Page Table and the Transaction Table are initialized to the copies of those structures in the end_checkpoint. Our new Analysis phase remains the same untill here. In the old algorithm, Analysis scans the log in the forward direction until it reaches the end of the log. In the modified algorithm, Analysis goes back to the begin_checkpoint and scans the log in the forward direction.

3. The end_checkpoint record contains the transaction table and the dirty page table as of the time of the begin_checkpoint (LSN 00 in this case). Since we are assuming these tables to be empty before LSN 00, the end_checkpoint record will indicate an empty transaction table and an empty dirty page table.

4. Instead of starting from LSN 80, Analysis goes back to LSN 10 and executes as follows:

LSN 10	Add (T1,10) to TT and (P1, 10) to DPT
LSN 20	Change status of T1 from U to C.
LSN 30	Add (T2,30) to TT and (P2, 30) to DPT
LSN 40	Remove (T1,10) from TT
LSN 50	No action
LSN 60	Add (T3,60) to TT and (P3, 60) to DPT
LSN 70	No action
LSN 80	Change status of T3 from U to C.

Exercise 18.10 Answer the following questions briefly:

1. Explain how media recovery is handled in ARIES.

2. What are the pros and cons of using fuzzy dumps for media recovery?

3. What are the similarities and differences between checkpoints and fuzzy dumps?

4. Contrast ARIES with other WAL-based recovery schemes.

5. Contrast ARIES with shadow-page-based recovery.

Answer 18.10 The answer to each question is given below.

1. Media recovery relies upon periodically making a copy of the database. In ARIES, when some database entity such as a file or a page is corrupted, the copy of that entity is brought up-to date by using the log to identify and reapply the changes of committed transactions and undo the changes of uncommitted transactions (as of the time of the media recovery operation).

2. Pros: Copying a large entity such as a file can take a long time, using fuzzy dumps for media recovery would allow the DBMS to continue with its operations concurrently.

   Cons: When we want to bring a database entity up-to-date, we need to compare the smallest recLSN of a dirty page in the corresponding end-checkpoint record with the LSN of the begin-checkpoint record and call the smaller of these two LSNs I: all log records with LSNs greater than I must be re-applied to the copy and this may be a big overhead.

3. Both checkpoints and fuzzy dumps are done periodically. Taking checkpoint is like taking a snapshot the DBMS state while fuzzy dumps tries to capture a snapshot of a database entity such as a file.

4. A major distinction between ARIES and other WAL based recovery schemes is the Redo phase in ARIES 'repeats history', i.e., re-does the actions of all transactions, not just the non-losers. Other algorithms redo only the non-losers, and the Redo phase follows the Undo phase, in which the actions of losers are rolled back. Due to this repeating history paradigm, and the use of CLRs, ARIES is able to support fine-granularity locks (record-level locks) and logging of logical operations, rather than just byte-level modifications.

5. In shadow-page based recovery, there is no logging and no WAL protocol. When a transaction makes changes to a data page, it actually makes a copy of the page, called shadow of the page and changes the shadow page. Aborting a transaction means just discarding its shadow versions of the page table and the data pages and committing a transaction means making its version of the page table public and discarding the original data pages that are superseded by shadow pages.

19

SCHEMA REFINEMENT AND NORMAL FORMS

Exercise 19.1 Briefly answer the following questions:

1. Define the term *functional dependency*.
2. Why are some functional dependencies called *trivial*?
3. Give a set of FDs for the relation schema *R(A,B,C,D)* with primary key AB under which R is in 1NF but not in 2NF.
4. Give a set of FDs for the relation schema *R(A,B,C,D)* with primary key AB under which R is in 2NF but not in 3NF.
5. Consider the relation schema *R(A,B,C)*, which has the FD $B \rightarrow C$. If A is a candidate key for R, is it possible for R to be in BCNF? If so, under what conditions? If not, explain why not.
6. Suppose we have a relation schema *R(A,B,C)* representing a relationship between two entity sets with keys A and B, respectively, and suppose that R has (among others) the FDs $A \rightarrow B$ and $B \rightarrow A$. Explain what such a pair of dependencies means (i.e., what they imply about the relationship that the relation models).

Answer 19.1

1. Let R be a relational schema and let X and Y be two subsets of the set of all attributes of R. We say Y is functionally dependent on X, written X $\rightarrow$ Y, if the Y-values are determined by the X-values. More precisely, for any two tuples r_1 and r_2 in (any instance of) R

$$\pi_X(r_1) = \pi_X(r_2) \quad \Rightarrow \quad \pi_Y(r_1) = \pi_Y(r_2)$$

2. Some functional dependencies are considered trivial because they contain superfluous attributes that do not need to be listed. Consider the FD: $A \rightarrow AB$. By reflexivity, A always implies A, so that the A on the right hand side is not necessary and can be dropped. The proper form, without the trivial dependency would then be $A \rightarrow B$.

3. Consider the set of FD: $AB \rightarrow CD$ and $B \rightarrow C$. AB is obviously a key for this relation since $AB \rightarrow CD$ implies $AB \rightarrow ABCD$. It is a primary key since there are no smaller subsets of keys that hold over *R(A,B,C,D)*. The FD: $B \rightarrow C$ violates 2NF since:
   - $C \in B$ is false; that is, it *is not* a trivial FD
   - B *is not* a superkey
   - C *is not* part of some key for R
   - B *is* a proper subset of the key AB (transitive dependency)
4. Consider the set of FD: $AB \rightarrow CD$ and $C \rightarrow D$. AB is obviously a key for this relation since $AB \rightarrow CD$ implies $AB \rightarrow ABCD$. It is a primary key since there are no smaller subsets of keys that hold over *R(A,B,C,D)*. The FD: $C \rightarrow D$ violates 3NF but not 2NF since:
   - $D \in C$ is false; that is, it *is not* a trivial FD
   - C *is not* a superkey
   - D *is not* part of some key for R
5. The only way R could be in BCNF is if B includes a key, *i.e.* B is a key for R.
6. It means that the relationship is one to one. That is, each A entity corresponds to at most one B entity and vice-versa. (In addition, we have the dependency $AB \rightarrow C$, from the semantics of a relationship set.)

Exercise 19.2 Consider a relation R with five attributes $ABCDE$. You are given the following dependencies: $A \rightarrow B$, $BC \rightarrow E$, and $ED \rightarrow A$.

1. List all keys for R.
2. Is R in 3NF?
3. Is R in BCNF?

Answer 19.2

1. CDE, ACD, BCD
2. R is in 3NF because B, E and A are all parts of keys.
3. R is not in BCNF because none of A, BC and ED contain a key.

Exercise 19.3 Consider the relation shown in Figure 19.1.

1. List all the functional dependencies that this relation instance satisfies.

X	Y	Z
x_1	y_1	z_1
x_1	y_1	z_2
x_2	y_1	z_1
x_2	y_1	z_3

Figure 19.1 Relation for Exercise 19.3.

2. Assume that the value of attribute Z of the last record in the relation is changed from z_3 to z_2. Now list all the functional dependencies that this relation instance satisfies.

Answer 19.3

1. The following functional dependencies hold over R: $Z \rightarrow Y$, $X \rightarrow Y$, and $XZ \rightarrow Y$
2. Same as part 1. Functional dependency set is unchanged.

Exercise 19.4 Assume that you are given a relation with attributes $ABCD$.

1. Assume that no record has NULL values. Write an SQL query that checks whether the functional dependency $A \rightarrow B$ holds.
2. Assume again that no record has NULL values. Write an SQL assertion that enforces the functional dependency $A \rightarrow B$.
3. Let us now assume that records could have NULL values. Repeat the previous two questions under this assumption.

Answer 19.4 Assuming...

1. The following statement returns 0 iff no statement violates the FD $A \rightarrow B$.

```
SELECT COUNT (*)
FROM   R AS R1, R AS R2
WHERE  (R1.B != R2.B) AND (R1.A = R2.A)
```

2.
```
CREATE ASSERTION ADeterminesB
CHECK  ((SELECT COUNT (*)
         FROM   R AS R1, R AS R2
         WHERE  (R1.B != R2.B) AND (R1.A = R2.A))
         =0)
```

3. Note that the following queries can be written with the NULL and NOT NULL interchanged. Since we are doing a full join of a table and itself, we are creating tuples in sets of two therefore the order is not important.

```
SELECT COUNT (*)
FROM   R AS R1, R AS R2
WHERE  ((R1.B != R2.B) AND (R1.A = R2.A))
       OR ((R1.B is NULL) AND (R2.B is NOT NULL)
              AND (R1.A = R2.A))

CREATE ASSERTION ADeterminesBNull
CHECK  ((SELECT COUNT (*)
       FROM   R AS R1, R AS R2
       WHERE  ((R1.B != R2.B) AND (R1.A = R2.A)))
              OR ((R1.B is NULL) AND (R2.B is NOT NULL)
                     AND (R1.A = R2.A))
       =0)
```

Exercise 19.5 Consider the following collection of relations and dependencies. Assume that each relation is obtained through decomposition from a relation with attributes *ABCDEFGHI* and that all the known dependencies over relation *ABCDEFGHI* are listed for each question. (The questions are independent of each other, obviously, since the given dependencies over *ABCDEFGHI* are different.) For each (sub)relation: (a) State the strongest normal form that the relation is in. (b) If it is not in BCNF, decompose it into a collection of BCNF relations.

1. *R1(A,C,B,D,E)*, $A \rightarrow B$, $C \rightarrow D$
2. *R2(A,B,F)*, $AC \rightarrow E$, $B \rightarrow F$
3. *R3(A,D)*, $D \rightarrow G$, $G \rightarrow H$
4. *R4(D,C,H,G)*, $A \rightarrow I$, $I \rightarrow A$
5. *R5(A,I,C,E)*

Answer 19.5

1. 1NF. BCNF decomposition: AB, CD, ACE.
2. 1NF. BCNF decomposition: AB, BF
3. BCNF.
4. BCNF.
5. BCNF.

Exercise 19.6 Suppose that we have the following three tuples in a legal instance of a relation schema S with three attributes ABC (listed in order): (1,2,3), (4,2,3), and (5,3,3).

1. Which of the following dependencies can you infer does *not* hold over schema S?

   (a) $A \rightarrow B$, (b) $BC \rightarrow A$, (c) $B \rightarrow C$

2. Can you identify any dependencies that hold over S?

Answer 19.6

1. $BC\rightarrow A$ does not hold over S (look at the tuples (1,2,3) and (4,2,3)). The other tuples hold over S.

2. No. Given just an instance of S, we can say that certain dependencies (e.g., $A \rightarrow B$ and B $\rightarrow C$) are not violated by this instance, but we cannot say that these dependencies hold with respect to S. To say that an FD holds w.r.t. a relation is to make a statement about *all* allowable instances of that relation!

Exercise 19.7 Suppose you are given a relation R with four attributes $ABCD$. For each of the following sets of FDs, assuming those are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) Identify the best normal form that R satisfies (1NF, 2NF, 3NF, or BCNF). (c) If R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

1. $C \rightarrow D$, $C \rightarrow A$, $B \rightarrow C$
2. $B \rightarrow C$, $D \rightarrow A$
3. $ABC \rightarrow D$, $D \rightarrow A$
4. $A \rightarrow B$, $BC \rightarrow D$, $A \rightarrow C$
5. $AB \rightarrow C$, $AB \rightarrow D$, $C \rightarrow A$, $D \rightarrow B$

Answer 19.7

1. (a) Candidate keys: B
   (b) R is in 2NF but not 3NF.
   (c) $C \rightarrow D$ and $C \rightarrow A$ both cause violations of BCNF. One way to obtain a (lossless) join preserving decomposition is to decompose R into AC, BC, and CD.
2. (a) Candidate keys: BD
   (b) R is in 1NF but not 2NF.

