DATABASE MANAGEMENT SYSTEMS SOLUTIONS MANUAL

Raghu Ramakrishnan
University of Wisconsin
Madison, WI, USA

Johannes Gehrke, Jeff Derstadt, 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

`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

OVERVIEW OF 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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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.

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 Answer omitted.

Exercise 2.3 Consider the following information about a university database:

- 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.

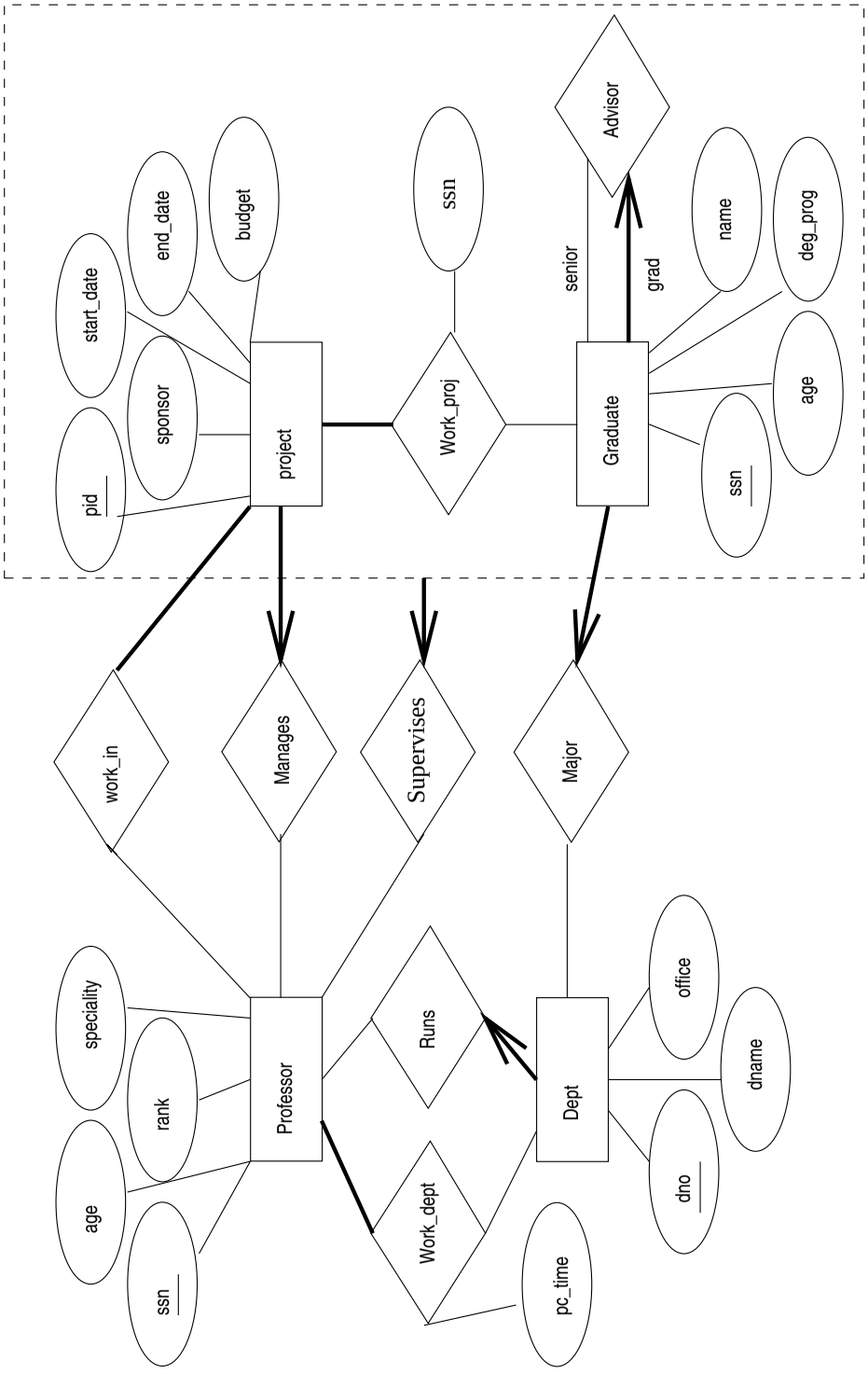


Figure 2.1 ER Diagram for Exercise 2.3

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 Answer omitted.