(c) Both $B \rightarrow C$ and $D \rightarrow A$ cause BCNF violations. The decomposition: AD, BC, BD (obtained by first decomposing to AD, BCD) is BCNF and lossless and join-preserving.

3. (a) Candidate keys: ABC, BCD

   (b) R is in 3NF but not BCNF.

   (c) $ABCD$ is not in BCNF since $D \rightarrow A$ and D is not a key. However if we split up R as AD, BCD we cannot preserve the dependency $ABC \rightarrow D$. So there is no BCNF decomposition.

4. (a) Candidate keys: A

   (b) R is in 2NF but not 3NF (because of the FD: $BC \rightarrow D$).

   (c) $BC \rightarrow D$ violates BCNF since BC does not contain a key. So we split up R as in: BCD, ABC.

5. (a) Candidate keys: AB, BC, CD, AD

   (b) R is in 3NF but not BCNF (because of the FD: C → A).

   (c) $C \rightarrow A$ and $D \rightarrow B$ both cause violations. So decompose into: AC, BCD but this does not preserve $AB \rightarrow C$ and $AB \rightarrow D$, and BCD is still not BCNF because $D \rightarrow B$. So we need to decompose further into: AC, BD, CD. However, when we attempt to revive the lost functioanl dependencies by adding ABC and ABD, we that these relations are not in BCNF form. Therefore, there is no BCNF decomposition.

Exercise 19.8 Consider the attribute set $R = ABCDEGH$ and the FD set $F = \{AB \rightarrow C,\ AC \rightarrow B,\ AD \rightarrow E,\ B \rightarrow D,\ BC \rightarrow A,\ E \rightarrow G\}$.

1. For each of the following attribute sets, do the following: (i) Compute the set of dependencies that hold over the set and write down a minimal cover. (ii) Name the strongest normal form that is not violated by the relation containing these attributes. (iii) Decompose it into a collection of BCNF relations if it is not in BCNF.

   (a) ABC, (b) $ABCD$, (c) $ABCEG$, (d) $DCEGH$, (e) $ACEH$

2. Which of the following decompositions of $R = ABCDEG$, with the same set of dependencies F, is (a) dependency-preserving? (b) lossless-join?

   (a) $\{AB,\ BC,\ ABDE,\ EG\}$

   (b) $\{ABC,\ ACDE,\ ADG\}$

Answer 19.8

1. (a) i. $R1 = ABC$: The FD's are $AB \rightarrow C$, $AC \rightarrow B$, $BC \rightarrow A$.

ii. This is already a minimal cover.

iii. This is in BCNF since AB, AC and BC are candidate keys for $R1$. (In fact, these are all the candidate keys for $R1$).

(b) i. $R2 = ABCD$: The FD's are $AB \rightarrow C$, $AC \rightarrow B$, $B \rightarrow D$, $BC \rightarrow A$.

ii. This is a minimal cover already.

iii. The keys are: AB, AC, BC. $R2$ is not in BCNF or even 2NF because of the FD, $B \rightarrow D$ (B is a proper subset of a key!) However, it is in 1NF. Decompose as in: ABC, BD. This is a BCNF decomposition.

(c) i. $R3 = ABCEG$; The FDs are $AB \rightarrow C$, $AC \rightarrow B$, $BC \rightarrow A$, $E \rightarrow G$.

ii. This is in minimal cover already.

iii. The keys are: ABE, ACE, BCE. It is not even in 2NF since E is a proper subset of the keys and there is a FD $E \rightarrow G$. It is in 1NF . Decompose as in: ABE, ABC, EG. This is a BCNF decompostion.

(d) i. $R4 = DCEGH$; The FD is $E \rightarrow G$.

ii. This is in minimal cover already.

iii. The key is $DCEH$; It is not in BCNF since in the FD $E \rightarrow G$, E is a subset of the key and is not in 2NF either. It is in 1 NF Decompose as in: $DCEH$, EG

(e) i. $R5 = ACEH$; No FDs exist.

ii. This is a minimal cover.

iii. Key is $ACEH$ itself.

iv. It is in BCNF form.

2. (a) The decomposition. { AB, BC, ABDE, EG } is *not* lossless. To prove this consider the following instance of R:

$$\{(a_1, b, c_1, d_1, e_1, g_1),\ (a_2, b, c_2, d_2, e_2, g_2)\}$$

Because of the functional dependencies $BC \rightarrow A$ and $AB \rightarrow C$, $a_1 \neq a_2$ if and only if $c_1 \neq c_2$. It is easy to that the join AB $\bowtie$ BC contains 4 tuples:

$$\{(a_1, b, c_1),\ (a_1, b, c_2),\ (a_2, b, c_1),\ (a_2, b, c_2)\}$$

So the join of AB, BC, $ABDE$ and EG will contain at least 4 tuples, (actually it contains 8 tuples) so we have a lossy decomposition here.

This decomposition does not preserve the FD, $AB \rightarrow C$ (or $AC \rightarrow B$)

(b) The decomposition {ABC, ACDE, ADG } is lossless. Intuitively, this is because the join of ABC, $ACDE$ and ADG can be constructed in two steps; first construct the join of ABC and ACDE: this is lossless because their (attribute) intersection is AC which is a key for $ABCDE$ (in fact $ABCDEG$) so this is lossless. Now join this intermediate join with ADG. This is also lossless because the attribute intersection is is AD and $AD \rightarrow ADG$. So by the test mentioned in the text this step is also a lossless decomposition.

The projection of the FD's of R onto ABC gives us: $AB \rightarrow C$, $AC \rightarrow B$ and $BC \rightarrow A$. The projection of the FD's of R onto $ACDE$ gives us: $AD \rightarrow E$ and The projection of the FD's of R onto ADG gives us: $AD \rightarrow G$ (by transitivity) The closure of this set of dependencies does not contain $E \rightarrow G$ nor does it contain $B \rightarrow D$. So this decomposition is not dependency preserving.

Exercise 19.9 Let R be decomposed into R_1, R_2, ..., R_n. Let F be a set of FDs on R.

1. Define what it means for F to *be preserved* in the set of decomposed relations.
2. Describe a polynomial-time algorithm to test dependency-preservation.
3. Projecting the FDs stated over a set of attributes X onto a subset of attributes Y requires that we consider the closure of the FDs. Give an example where considering the closure is important in testing dependency-preservation, that is, considering just the given FDs gives incorrect results.

Answer 19.9

1. Let F_i denote the projection of F on R_i. F is *preserved* if the closure of the (union of) the F_i's equals F (note that F is always a superset of this closure.)
2. We shall describe an algorithm for testing dependency preservation which is polynomial in the cardinality of F. For each dependency X $\rightarrow$ Y $\in$ F check if it is in F as follows: start with the set S (of attributes in) X. For each relation R_i, compute the closure of $S \cap R_i$ relative to F and project this closure to the attributes of R_i. If this results in additional attributes, add them to S. Do this repeatedly until there is no change to S.
3. There is an example in the text in Section 19.5.2.

Exercise 19.10 Suppose you are given a relation *R(A,B,C,D)*. For each of the following sets of FDs, assuming they are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) State whether or not the proposed decomposition of R into smaller relations is a good decomposition and briefly explain why or why not.

1. $B \rightarrow C$, $D \rightarrow A$; decompose into BC and AD.
2. $AB \rightarrow C$, $C \rightarrow A$, $C \rightarrow D$; decompose into ACD and BC.
3. $A \rightarrow BC$, $C \rightarrow AD$; decompose into ABC and AD.
4. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB and ACD.

5. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB, AD and CD.

Answer 19.10

1. Candidate key(s): BD. The decomposition into BC and AD is unsatisfactory because it is lossy (the join of BC and AD is the cartesian product which could be much bigger than $ABCD$)

2. Candidate key(s): AB, BC. The decomposition into ACD and BC is lossless since $ACD \cap BC$ (which is C) $\rightarrow ACD$. The projection of the FD's on ACD include C $\rightarrow$ D, C $\rightarrow$ A (so C is a key for ACD) and the projection of FD on BC produces no nontrivial dependencies. In particular this is a BCNF decomposition (check that R is not!). However, it is not dependency preserving since the dependency AB $\rightarrow$ C is not preserved. So to enforce preservation of this dependency (if we do not want to use a join) we need to add ABC which introduces redundancy. So implicitly there is some redundancy across relations (although none inside ACD and BC).

3. Candidate key(s): A, C. Since A and C are both candidate keys for R, it is already in BCNF. So from a normalization standpoint it makes no sense to decompose R. Further more, the decompose is not dependency-preserving since $C \rightarrow AD$ can no longer be enforced.

4. Candidate key(s): A. The projection of the dependencies on AB are: $A \rightarrow B$ and those on ACD are: $A \rightarrow C$ and $C \rightarrow D$ (rest follow from these). The scheme ACD is not even in 3NF, since C is not a superkey, and D is not part of a key. This is a lossless-join decomposition (since A is a key), but not dependency preserving, since $B \rightarrow C$ is not preserved.

5. Candidate key(s): A (just as before) This is a lossless BCNF decomposition (easy to check!) This is, however, not dependency preserving (B *consider* $\rightarrow C$). So it is not free of (implied) redundancy. This is not the best decomposition (the decomposition AB, BC, CD is better.)

Exercise 19.11 Consider a relation R that has three attributes ABC. It is decomposed into relations R_1 with attributes AB and R_2 with attributes BC.

1. State the definition of a lossless-join decomposition with respect to this example. Answer this question concisely by writing a relational algebra equation involving R, R_1, and R_2.

2. Suppose that $B \rightarrow C$. Is the decomposition of R into R_1 and R_2 lossless-join? Reconcile your answer with the observation that neither of the FDs $R_1 \cap R_2 \rightarrow R_1$ nor $R_1 \cap R_2 \rightarrow R_2$ hold, in light of the simple test offering a necessary and sufficient condition for lossless-join decomposition into two relations in Section 15.6.1.

3. If you are given the following instances of R_1 and R_2, what can you say about the instance of R from which these were obtained? Answer this question by listing tuples that are definitely in R and tuples that are possibly in R.

   Instance of R_1 = {(5,1), (6,1)}
   Instance of R_2 = {(1,8), (1,9)}

   Can you say that attribute B definitely *is* or *is not* a key for R?

Answer 19.11

1. The decomposition of R into R1 and R2 is lossless if and only if:

   $$R1 \bowtie_{R1.B=R2.B} R2 \quad = \quad R$$

   Note that this is a statement about relation schemas, not some specific instances of them.

2. Answer Omitted.

3. All we can say is that the instance of R from which the given instances of R_1 and R_2 were obtained, must be a subset of the set of ABC tuples: {(5,1,8), (5,1,9), (6,1,8), (6,1,9)} which is also, at the same time, a superset of {(5,1,8), (6,1,9)} or a superset of {(5,1,9), (6,1,8)}. In particular, R contains at least two tuples but no more than 4. This also implies the attribute B is *not* a key for R (because R has at least 2 distinct tuples but each tuple in R has the same B value.)

Exercise 19.12 Suppose that we have the following four tuples in a relation S with three attributes ABC: (1,2,3), (4,2,3), (5,3,3), (5,3,4). Which of the following functional ($\rightarrow$) and multivalued ($\rightarrow\!\!\rightarrow$) dependencies can you infer does *not* hold over relation S?

1. $A \rightarrow B$
2. $A \rightarrow\!\!\rightarrow B$
3. $BC \rightarrow A$
4. $BC \rightarrow\!\!\rightarrow A$
5. $B \rightarrow C$
6. $B \rightarrow\!\!\rightarrow C$

Answer 19.12

1. (A $\rightarrow$ B) Cannot say anything.

2. (A →→ B) Cannot say anything.
3. (BC → A) does *not* hold. (Look at the tuples (1,2,3) and (4,2,3) the BC-values are the same but A values differ.)
4. (BC →→ A) Cannot say anything.
5. (B → C) does *not* hold. (Look at the tuples (5,3,3) and (5,3,4))
6. (B →→ C) Cannot say anything. The tuples (5,3,3) and (5,3,4) could shed some light if their A-values differed but that is not the case, here.

   In summary, we can conclude from the given information that BC → A and B → C do not hold.

Exercise 19.13 Consider a relation R with five attributes $ABCDE$.

1. For each of the following instances of R, state whether it violates (a) the FD $BC \rightarrow D$ and (b) the MVD $BC \twoheadrightarrow D$:
   (a) { } (i.e., empty relation)
   (b) {(a,2,3,4,5), (2,a,3,5,5)}
   (c) {(a,2,3,4,5), (2,a,3,5,5), (a,2,3,4,6)}
   (d) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5)}
   (e) {(a,2,3,4,5), (2,a,3,7,5), (a,2,3,4,6)}
   (f) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5), (a,2,3,6,6)}
   (g) {(a,2,3,4,5), (a,2,3,6,5), (a,2,3,6,6), (a,2,3,4,6)}
2. If each instance for R listed above is legal, what can you say about the FD $A \rightarrow B$?

Answer 19.13

1. **Note**: The answer sometimes depends on the value of a. Unless otherwise mentioned, the answer applies to all values of a.

   (a) { } (i.e., empty relation):
   does not violate either dependency.
   (b) {(a,2,3,4,5), (2,a,3,5,5)}:
   If a = 2, then BC → D is violated (otherwise it is not).
   BC →→ D is not violated (for any value of a)

(c) {(a,2,3,4,5), (2,a,3,5,5), (a,2,3,4,6)}:
BC → D is violated if a = 2 (otherwise not).
If a = 2 then BC →→ D is violated (consider the tuples (2,a,3,5,5) and (a,2,3,4,6); if a equals 2 must also have (2,a,3,5,6))

(d) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5)}:
BC → D is violated (consider the first and the third tuples (a,2,3,4,5) and (a,2,3,6,5)).
BC →→ D is not violated.

(e) {(a,2,3,4,5), (2,a,3,7,5), (a,2,3,4,6)}:
If a = 2, then BC → D is violated (otherwise it is not).
If a = 2, then BC →→ D is violated (otherwise it is not). To prove this look at the last two tuples; there must also be a tuple (2,a,3,7,6) for BC →→ to hold.

(f) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5), (a,2,3,6,6)}:
BC → D does not hold. (Consider the first and the third tuple).
BC →→ C is violated. Consider the 1st and the 4th tuple. For this dependency to hold there should be a tuple (a,2,3,4,6).

(g) {(a,2,3,4,5), (a,2,3,6,5), (a,2,3,6,6), (a,2,3,4,6)}:
BC → D does not hold. (Consider the first and the third tuple).
BC →→ C is not violated.

2. We can *not* say anything about the functional dependency A → B.

Exercise 19.14 JDs are motivated by the fact that sometimes a relation that cannot be decomposed into two smaller relations in a lossless-join manner can be so decomposed into three or more relations. An example is a relation with attributes *supplier*, *part*, and *project*, denoted *SPJ*, with no FDs or MVDs. The JD $\bowtie \{SP,\ PJ,\ JS\}$ holds.

From the JD, the set of relation schemes *SP, PJ,* and *JS* is a lossless-join decomposition of *SPJ*. Construct an instance of *SPJ* to illustrate that no two of these schemes suffice.

Answer 19.14
Consider the following instance of the schema SPJ:

$$SPJ = \{(s_1, p_1, j_1),\ (s_2, p_1, j_2),\ (s_1, p_2, j_2),\ (s_1, p_1, j_2)\}$$

Then

$$\begin{aligned} SP &= \{(s_1, p_1),\ (s_1, p_2),\ (s_2, p_1)\} \\ PJ &= \{(p_1, j_1),\ (p_1, j_2),\ (p_2, j_2)\} \\ JS &= \{(j_1, s_1),\ (j_2, s_1),\ (j_2, s_2)\} \end{aligned}$$

Let us compute all three 2-joins:

$$\begin{aligned} SP \bowtie PJ &= \{(s_1, p_1, j_1), (s_1, p_1, j_2), (s_2, p_1, j_1), (s_2, p_1, j_2), (s_1, p_2.j_2)\} \\ PJ \bowtie JS &= \{(s_1, p_1, j_1), (s_1, p_1, j_2), (s_1, p_2, j_2), (s_2, p_1, j_2), (s_2, p_2, j_2)\} \\ SP \bowtie JS &= \{(s_1, p_1, j_1), (s_1, p_1, j_2), (s_1, p_2, j_1), (s_1, p_2, j_2), (s_2, p_1, j_2)\} \end{aligned}$$

none of which equals SPJ. But, on the other hand, SPJ is the join of all three (if you put sufficiently many "constraints " you can always reconstruct the original relation SPJ by taking the join. Joining SP, PJ and JS amounts to putting all possible equality constraints. I am only giving you the intuition here; you need to work this out and check the details!) **QED.**

Exercise 19.15 Answer the following questions

1. Prove that the algorithm shown in Figure 19.4 correctly computes the attribute closure of the input attribute set X.
2. Describe a linear-time (in the size of the set of FDs, where the size of each FD is the number of attributes involved) algorithm for finding the attribute closure of a set of attributes with respect to a set of FDs. Prove that your algorithm correctly computes the attribute closure of the input attribute set.

Answer 19.15 The answer to each question is given below.

1. Proof Omitted.
2. Recall that the *attribute closure* of (attribute) X relative to a set of FD's Σ is the set of attributes A such that Σ satisfies $X \rightarrow A$.

```
// Initialize
X+    := X;
FdSet := Σ;

do
{
      for each FD Y → Z in FdSet such that X+ ⊇ Y
      {
          X+ := X+ union Z;
          Remove Y → Z from FdSet;
      }
} until ( X+ does not change) ;
```

Let $n = |\ \Sigma\ |$ denote the cardinality of Σ. Then the loop repeats at most n times since for each iteration we either permanently remove a functional dependency from the set Σ, or stop all together. With the proper choice of data structures it can be show that this algorithm is linear in the size of Σ.

As for correctness, we claim that this algorithm is equivalent to the standard attribute closure algorithm. If we throw away a functional dependency $Y \rightarrow Z$ at a given step, then it must be the case that $X \rightarrow Y$ since $Y \in X^+$ at that step, therefore by transitivity $X \rightarrow Z$. Since Z was added to X^+ we no longer need the functional dependency $Y \rightarrow Z$ for finding the attribute closure of X, since it is implied by $X \rightarrow X^+$.

The rest of the proof of correctness follows from part 1 of this exercise.

Exercise 19.16 Let us say that an FD $X \rightarrow Y$ is *simple* if Y is a single attribute.

1. Replace the FD $AB \rightarrow CD$ by the smallest equivalent collection of simple FDs.
2. Prove that every FD $X \rightarrow Y$ in a set of FDs F can be replaced by a set of simple FDs such that F^+ is equal to the closure of the new set of FDs.

Answer 19.16

1. We claim $\{\ AB \rightarrow C,\ AB \rightarrow D\ \}$ is the smallest such set. First, this collection is equivalent to the single FD: $AB \rightarrow CD$ *i.e.* every database that satisfies the first also satisfies the second, and vice versa. Also no proper subset of this collection satisfies this property.
2. Replace each FD: $X_1X_2 \ldots X_m \rightarrow Y_1Y_2 \ldots Y_n$ by the collection $\{X_1 \ldots X_m \rightarrow Y_i \ \ | \ \ i \leq i \leq n\}$. By using the *decomposition* and *union* axioms, it is easy to show that we can go from one representation to the other both forwards and back. Note however, that this may not be the minimal such set.

Exercise 19.17 Prove that Armstrong's Axioms are sound and complete for FD inference. That is, show that repeated application of these axioms on a set F of FDs produces exactly the dependencies in F^+.

Answer 19.17 Proof Omitted.

Exercise 19.18 Consider a relation R with attributes $ABCDE$. Let the following FDs be given: $A \rightarrow BC$, $BC \rightarrow E$, and $E \rightarrow DA$. Similarly, let S be a relation with attributes $ABCDE$ and let the following FDs be given: $A \rightarrow BC$, $B \rightarrow E$, and $E \rightarrow DA$. (Only the second dependency differs from those that hold over R.) You do not know whether or which other (join) dependencies hold.

1. Is R in BCNF?
2. Is R in 4NF?
3. Is R in 5NF?
4. Is S in BCNF?
5. Is S in 4NF?
6. Is S in 5NF?

Answer 19.18

1. The schema R has keys A, E and BC. It follows that R is indeed in BCNF.
2. By Exercise 26 (part 1), it follows that R is also in 4NF (since the relation scheme has a single-attribute key).
3. R is in 5NF because the schema does not have any JD (besides those that are implied by the FD's of the schema; but these cannot violate the 5NF condition). Note that this alternative argument may be used in some of the other parts of this problem as well.
4. The schema S has keys A, B and E. It follows that S is indeed in BCNF.
5. By exercise 26 (part 1), it follows that S is also in 4NF (since the relation scheme has a single-attribute key).
6. By exercise 26 (part 2), it follows that S is also in 5NF (since each key is a single-attribute key.)

Exercise 19.19 Let R be a relation schema with a set F of FDs. Prove that the decomposition of R into R_1 and R_2 is lossless-join if and only if F^+ contains $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

Answer 19.19 For both directions (if and only-if) we use the notation

$C = R_1 \cap R_2, \quad X = R_1 - C, \quad Y = R_2 - C$, so that $R_1 = XC, \quad R_2 = CY$, and $R = XCY$.

($\Leftarrow$): For this direction, assume we are given the dependency $C \rightarrow X$. (The other case $C \rightarrow Y$ is similar.)

So let r be an instance of schema R and let (x_1, c, y_1) and (x_2, c, y_2) be two tuples in r. The FD, $C \rightarrow X$ implies that $x_1 = x_2$. Thus, (x_1, c, y_2) is the same as (x_2, c, y_2) and (x_2, c, y_1) is the same as (x_1, c, y_1), so that both these tuples (x_1, c, y_2) and (x_2, c, y_1) are in r. Thus r satisfies the JD: $R = R_1 \bowtie R_2$. Since r is an arbitrary instance, we have proved that the decomposition is lossless.