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.
- 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.

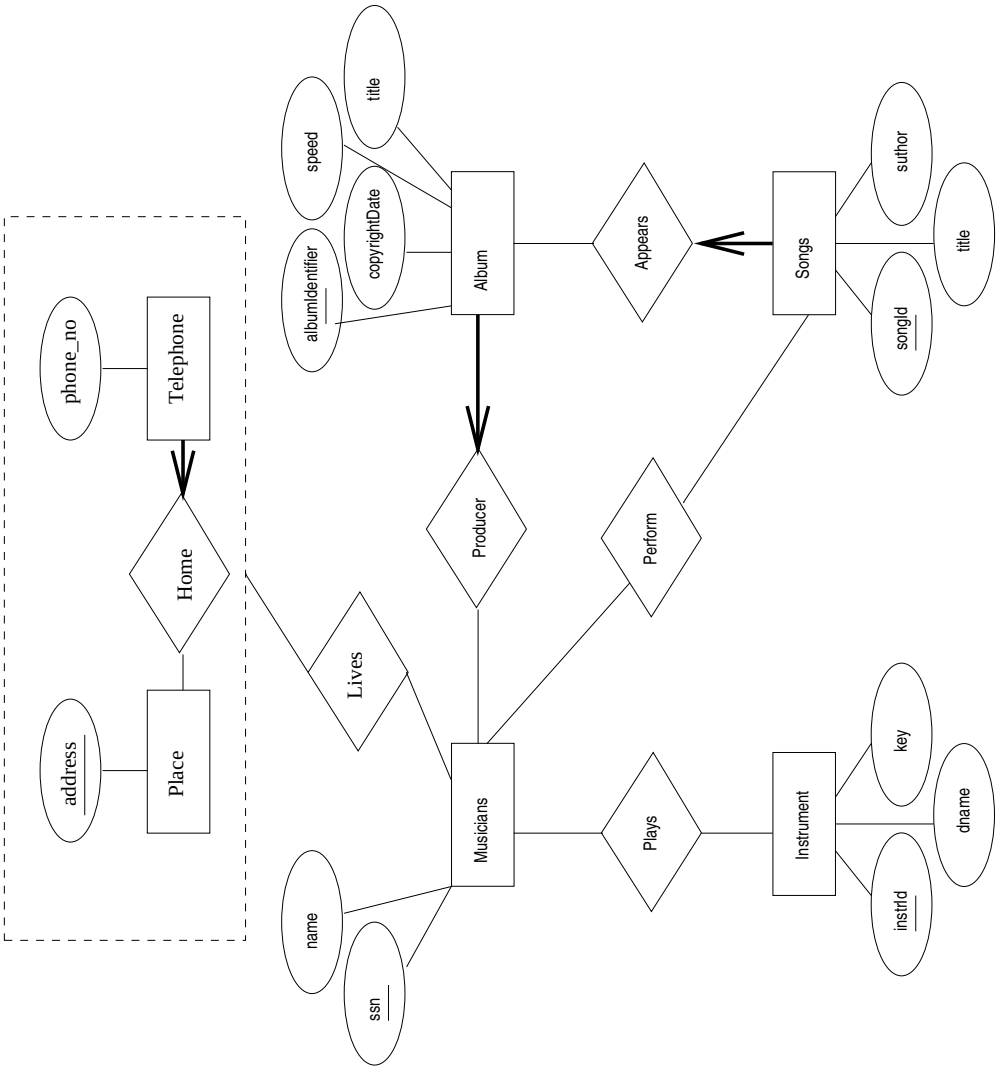


Figure 2.2 ER Diagram for Exercise 2.5

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.
- 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 Answer omitted.