($\Rightarrow$): Now for the other direction, assume that neither $C \rightarrow X$ nor $C \rightarrow Y$ holds. We shall prove that the join is lossy by exhibiting a relation instance that violates the JD: $R_1 \bowtie R_2$. Actually we will prove a slightly more general result. Suppose we are given some set of FD's Σ, such that R has a lossless join w.r.t. Σ. This means that for any instance r satisfying Σ, we have

$$r = r_1 \bowtie r_2 \text{ where } r_1 = \pi_{R_1}(r),\ r_2 = \pi_{R_2}(r).$$

Then we prove that

$$\{C \rightarrow X,\ C \rightarrow Y\} \ \cap\ \Sigma^+ \neq \emptyset.$$

The proof is by contradiction. Assume that the intersection $\{\ C \rightarrow X, C \rightarrow Y\ \} \cap \Sigma^+$ is empty. Suppose r_1 is an instance of the schema that does not satisfy the FD: $C \rightarrow X$ and r_2 is an instance that does not satisfy the FD: $C \rightarrow Y$. Choose c such that there are tuples $(x_1, c, y_1),\ (x_2, c, y_2) \in r_1$ for which $x_1 \neq x_2$ and c' such that there are tuples $(x_1', c', y_1'),\ (x_2', c', y_2') \in r_2$ for which ${y_1}' \neq {y_2}'$.

Use selection to replace r_1 by $\pi_{R.C=c}(r_1)$ and r_2 by $\pi_{R.C=c'}(r_2)$. Since r_1 and r_2 are finite and the domain sets are infinite, we can assume without loss of generality (by modifying some of the values of the tuples in r_1 and r_2, if necessary) so that

$$\begin{aligned} c &= c' \\ \pi_A(r_1) \cap \pi_A(r_2) &= \emptyset \quad \text{for each attribute } A \in X \\ \pi_B(r_1) \cap \pi_B(r_2) &= \emptyset \quad \text{for each attribute } B \in Y. \end{aligned}$$

Now consider the relation $r_1 \cup r_2$. This is an instance of the schema R that satisfies Σ. However, $(x_1, c, y_1') \notin r_1 \cup r_2$, so the instance $r_1 \cup r_2$ does not satisfy the JD: $R_1 \bowtie R_2$.

Exercise 19.20 Consider a scheme R with FDs F that is decomposed into schemes with attributes X and Y. Show that this is dependency-preserving if $F \subseteq (F_X \cup F_Y)^+$.

Answer 19.20 We need to show that $F \subseteq (F_X \cup F_Y)^+$ implies $F^+ = (F_X \cup F_Y)^+$. We can do this by showing the containments, $(F_X \cup F_Y)^+ \subseteq F^+$ and $F^+ \subseteq (F_X \cup F_Y)^+$, are both true. We make use the following two observations:

1. If $A \subseteq B$ are two sets of FD's then $A^+ \subseteq B^+$ and
2. $A^{++} = A^+$.

The includsion $(F_X \cup F_Y)^+ \subseteq F^+$ follows from observing that, by definition, $F_X \subseteq F^+$ and $F_Y \subseteq F^+$ so that $F_X \cup F_Y \subseteq F^+$ (now apply observations 1 and 2).

The other containment, $F^+ \subseteq (F_X \cup F_Y)^+$ follows from the hypothesis, $F \subseteq (F_X \cup F_Y)^+$ and observations 1 and 2.

Therefore, since both containments are true, $F^+ = (F_X \cup F_Y)^+$ also holds, which means the decomposition is dependency preserving.

Exercise 19.21 Prove that the optimization of the algorithm for lossless-join, dependency-preserving decomposition into 3NF relations (Section 19.6.2) is correct.

Answer 19.21 Answer Omitted.

Exercise 19.22 Prove that the 3NF synthesis algorithm produces a lossless-join decomposition of the relation containing all the original attributes.

Answer 19.22
Proof: Let R denote the set of all attributes. N is a minimal cover for the set of all FD's satsified by the schema and K some key for the schema. We will show that the decomposition $\{XA|X \rightarrow A \in N\} \quad \{K\}$, where K is any key gives a lossless join 3NF decomposition. First, note that the subschema K is in 3NF because any FD that holds over it will have its right hand side (attribute) contained in a key (namely, K).

The proof that the decomposition is lossless is a little complicated notationally but the basic idea is this: Enumerate the set of subschema XA in the decomposition as $R_1, R_2, \ldots R_m$. Let r be an instance and let t_i and a be tuples in r. We need to show the "joins" of all these tuples are also in r. A formal proof of this will proceed by induction and is based on using the tuple a and the fact that K is a key dependency that follows from the FD's $X \rightarrow A$ to "connect" the other tuples and force each tuple in the join to lie in r.

Exercise 19.23 Prove that an MVD $X \twoheadrightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie \{XY,\ X(R-Y)\}$.

Answer 19.23 Write $Z = R - Y$. Thus, $R = YXZ$. $X \twoheadrightarrow Y$ says that if $(y_1, x, z_1), (y_2, x, z_2) \in R$ then $(y_1, x, z_2), (y_2, x, z_1)$ also $\in R$. But this is precisely the same as saying $R = \bowtie \{\ XY,\ X(R-Y)\ \}$.

Exercise 19.24 Prove that, if R has only one key, it is in BCNF if and only if it is in 3NF.

Answer 19.24 Let F (F^+) denote the (closure of the) set of functional dependencies satisfied by the schema R which is assumed to be in 3NF. We need to show that for each nontrivial dependency $X \rightarrow A$ in F^+, X is a superkey. To this end, consider such a dependency. If X is *not* a superkey, the 3NF property guarantees that the attribute A is part of a key. Since all keys are simple by assumption, we have that A is a key. This last fact together with the dependency $X \rightarrow A$ implies that X is a superkey (this follows, from the transitivity axiom) which is a contradiction.

Exercise 19.25 Prove that, if R is in 3NF and every key is simple, then R is in BCNF.

Answer 19.25 Since every key is simple, then we know that for any FD that satisfies $X \rightarrow A$, where A is part of some key implies that A is a key. By the definition of an FD, if X is known, then A is known. This means that if X is known, we know a key for the relation, so X must be a superkey. This satisfies all of the properties of BCNF.

Exercise 19.26 Prove these statements:

1. If a relation scheme is in BCNF and at least one of its keys consists of a single attribute, it is also in 4NF.
2. If a relation scheme is in 3NF and each key has a single attribute, it is also in 5NF.

Answer 19.26 The answer to each question is given below.

1. Proof Omitted.
2. Proof Omitted.

Exercise 19.27 Give an algorithm for testing whether a relation scheme is in BCNF. The algorithm should be polynomial in the size of the set of given FDs. (The *size* is the sum over all FDs of the number of attributes that appear in the FD.) Is there a polynomial algorithm for testing whether a relation scheme is in 3NF?

Answer 19.27 Let $|F|$ denote the size of the representation of the schema *i.e.,* set of all the FD's of the schema. Also, let $|f|$ denote the number of FD's in the schema. By exercise 19.15 we know that for each attribute in the schema, we can compute the attribute closure of its left hand side in time $O(|F|)$.

The algorithm to test if R is in BCNF consists of computing the attribute closure of the left hand side of each FD. If one of them doesn not equal U where U is the set of all attributes, then R is not in BCNF. Otherwise conclude that R is in BCNF.

Clearly the worst case complexity of this algorithm is $O(|f| \cdot |F|)$. Since $|f|$ is bounded by $|F|$ this yields a polynomial time algorithm in $|F|$.

On the other hand, *a priori*, there is no polynomial time algorithm for testing for 3NF. This is because to test whether or not a given FD violates 3NF we may need to check if the right hand side is prime *i.e.,* is a subset of some key of the schema. But identifying (all) the keys of the schema involves checking all subsets of U, and there are $2^{|U|}$ many of them. This last *prime attribute* problem is known to be NP-complete and the 3NF problem is clearly as hard (in fact polynomially reducible to the other) and, hence is also NP-complete.

Exercise 19.28 Give an algorithm for testing whether a relation scheme is in BCNF. The algorithm should be polynomial in the size of the set of given FDs. (The 'size' is the sum over all FDs of the number of attributes that appear in the FD.) Is there a polynomial algorithm for testing whether a relation scheme is in 3NF?

Answer 19.28 Incorrect question listed in the textbook. Please see solution to Exercise 19.27.

Exercise 19.29 Prove that the algorithm for decomposing a relation schema with a set of FDs into a collection of BCNF relation schemas as described in Section 19.6.1 is correct (i.e., it produces a collection of BCNF relations, and is lossless-join) and terminates.

Answer 19.29 First, we will repeat the algorithm so as to keep consistent notation:

1. Let $X \subset R$, A be a single atribute in R and $X \rightarrow A$ be a FD that causes a violation of BCNF. Decompose into $R - A$ and XA.
2. If either $R - A$ or XA is not in BCNF, decompose them further by a recursive application of this algorithm.

Proving the correctness of the algorithm is divided into 3 parts:

- **Proof that every Decomposition is Lossless:**
  For any decomposition of a relation R into $R - A$ and XA that the algorithm takes, it is trivially loseless by Thoerem 3 of this chapter. First, we claim that $(R - A) \bigcap (XA) = X$ since: $X \subset R$ by construction, and A is not in X (else it would be a trivially functional dependency and not violate BCNF, which is a contradiction). The given functional dependency $X \rightarrow A$ then implies $X \rightarrow XA$ by the Union Rule, therefore $(R - A) \bigcap (XA) \rightarrow XA$ and by Thoerem 3, this decomposition is loseless. Note however, that this decomposition may not be dependency preserving.
- **Proof that Algorithm Terminates:**
  Every decomposition of a Relation R that the algorithm performs produces relations $R - A$ and XA with strictly fewer attributes then R. $R - A$ has strictly fewer attributes than R since by construction A is not null, and since the functional dependency violates BCNF also by construction, A must be contained in R, else the functional dependency would not be applicable to R. Further, XA has strictly fewer attributes than R since by construction $X \subset R$ and $XA \neq R$. This is clear since if we assume that $XA = R$, then we can conclude that X is a superkey because $XA \rightarrow R$ trivially and $X \rightarrow A$, so that $X \rightarrow R$ from Transivitivty. This

would contradict the assumption that the functional dependency violates BCNF, leaving us to conclude that $XA \neq R$.

If we let n denote the number of attributes in the original relation R, then there are at most $(2^n - 1)$ decompositions the algorithm will perform before it terminates. Once a relation contains just a single attribute, it is in BCNF and cannot be decomposed further since there are no non-trivial functional dependencies we can apply to it that would violate BCNF.

- **Proof that every Relation in Final Set is in BCNF:**
  As discussed in the previous part of the problem, in the worst case the algorithm decomposes R into a set of n unique single attribute relations where n is the number of attributes in the original relation R. As also discussed above, each relation is clearly in BCNF. The decomposition process may, and in most cases should, terminate before we are down to all single attribute relations but irregardless, the algorithm will only stop when all subsets of R are in BCNF.

20

PHYSICAL DATABASE DESIGN AND TUNING

Exercise 20.1 Consider the following BCNF schema for a portion of a simple corporate database (type information is not relevant to this question and is omitted):

Emp (*eid, ename, addr, sal, age, yrs, deptid*)
Dept (*did, dname, floor, budget*)

Suppose you know that the following queries are the six most common queries in the workload for this corporation and that all six are roughly equivalent in frequency and importance:

- List the id, name, and address of employees in a user-specified age range.
- List the id, name, and address of employees who work in the department with a user-specified department name.
- List the id and address of employees with a user-specified employee name.
- List the overall average salary for employees.
- List the average salary for employees of each age; that is, for each age in the database, list the age and the corresponding average salary.
- List all the department information, ordered by department floor numbers.

1. Given this information, and assuming that these queries are more important than any updates, design a physical schema for the corporate database that will give good performance for the expected workload. In particular, decide which attributes will be indexed and whether each index will be a clustered index or an unclustered index. Assume that B+ tree indexes are the only index type supported by the DBMS and that both single- and multiple-attribute keys are permitted. Specify your physical design by identifying the attributes you recommend indexing on via clustered or unclustered B+ trees.

2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - List the id and address of employees with a user-specified employee name.
   - List the overall maximum salary for employees.
   - List the average salary for employees by department; that is, for each *deptid* value, list the *deptid* value and the average salary of employees in that department.
   - List the sum of the budgets of all departments by floor; that is, for each floor, list the floor and the sum.
   - Assume that this workload is to be tuned with an automatic index tuning wizard. Outline the main steps in the execution of the index tuning algorithm and the set of candidate configurations that would be considered.

Answer 20.1 The answer to each question is given below.

1.
   - If we create a dense unclustered B+ tree index on $\langle age, sal \rangle$ of the Emp relation we will be able to do an index-only scan to answer the 5th query. A hash index would not serve our purpose here, since the data entries will not be ordered by *age*! If index only scans are not allowed create a clustered B+ tree index on just the *age* field of Emp.
   - We should create an unclustered B+Tree index on *deptid* of the Emp relation and another unclustered index on $\langle dname, did \rangle$ in the Dept relation. Then, we can do an index only search on Dept and then get the Emp records with the proper *deptid*'s for the second query.
   - We should create an unclustered index on *ename* of the Emp relation for the third query.
   - We want a clustered sparse B+ tree index on $floor$ of the Dept index so we can get the department on each floor in $floor$ order for the sixth query.
   - Finally, a dense unclustered index on *sal* will allow us to average the salaries of all employees using an index only-scan. However, the dense unclustered B+ tree index on $\langle age, sal \rangle$ that we created to support Query (5) can also be used to compute the average salary of all employees, and is almost as good for this query as an index on just *sal*. So we should not create a separate index on just *sal*.

2.
   - We should create an unclustered B+Tree index on *ename* for the Emp relation so we can efficiently find employees with a particular name for the first query. This is not an index-only plan.
   - An unclustered B+ tree index on *sal* for the Emp relation will help find the maximum salary for the second query. (This is better than a hash index because the aggregate operation involved is `MAX`—we can simply go down to the rightmost leaf page in the B+ tree index.) This is not an index-only plan.

- We should create a dense unclustered B+ tree index on $\langle deptid, sal \rangle$ of the Emp relation so we can do an index-only scan on all of a department's employees. If index only plans are not supported, a sparse, clustered B+ tree index on *deptid* would be best. It would allow us to retrieve tuples by *deptid*.
- We should create a dense, unclustered index on $\langle floor, budget \rangle$ for Dept. This would allow us to sum budgets by floor using an index only plan. If index-only plans are not supported, we should create a sparse clustered B+ tree index on $floor$ for the Dept relation, so we can find the departments on each floor in order by floor.

Exercise 20.2 Consider the following BCNF relational schema for a portion of a university database (type information is not relevant to this question and is omitted):

Prof(*ssno, pname, office, age, sex, specialty, dept_did*)
Dept(*did, dname, budget, num_majors, chair_ssno*)

Suppose you know that the following queries are the five most common queries in the workload for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and offices of professors of a user-specified sex (male or female) who have a user-specified research specialty (e.g., *recursive query processing*). Assume that the university has a diverse set of faculty members, making it very uncommon for more than a few professors to have the same research specialty.
- List all the department information for departments with professors in a user-specified age range.
- List the department id, department name, and chairperson name for departments with a user-specified number of majors.
- List the lowest budget for a department in the university.
- List all the information about professors who are department chairpersons.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the university database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS and that both single- and multiple-attribute index search keys are permitted.

1. Specify your physical design by identifying the attributes you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Assume that this workload is to be tuned with an automatic index tuning wizard. Outline the main steps in the algorithm and the set of candidate configurations considered.
3. Redesign the physical schema, assuming that the set of important queries is changed to be the following:
   - List the number of different specialties covered by professors in each department, by department.
   - Find the department with the fewest majors.
   - Find the youngest professor who is a department chairperson.

Answer 20.2 The answer to each question is given below.

1. - We should create an unclustered hash index on $\langle specialty, sex \rangle$ on the Prof relation. This will enable us to efficiently find professors of a given specialty and sex for the first query. It is likely that just having the index on *specialty* would be enough since there are only two sexes. This may in fact be better since the index is smaller. (On the other hand, it is unlikely that *sex* will be updated often).
   - We should create a dense clustered B+ tree index on $\langle age, dept\_did \rangle$ on the Prof relation along with an unclustered hash index on *did* in the department relation. We can then find the department with professors in a specified age range efficiently with an index only search and then hash into the Dept relation to get the information we need for the second query.
   - We should create an unclustered hash index on *nummajors* in the Dept relation, in order to efficiently find those departments with a given number of majors for the third query.
   - We should create a dense clustered B+ tree index on *budget* in the Dept relation so we can efficently find the department with the smallest budget for the fourth query.
   - We should create a dense unclustered B+ tree index on *chair_ssno* for the Dept relation along with a dense unclustered hash index on *ssno* for the Prof relation so we can find the *ssno* of all chairpersons and then find information about them efficiently by doing an equality search on *ssno* on Prof. The scan on Dept can be made index only for increased efficiency.
2. - For the first query, we should create a dense clustered B+ tree index on $\langle dept\_did, specialty \rangle$ for the Prof relation. We can then use an index only scan to count the different specialties in each department.

- To find the department with the fewest majors (the second query), we should create an unclustered B+ tree index on *nummajors* for the Dept relation. We can then go down the tree to find the department with the fewest majors.
- To find the youngest professor that is a department chairperson we must create an unclustered hash index on *ssno* in the Prof relation. We must also create an unclustered B+ tree index on *chair_ssno* in the Dept relation. We first do an index only scan to find all the chairpersons, and then hash into Prof to get his or her age. To find the smallest age, we can just keep a counter.

Exercise 20.3 Consider the following BCNF relational schema for a portion of a company database (type information is not relevant to this question and is omitted):

Project(*pno, proj_name, proj_base_dept, proj_mgr, topic, budget*)
Manager(*mid, mgr_name, mgr_dept, salary, age, sex*)

Note that each project is based in some department, each manager is employed in some department, and the manager of a project need not be employed in the same department (in which the project is based). Suppose you know that the following queries are the five most common queries in the workload for this university and all five are roughly equivalent in frequency and importance:

- List the names, ages, and salaries of managers of a user-specified sex (male or female) working in a given department. You can assume that, while there are many departments, each department contains very few project managers.
- List the names of all projects with managers whose ages are in a user-specified range (e.g., younger than 30).
- List the names of all departments such that a manager in this department manages a project based in this department.
- List the name of the project with the lowest budget.
- List the names of all managers in the same department as a given project.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the company database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS, and that both single- and multiple-attribute index keys are permitted.

1. Specify your physical design by identifying the attributes you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Assume that this workload is to be tuned with an automatic index tuning wizard. Outline the main steps in the algorithm and the set of candidate configurations considered.
3. Redesign the physical schema assuming the set of important queries is changed to be the following:
   - Find the total of the budgets for projects managed by each manager; that is, list *proj_mgr* and the total of the budgets of projects managed by that manager, for all values of *proj_mgr*.
   - Find the total of the budgets for projects managed by each manager but only for managers who are in a user-specified age range.
   - Find the number of male managers.
   - Find the average age of managers.

Answer 20.3 The answer to each question is given below.

1. - For the first query, we should create a dense unclustered hash index on *mgr_dept* for the Manager relation. We omit *sex* from the key in this index since it is not very selective; however, including it is probably not very expensive since this field is unlikely to be updated.
   - We should create a unclustered B+ tree index on $\langle age, mgr\_dept, mid\rangle$ for the Manager relation, and an unclustered hash index on $\langle proj\_base\_dept, proj\_mgr\rangle$ for the Project relation. We can do an index only scan to find managers whose age is in the specified range, and then hash into the Project relation to get the project names. If index only scans are not supported, the index on manager should be a clustered index on *age*.
   - For the third query we don't need a new index. We can scan all managers and use the hash index on $\langle proj\_base\_dept, proj\_mgr\rangle$ on the Project relation to check if $mgr\_dept = proj\_base\_dept$.
   - We can create an unclustered B+ tree index on *budget* in the Project relation and then go down the tree to find the lowest budget for the fourth query.
   - For the fifth query, we should create dense unclustered hash index on *pno* for the Project relation. We can can get the *proj_base_dept* of the project by using this index, and then use the hash index on *mgr_dept* to get the managers in this department. Note that an index on $\langle pno, proj\_base\_dept$ for Project would allow us to do an index only scan on Project. However, since there is exactly one base department for each project (*pno* is the key) this is not likely to be significantly faster. (It does save us one I/O per project.)

2. ■ For the first query, we should create an unclustered B+Tree index on $\langle proj\_mgr, budget\rangle$ for the Project relation. An index only scan can then be used to solve the query. If index only scans are not supported, a clustered index on $proj\_mgr$ would be best.
   ■ If we create a sparse clustered B+ tree index on $\langle age, mid\rangle$ for Manager, we can do an index only scan on this index to find the ids of managers in the given range. Then, we can use an index only scan of the B+Tree index on $\langle proj\_mgr, budget\rangle$ to compute the total of the budgets of the projects that each of these managers manages. If index only scans are not supported, the index on Manager should be a clustered B+ tree index on age.
   ■ An unclustered hash index on sex will divide the managers by sex and allow us to count the number that are male using an index only scan. If index only scans are not allowed, then no index will help us for the third query.
   ■ We should create an unclustered hash index on age for the fourth query. All we need to do is average the ages using an index-only scan. If index-only plans are not allowed no index will help us.

Exercise 20.4 The Globetrotters Club is organized into chapters. The president of a chapter can never serve as the president of any other chapter, and each chapter gives its president some salary. Chapters keep moving to new locations, and a new president is elected when (and only when) a chapter moves. This data is stored in a relation *G(C,S,L,P)*, where the attributes are chapters (C), salaries (S), locations (L), and presidents (P). Queries of the following form are frequently asked, and you *must* be able to answer them without computing a join: "Who was the president of chapter X when it was in location Y?"

1. List the FDs that are given to hold over G.
2. What are the candidate keys for relation G?
3. What normal form is the schema G in?
4. Design a good database schema for the club. (Remember that your design *must* satisfy the stated query requirement!)
5. What normal form is your good schema in? Give an example of a query that is likely to run slower on this schema than on the relation G.
6. Is there a lossless-join, dependency-preserving decomposition of G into BCNF?
7. Is there ever a good reason to accept something less than 3NF when designing a schema for a relational database? Use this example, if necessary adding further constraints, to illustrate your answer.

Answer 20.4 The answer to each question is given below.

1. The FDs that hold over G are CL → P, C → S, P → C
2. The candidate keys are PL and CL.
3. G is in 1NF; the second dependency violates 2NF.
4. A good database schema is as follows. G1(C,L,P) and G2(C,S) . The schema still is not in BCNF, but it is in 3NF. The size of the original relation has been reduced by taking the salary attribute to a new relation. (We cannot make it into a BCNF relation without putting the P and the C atributes in separate relations thus requiring a join for answering the query.)
5. The "good" schema is in 3NF but not BCNF. The query "Give the salary for the President P when he was in Location X" would run slower due to the extra join that is to be computed.
6. No, there is no lossless and dependency-preserving decomposition; consider the FDs CL → P and P → C to see why.
7. Yes. Suppose there is an important query that can be computed without a join using a non-3NF schema but requires a join when transformed to 3NF. Then it may be better to accept the redundancy, especially if the database is infrequently updated. In our example, if we wanted to find presidents of a given chapter in a given location, and to find the president's salary, we might prefer to use the (non-3NF) schema CSLP.

Exercise 20.5 Consider the following BCNF relation, which lists the ids, types (e.g., nuts or bolts), and costs of various parts, along with the number available or in stock:

Parts (*pid, pname, cost, num_avail*)

You are told that the following two queries are extremely important:

- Find the total number available by part type, for all types. (That is, the sum of the *num_avail* value of all nuts, the sum of the *num_avail* value of all bolts, and so forth)
- List the *pid*s of parts with the highest cost.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for the set of Parts records, and what indexes would you create?

2. Suppose your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization you chose for the Parts relation in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.

3. How would your answers to the two questions change, if at all, if your system did not support indexes with multiple-attribute search keys?

Answer 20.5 The answer to each question is given below.

1. A heap file structure could be used for the relation Parts. A dense unclustered B+Tree index on $\langle pname, num\_avail \rangle$ and a dense unclustered B+ Tree index on $\langle cost, pid \rangle$ can be created to efficiently answers the queries.

2. The problem could be that the optimizer may not be considering the index only plans that could be obtained using the previously described schema. So we can instead create clustered indexes on $\langle pid, cost \rangle$ and $\langle pname, num\_avail \rangle$. To do this we have to vertically partition the relation into two relations like Parts1(pid, cost) and Parts2(pid, pname, num_avail). (If the indexes themselves have not been implemented properly, then we can instead use sorted file organizations for these two split relations).

3. If the multi attribute keys are not allowed then we can have a clustered B+ Tree indexes on *cost* and on *pname* on the two relations.

Exercise 20.6 Consider the following BCNF relations, which describe employees and the departments they work in:

Emp (*eid*, *sal*, *did*)
Dept (*did*, *location*, *budget*)

You are told that the following queries are extremely important:

- Find the location where a user-specified employee works.
- Check whether the budget of a department is greater than the salary of each employee in that department.

1. Describe the physical design you would choose for this relation. That is, what kind of a file structure would you choose for these relations, and what indexes would you create?

2. Suppose that your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization that you chose for the relations in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.

3. Suppose that your database system has very inefficient implementations of index structures. What kind of a design would you try in this case?

Answer 20.6 The answer to each question is given below.

1. We can have a heap file organisation for the two relations. To speed up the queries we can go in for a unclustered hash index on *eid* on the Emp relation, a unclustered hash on *did* on the Dept relation and a dense unclustered B+ Tree index with search key $\langle did, sal \rangle$ on the Emp relation.

2. The schema specified above involves searching in two indexes for each query. This is due to the fact that the attributes required for each query are present in both relations. One possible enhancement is to try denormalisation of the 2 relations. Thus we would have the relation Emp(*eid, sal, did, location*) and Dept (*did, budget*). We can then create unclustered hash index on Emp with key *eid* and another hash on Dept with key *did* and an unclustered B+ Tree index on $\langle did, sal \rangle$ on the Emp relation. If there is still some performance degradation we can merge the Dept relation with Emp relation and have a single hash on *eid* and a unclustered B+Tree on $\langle did, sal, budget \rangle$.

3. If the index implementations are not very good, then we can try sorted file organiztions. We can have a sorted file on Emp(*eid, did, location*) and Dept(*did, eid, sal, budget*)

Exercise 20.7 Consider the following BCNF relations, which describe departments in a company and employees:

Dept(*did, dname, location, managerid*)
Emp(*eid, sal*)

You are told that the following queries are extremely important:

- List the names and ids of managers for each department in a user-specified location, in alphabetical order by department name.
- Find the average salary of employees who manage departments in a user-specified location. You can assume that no one manages more than one department.

1. Describe the file structures and indexes that you would choose.
2. You subsequently realize that updates to these relations are frequent. Because indexes incur a high overhead, can you think of a way to improve performance on these queries without using indexes?

Answer 20.7 The answer to each question is given below.

1. A heap file organization for the two relations is sufficient if we create the following indexes. For the first, a clustered B+ tree index on $\langle location, dname \rangle$ would improve performance (we cannot list the names of the managers because there is no name attribute present). We can also have a hash index on *eid* on the Emp relation to speed up the second query: we find all of the *managerids* from the B+ tree index, and then use the hash index to find their salaries.
2. Without indexes, we can use horizontal decompostion of the Dept relation based on the location. We can also try sorted file organizations, with the relation Dept sorted on *dname* and Emp on *eid*.

Exercise 20.8 For each of the following queries, identify one possible reason why an optimizer might not find a good plan. Rewrite the query so that a good plan is likely to be found. Any available indexes or known constraints are listed before each query; assume that the relation schemas are consistent with the attributes referred to in the query.

1. An index is available on the *age* attribute:

```
SELECT E.dno
FROM   Employee E
WHERE  E.age=20 OR E.age=10
```

2. A B+ tree index is available on the *age* attribute:

```
SELECT E.dno
FROM   Employee E
WHERE  E.age<20 AND E.age>10
```

3. An index is available on the *age* attribute:

```
SELECT E.dno
FROM   Employee E
WHERE  2*E.age<20
```

4. No index is available:

```
SELECT DISTINCT *
FROM   Employee E
```

5. No index is available:

```
SELECT   AVG (E.sal)
FROM     Employee E
GROUP BY E.dno
HAVING   E.dno=22
```

6. The *sid* in Reserves is a foreign key that refers to Sailors:

```
SELECT   S.sid
FROM     Sailors S, Reserves R
WHERE    S.sid=R.sid
```

Answer 20.8 The answer to each question is given below.

1. The optimizer will not consider the index on age as it is misled by the `OR` predicate. To make it consider the index we can use `UNION` instead of an `OR` .

```
SELECT E.dno
FROM   Employee E
WHERE  E.age=20
UNION
SELECT E.dno
FROM   Employee E
WHERE  E.age=10
```

2. The optimizer might not consider the combined selectivity of the two expressions and so might not consider using the index on age. One way to make it use it is to use the `BETWEEN` predicate instead of the `AND` .

```
SELECT E.dno
FROM   Employee E
WHERE  E.age<20 BETWEEN E.age>10
```

3. Optimizers do not analyze the selectivity in mathematical expressions and may blindly use a filescan instead of the index on age. We can modify the SQL query as follows.

```
SELECT E.dno
FROM   Employee E
WHERE  E.age<10
```

4. Here the DISTINCT clause is used unnecesarily. When we consider the whole tuple they will always be distinct because each tuple contains a key. The optimizer would think of sorting the results obtained to eliminate duplicates.

```
SELECT *
FROM   Employee E
```

5. Here the use of GROUP BY predicate is unnecessary and would lead the optimizer into thinking of a plan for sorting the Employees relation on dno and then grouping them based on it.

```
SELECT   AVG (E.sal)
FROM     Employee E
WHERE    E.dno=22
```

6. Here the condition is that *sid* is a foreign key. Then it is obvious that *sid* should either be NULL or refer a tuple in Sailors. The optimizer would think up of a plan for joining the 2 relations, which is quite unnecessary.

```
SELECT   R.sid
FROM     Reserves R
WHERE    R.sid NOT NULL
```

Exercise 20.9 Consider two ways to compute the names of employees who earn more than $100,000 and whose age is equal to their manager's age. First, a nested query:

```
SELECT   E1.ename
FROM     Emp E1
WHERE    E1.sal > 100 AND E1.age = ( SELECT E2.age
                                     FROM   Emp E2, Dept D2
                                     WHERE  E1.dname = D2.dname
                                            AND D2.mgr = E2.ename )
```

Second, a query that uses a view definition:

```
SELECT   E1.ename
FROM     Emp E1, MgrAge A
WHERE    E1.dname = A.dname AND E1.sal > 100 AND E1.age = A.age

CREATE VIEW MgrAge (dname, age)
       AS SELECT D.dname, E.age
          FROM   Emp E, Dept D
          WHERE  D.mgr = E.ename
```

1. Describe a situation in which the first query is likely to outperform the second query.
2. Describe a situation in which the second query is likely to outperform the first query.
3. Can you construct an equivalent query that is likely to beat both these queries when every employee who earns more than $100,000 is either 35 or 40 years old? Explain briefly.

Answer 20.9 1. Consider the case when there are very few or no employees having salary more than 100K. Then in the first query the nested part would not be computed (due to short circuit evaluation) whereas in the second query the join of Emp and MgrAge would be computed irrespective of the number of Employees with sal > 100K.

Also, if there is an index on *dname*, then the nested portion of of the first query will be efficient. However, the index does not affect the view in the second query since it is used from a view.

2. In the case when there are a large number of employees with sal > 100K and the Dept relation is large, in the first query the join of Dept and Emp would be computed for each tuple in Emp that satisfies the condition E1.sal > 100K, whereas in the latter the join is computed only once.

3. In this case the selectivity of age may be very high. So if we have a B+ Tree index on $\langle age, sal \rangle$, then the following query may perform better.

```
SELECT   E1.ename
FROM     Emp E1
WHERE    E1.age=35 AND E1.sal > 100 AND E1.age =
         ( SELECT E2.age
         FROM     Emp E2, Dept D2
         WHERE    E1.dname = D2.dname AND D2.mgr = E2.ename)
UNION
SELECT   E1.ename
FROM     Emp E1
WHERE    E1.age = 40 AND E1.sal > 100 AND E1.age =
         ( SELECT E2.age
         FROM     Emp E2, Dept D2
         WHERE    E1.dname = D2.dname AND D2.mgr = E2.ename)
```

21

SECURITY

Exercise 21.1 Briefly answer the following questions:

1. Explain the intuition behind the two rules in the Bell-LaPadula model for mandatory access control.
2. Give an example of how covert channels can be used to defeat the Bell-LaPadula model.
3. Give an example of polyinstantiation.
4. Describe a scenario in which mandatory access controls prevent a breach of security that cannot be prevented through discretionary controls.
5. Describe a scenario in which discretionary access controls are required to enforce a security policy that cannot be enforced using only mandatory controls.
6. If a DBMS already supports discretionary and mandatory access controls, is there a need for encryption?
7. Explain the need for each of the following limits in a statistical database system:
   (a) A maximum on the number of queries a user can pose.
   (b) A minimum on the number of tuples involved in answering a query.
   (c) A maximum on the intersection of two queries (i.e., on the number of tuples that both queries examine).
8. Explain the use of an audit trail, with special reference to a statistical database system.
9. What is the role of the DBA with respect to security?
10. Describe AES and its relationship to DES.

11. What is public-key encryption? How does it differ from the encryption approach taken in the Data Encryption Standard (DES), and in what ways is it better than DES?

12. Explain how a company offering services on the Internet could use encryption-based techniques to make its order-entry process secure. Discuss the role of DES, AES, SSL, SET, and digital signatures. Search the Web to find out more about related techniques such as *electronic cash*.

Answer 21.1 The answer to each question is given below.

1. The *Simple Security Property* states that subjects can only interact with objects with a lesser or equal security class. This ensures subjects with low security classes from accessing high security objects. The **-Property* states that subjects can only create objects with a greater or equal security class. This prevents a high security subject from mistakenly creating an object with a low security class (which low security subjects could then access!).

2. One example of a covert channel is in statistical databases. If a malicious subject wants to find the salary of a new employee, and can issue queries to find the average salary in a department, and the total number of current employees in the depatment, then the malicious subject can calculate the new employees salary based on the increase in average salary and number of employees.

3. Say relation R contains the following values:

cid	*carname*	Security Class
1	Honda	U
1	Porsche	C
2	Toyota	C
3	Mazda	C
3	Ferrari	TS

Then subjects with security class U will see R as:

cid	*carname*	Security Class
1	Honda	U

Subjects with security class C will see R as:

cid	*carname*	Security Class
1	Honda	U
1	Porsche	C
2	Toyota	C
3	Mazda	C

Subjects with security class TS will see R as:

cid	*carname*	Security Class
1	Honda	U
1	Porsche	C
2	Toyota	C
3	Mazda	C
3	Ferrari	TS

4. Trojan horse tables are an example where discretionary access controls are not sufficient. If a malicious user creates a table and has access to the source code of some other user with privileges to other tables, then the malicious user can modify the source code to copy tuples from privileged tables to his or her non-privileged table.

5. Manditory access controls do not distinguish between people in the same clearance level so it is not possible to limit permissions to certain users within the same clearance level. Also, it is not possible to give only insert or select privileges to different users in the same level: all users in the same clearance level have select, insert, delete and update privileges.

6. Yes, especially if the data is transmitted over a network in a distributed environment. In these cases it is important to encrypt the data so people 'listening' on the wire cannot directly access the information.

7. (a) If a user can issue an unlimited number of queries, he or she can repeatedly decompose statistical information by gathering the statistics at each level (for example, at age ¿ 20, age ¿ 21, etc.).

   (b) If a malicious subject can query a database and retrieve single rows of statistical information, he or she may be able to isolate sensitive information such as maximum and minimum values.

   (c) Often the information from two queries can be combined to deduce or infer specific values. This is often the case with average and total aggregates. This can be prevented by restricting the tuple overlap between queries.

8. The *audit trail* is a log of updates with the authorization id of the user who issued the update. Since it is possible to infer information from statistical databases using repeated queries, or queries that target a common set of tuples, the DBA can use an audit trail to see which people issued these security-breaking queries.

9. The DBA creates new accounts, ensures that passwords are safe and changed often, assigns mandatory access control levels, and can analyze the audit trail to look for security breaches. They can also assist users with their discretionary permissions.

10. Public-key encryption is an encryption scheme that uses a public encryption key and a private decryption key. These keys are part of one-way functions whose inverse is very difficult to determine (which is why large prime numbers are involved in encryption algorithms...factoring is difficult!). The public key and private key are inverses which allow a user to encrypt any information, but only the person with the private key can decode the messages. DES has only one key and a specific decrypting algorithm. DES decoding can be more difficult and relies on only one key so both the sender and the receiver must know it.

11. A one-way function is a mathematical function whose inversese is very difficult to determine. These are used to determine the public and private keys, and to do the actual decoding: a message is encoding using the function and is decoded using the inverse of the function. Since the inverse is difficult to find, the code can not be broken easily.

12. An internet server could issue each user a public key with which to encrypt his or her data and send it back to the server (which holds all of the private keys). This way users cannot decode other users' messages, and even knowledge of the public key is not sufficient to decode the message. With DES, the encryption key is used both in encryption and decryption so sending keys to users is risky (anyone who intercepts the key can potentially decode the message).

Exercise 21.2 You are the DBA for the VeryFine Toy Company and create a relation called Employees with fields *ename*, *dept*, and *salary*. For authorization reasons, you also define views EmployeeNames (with *ename* as the only attribute) and DeptInfo with fields *dept* and *avgsalary*. The latter lists the average salary for each department.

1. Show the view definition statements for EmployeeNames and DeptInfo.

2. What privileges should be granted to a user who needs to know only average department salaries for the Toy and CS departments?

3. You want to authorize your secretary to fire people (you will probably tell him whom to fire, but you want to be able to delegate this task), to check on who is an employee, and to check on average department salaries. What privileges should you grant?

4. Continuing with the preceding scenario, you do not want your secretary to be able to look at the salaries of individuals. Does your answer to the previous question ensure this? Be specific: Can your secretary possibly find out salaries of *some* individuals (depending on the actual set of tuples), or can your secretary always find out the salary of any individual he wants to?

5. You want to give your secretary the authority to allow other people to read the EmployeeNames view. Show the appropriate command.

6. Your secretary defines two new views using the EmployeeNames view. The first is called AtoRNames and simply selects names that begin with a letter in the range A to R. The second is called HowManyNames and counts the number of names. You are so pleased with this achievement that you decide to give your secretary the right to insert tuples into the EmployeeNames view. Show the appropriate command and describe what privileges your secretary has after this command is executed.

7. Your secretary allows Todd to read the EmployeeNames relation and later quits. You then revoke the secretary's privileges. What happens to Todd's privileges?

8. Give an example of a view update on the preceding schema that cannot be implemented through updates to Employees.

9. You decide to go on an extended vacation, and to make sure that emergencies can be handled, you want to authorize your boss Joe to read and modify the Employees relation and the EmployeeNames relation (and Joe must be able to delegate authority, of course, since he is too far up the management hierarchy to actually do any work). Show the appropriate SQL statements. Can Joe read the DeptInfo view?

10. After returning from your (wonderful) vacation, you see a note from Joe, indicating that he authorized his secretary Mike to read the Employees relation. You want to revoke Mike's `SELECT` privilege on Employees, but you do not want to revoke the rights you gave to Joe, even temporarily. Can you do this in SQL?

11. Later you realize that Joe has been quite busy. He has defined a view called AllNames using the view EmployeeNames, defined another relation called StaffNames that he has access to (but you cannot access), and given his secretary Mike the right to read from the AllNames view. Mike has passed this right on to his friend Susan. You decide that, even at the cost of annoying Joe by revoking some of his privileges, you simply have to take away Mike and Susan's rights to see your data. What `REVOKE` statement would you execute? What rights does Joe have on Employees after this statement is executed? What views are dropped as a consequence?

Answer 21.2

1. EmployeeNames and DeptInfo are defined below:

```
CREATE VIEW EmployeeNames (ename)
       AS SELECT E.ename
          FROM   Employees E
```

```
CREATE VIEW DeptInfo (dept, avgsalary)
       AS SELECT   DISTINCT E.dept, AVG (E.salary) AS avgsalary
          FROM     Employees E
          GROUP BY E.dept
```

2. `SELECT` privilege on the `VIEW` DeptInfo.

   Note that it is impossible to allow the user to access only the average salaries of 'Toy' and 'CS' departments but not those of the other departments. If we really want the average salaries of other departments to be hidden from this user, we have no choice but to create another view.

3. a) `DELETE` on Employees
   b) `SELECT` on EmployeeNames
   c) `SELECT` on DeptInfo

4. No it does not ensure that. It is not possible for the secretary to find out the salary of any employee using just the relations alone.
   If the tuples are such that there is just one employee in a department and the secretary knows this information along with the name of the employee who works there, then he can possibly find out the salary. However, the relations themselves do not allow the secretary to deduce such a fact.

5. `GRANT SELECT ON` Employees `TO` Secretary `WITH GRANT OPTION`

6. `GRANT INSERT ON` Employees `TO` Secretary
   The secretary can now also insert tuples into AtoRNames, which is an updatable view created by the secretary. However, the secretary still cannot insert tuples into HowManyNames because this view is not updatable.

7. Todd's privileges are also revoked.

8. One example of a view update that cannot be implemented through updates to Employees is changing the average salary for a department since one doesn't know which salaries to change.

9. `GRANT SELECT, INSERT, UPDATE ON` Employees `TO` Joe `WITH GRANT OPTION`
   `GRANT SELECT, INSERT, UPDATE ON` EmployeeNames `TO` Joe `WITH GRANT OPTION`
   Joe cannot read the DeptInfo view, but could an identical view.

10. There is no way to do this in SQL: even though you granted privileges to Joe and Joe granted privileges to Mike, you cannot revoke Joe's privileges without also revoking Mike's.

11. Since you don't own AllNames, you can only prevent Mike and Susan from accessing it by revoking Joe's right to read EmployeeNames:

`REVOKE SELECT ON` EmployeeNames `FROM` Joe
The view AllNames is dropped as a consequence. Joe can still modify EmployeeNames without reading it.

Exercise 21.3 You are a painter and have an Internet store where you sell your paintings directly to the public. You would like customers to pay for their purchases with credit cards, and wish to ensure that these electronic transactions are secure.

Assume that Mary wants to purchase your recent painting of the Cornell Uris Library. Answer the following questions.

1. How can you ensure that the user who is purchasing the painting is really Mary?
2. Explain how SSL ensures that the communication of the credit card number is secure. What is the role of a certification authority in this case?
3. Assume that you would like Mary to be able to verify that all your email messages are really sent from you. How can you authenticate your messages without encrypting the actual text?
4. Assume that your customers can also negotiate the price of certain paintings and assume that Mary wants to negotiate the price of your painting of the Madison Terrace. You would like the text of this communication to be private between you and Mary. Explain the advantages and disadvantages of different methods of encrypting your communication with Mary.

Answer 21.3 The answer to each question is given below.

1. In order to determine whether the user who is purchasing the painting is really Mary, we need some level of verification when Mary first registers with the system. On the lowest level, we can simply ask the user to confirm things like Mary's address or social security number. To increase the level of security, we could also ask the user to verify Mary's credit card number. Since these numbers are deemed difficult to obtain, most merchant websites consider this sufficient evidence for proof of identity.

   For an even higher level of security, we can take external steps to verify Mary's information such as calling her up with the phone number provided, sending a letter to Mary's mailing address, or sending her an e-mail with instructions to reply back. In each instance, we attempt to validate the information the user provided so that the element of uncertainty in the provided information is decreased.

2. SSL Encryption is a form of public-key encryption where a third party certification authority acts to validate public keys between two clients. In a general public-key

encryption system, data is sent to a user encrypted with a publicly known key for that user, such that only the user's private key, known only to that user, can be used to decrypt the data. Attempts to decrypt the information using other keys will produce garbage data, and the ability to decipher the private key is considered computationally expensive even for the most modern computing systems.

In SSL Encryption, the client transmitting the data asks the certification authority for a certificate containing public key information about the other client. The first client then validates this information by decrypting the certificate to get the second client's public key. If the decrypted certificate matches up with the certification authority's information, the first client then uses this public key to encrypt a randomly generated session key and sends it to the second client. The first client and second client now have a randomly generated public-key system that they can use to communicate privately without fear that anyone else can decode their information.

Once complete, SSL encryption ensures that data such as credit card information transmitted between the two clients cannot be easily decrypted by others intercepting packets because the certification authority helped to generate a randomly created public-key that only the two clients involved can understand.

3. A message to Mary can be sent unencrypted with a message signature attached to the message. A signature is obtained by applying a one-way function to the message and is considerably smaller than the message itself. Mary can then apply the one-way function and if the results of it match the signature, she'll know it was authentic.

4. One method of sending Mary a message is to create a digital signature for the message by encrypting it twice. First, we encrypt the message using our private key, then we encrypt the results using Mary's public key. The first encryption ensures that the message did indeed come from us, since only we know our private key while the second encryption ensures that only Mary can read the message, since only Mary knows her private key.

   This system is very safe from tampering since it is hard to send a message pretending to be someone else as well as difficult to properly decode an intercepted message. The only disadvantages are that it requires that we have copies of each person's public key as well as spend the time to encrypt/decrypt the messages. For example, if Mary receives the message on her laptop or PDA while traveling, she may not have the resources or public keys to decrypt it and respond, and might need to wait until she returns to the office.

   Another method of communicating with Mary, discussed in the previous question,

is to use message signatures. This allows Mary to be able to read the message from almost any system since it is not encrypted and ensure that the message is authentic. The only disadvantage is that it does not safely prevent someone else from reading the message as well.

Exercise 21.4 Consider Exercises 6.6 to 6.9 from Chapter 6. For each exercise, identify what data should be accessible to different groups of users, and write the SQL statements to enforce these access control policies.

Answer 21.4 Please note that there is an impedance mismatch between the level of generality provided by SQL security and the level of granularity provided by most applications. For example, in SQL if we grant a user 'Bob' access to his records in the Customer table, we also give him access to everyone else's records in the Customer table. SQL cannot grant role level security without creating a View for *every user in the database* which is a *very* imcorrect model to use.

Therefore in most current applications, the role level security is reimplemented at the application server level. The following solutions reflect reasonable choices a DBA would make in order to limit some of the database information to entire classes of users.

- *Exercise 6.6*
  Using the Schema defined in the Solution Guide to Exercise 6.6 we define the follow security modifications to the database. In addition, the class of users for the website is denoted GeneralUser.
  - In order to search the database for a record, web users need read access to the record information tables.
    ```
    GRANT SELECT ON Album TO GeneralUser
    GRANT SELECT ON Songs TO GeneralUser
    GRANT SELECT ON Musicians TO GeneralUser
    GRANT SELECT ON Performs TO GeneralUser
    ```
  - To register and login, web users need the ability to insert records into the Users table, as well as read select data from it.
    ```
    CREATE VIEW UserLogin(userid,password)
           AS SELECT U.userid, U.password
              FROM Users U

    GRANT SELECT ON UserLogin TO GeneralUser
    GRANT INSERT ON Users TO GeneralUser
    ```
  - For the shopping basket, no database security is neccessary since this information is usually handled by the application server as temporary state data, i.e., changes are never saved to the database.

- To checkout, web users need the ability to create new orders.
  `GRANT INSERT ON` Orders `TO` GeneralUser

- *Exercise 6.7*
  Using the Schema defined in the Solution Guide to Exercise 6.7 we define the follow security modifications to the database. In addition, the class of users for the doctors is denoted DoctorUser and the class of users for the patients is denoted PatientUser. Both classes are members of the AllUsers Class and inherit all rights from it.

  - In order to access the website and lookup information about drugs, all the users need read access to the pharmacy/drug information tables.
    `GRANT SELECT ON` Drug `TO` AllUsers
    `GRANT SELECT ON` Pharmacy `TO` AllUsers
    `GRANT SELECT ON` Pham_co `TO` AllUsers
    `GRANT SELECT ON` Contract `TO` AllUsers
    `GRANT SELECT ON` Sell `TO` AllUsers
    `GRANT SELECT ON` Drug `TO` AllUsers
  - To create, modify, and view prescriptions, doctors need special access to the prescriptions table as well as the ability to lookup their patient's names so as to assign prescriptions properly.

    ```
    CREATE VIEW PatientName(ssn,name)
           AS SELECT P.ssn, P.name
              FROM Patient P

    GRANT SELECT ON PatientName TO DoctorUser
    GRANT INSERT, UPDATE, SELECT ON Prescription TO DoctorUser
    ```
  - Patients need the ability to change their primary physician, and in order to do so they need to be able to select from a list of doctor names.

    ```
    CREATE VIEW DoctorName(ssn,name)
           AS SELECT D.ssn, D.name
              FROM Patient D

    GRANT SELECT ON DoctorName TO PatientUser
    GRANT UPDATE (phy_snn) ON Pri_Phy_Patient TO PatientUser
    ```
  - To view the status of a prescription, patients need partial access to the Prescription table.

    ```
    CREATE VIEW PrescriptionStatus(phy_ssn,date,quatity,tradename,pharmacyname)
           AS SELECT P.phy_ssn, P.date, P.quantity, P.tradename, C.pharmacyname
              FROM Prescription P, Pharmacy_Co C
              WHERE (pssn=P.ssn)

    GRANT SELECT ON PrescriptionStatus TO PatientUser
    ```
  - To checkout, patients need the ability to create new orders.

```
CREATE VIEW PatientAddress(ssn,address)
       AS SELECT U.ssn,U.address
          FROM Users U

GRANT SELECT ON PatientAddress TO PatientUser
GRANT INSERT ON Orders TO PatientUser
```

- *Exercise 6.8*
  Using the Schema defined in the Solution Guide to Exercise 6.8 we define the follow security modifications to the database. In addition, the class of users for the faculty is denoted FacultyUser and the class of users for the students is denoted StudentUser.

  – Faculty needs to be able can view/create/delete classes.
  `GRANT SELECT, INSERT, DELETE ON Class TO FacultyUser`

  – To enroll in classes, students need to be able to view class information and insert information into the enrollment table. The view is neccessary because you do not want students to have access to faculty id numbers of not neccessary.

  ```
  CREATE VIEW ClassInfo(name,meets_at,room,fname)
         AS SELECT C.name, C.meets_at, C.room, F.fname
            FROM Class C, Faculty F
            WHERE (C.fid = F.fid)

  GRANT SELECT ON ClassInfo TO StudentUser
  GRANT INSERT ON Enrolled TO StudentUser
  ```

- *Exercise 6.9*
  Using the Schema defined in the Solution Guide to Exercise 6.9 we define the follow security modifications to the database. In addition, the class of users for the employees is denoted EmployeeUser and the class of users for the Passengers is denoted PassengerUser. Both classes are members of the AllUsers Class and inherit all rights from it.

  – In order to access the website and lookup information about flights, all the users need read access to the flight and aircraft information tables.
  `GRANT SELECT ON Flights TO AllUsers`
  `GRANT SELECT ON Aircraft TO AllUsers`
  `GRANT SELECT ON Certified TO AllUsers`

  – Employees need to be able to add new flights and delete old ones.
  `GRANT INSERT, DELETE ON Flights TO EmployeeUser`

  – Passengers need to be able to make reservations on flights.
  `GRANT INSERT ON Reservation TO PassangerUser`

Exercise 21.5 Consider Exercises 7.7 to 7.9 from Chapter 7. For each exercise, discuss where encryption, SSL, and digital signatures are appropriate.

Answer 21.5 The answer to each question is given below.

- *Exercise 7.7*
  For the Notown Records website, encryption plays an important part in ensuring that customers are able to safely interact and order records over the Internet. Before discussing what should be encrypted, it is important to also note what should not be encrypted. Many of operations including searching the database and browsing record catalogs do not require any encryption. These operations are performed often and encrypting every communication from the client to the website would severely drain the server's resources. As a result, it is better for the server to focus on encrypting only information that is of a more serious nature.

  There are three places where we can apply an encryption scheme to this system: user registration, user login, and user checkout. The purpose of encrypting data at registration and checkout is obvious, the user is transmitting sensitive personal information like address and credit card numbers, and it is of utmost importance to protect this information. We also encrypt the password transmitted during user login since that ensures that future communications with the user are safe after login.

  In practice, we would use SSL encryption via a certification authority, e.g., Verisign. Upon an encryption request, the client's web browser requests the Verisign certificate, validates it by decrypting it, and uses the public key from the decrypted certificate to encrypt a radomly generated session key that it then sends to the server. Using this session key, the certification authority is no longer needed, and the server/client then transmit information just as they would in a normal public key system.

  The Notown website could also use digital signatures to verify a user's registration information via e-mail, as well as send an order confirmation e-mail after an order has been placed. By checking the digital signature of e-mails sent, the website and user can help to double check each other's identities after a new registration has been processed or a transaction has been placed.

- *Exercise 7.8*
  For this question, security is *much* more important than in the previous question, since medical records are considered very serious. As such, we would want to encrypt most of the information transmitted during a doctor or patient's visit to the website. The only information we do not need to transit encrypted would be pages with read-only data, e.g., company information or help pages. Although

this puts a higher strain on the system, the demand for security from users in the medical field is much higher than that of users in the record sales industry.

As before, we would use an SSL encryption via a certification authorization to handle the actual encryption. When a new user registers, it would especially important to verify their identify as to prevent someone else from ordering prescriptions in their name. To this end, digital signatures would be much more important to verify the validity of the e-mail used to sign up with. In addition, external authorization including faxes, phone calls, and written mail should also be used to verify registration information when a user signs up.

- *Exercise 7.9*
  For the course enrollment system, security is important for many of the processes, but the system does not need to encrypt as much as it did in the online pharmacy system.

  For faculty members, their passwords should be encrypted when they login as well as their registration information when they sign up for the first time. When they create/delete existing courses, it is probably not necessary to encrypt their data so long as the system is sure the user is properly logged in. To this end, the system could ask the faculty to re-enter their passwords when they are about to submit a change to the system.

  For students, their login and registration should also be encrypted. Like the faculty system, it is probably not important to encrypt any of the information about what classes a student signs up for as long as the login information is accurate. Students would then be able to freely modify their schedule after logging in to the website and only be asked their password again when they were ready to submit their changes to the system.

  In both cases, SSL encryption would again be the method of choice for the actual encryption process. Digital signatures could be used in e-mails sent to the students confirming their course registration information.

- *Exercise 7.10*
  For the airline reservation website, encryption would again be important in user login and user registration information for both customers and employees. Other information like searching for flights should be left unencrypted as a customer may search for dozens of flights. When a customer purchases a flight, their order, including their flight information and credit card number, should be encrypted to prevent others from learning which flight they have selected, as well as protecting the customer's money. For airline employees, it is preferable to encrypt all of the transmitted data so as not to give hackers the opportunity to see any backend part of the system. Since it is likely there will be few employees per number of

clients, the drain on the system from the extra level of encryption for employees should be negligible.

Note that before when we considered the prescription website, we recommended encrypting all transmitted data, whereas when we considered the course enrollment system we recommended a looser form of encryption on both classes of users. By looser form of encryption, we refer to the fact that some of the transmitted data is encrypted while some data is not. Contrast this with the airline reservation system, where we suggest loose encryption for customers and total encryption for employees. For each, keep in mind that the true level of encryption can change depending on the available resources and sensitivity of the data.

As before, SSL encryption is the preferred method of data encryption. For the airline reservation system, digital signatures can be applied to confirm when customers have places orders via e-mail.