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.
- 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.

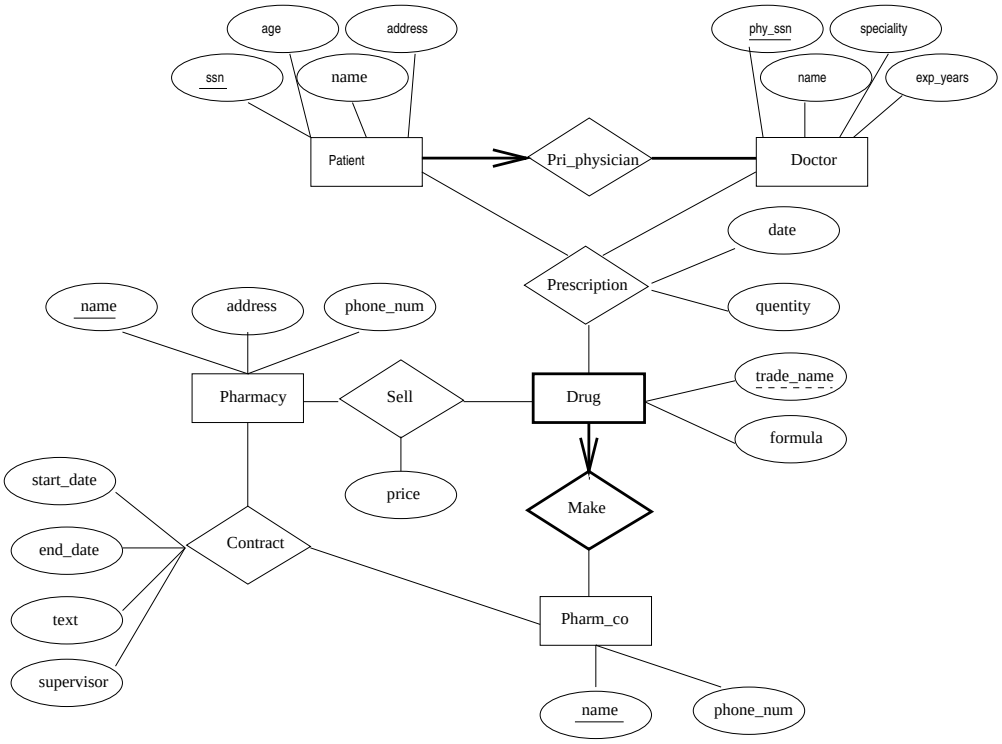


Figure 2.3 ER Diagram for Exercise 2.8

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 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 Answer omitted.

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?

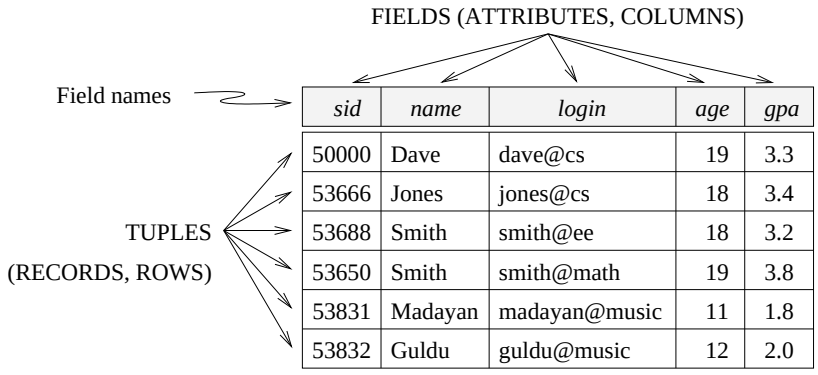
Answer 3.2 Answer omitted.

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 Answer omitted.



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 Answer omitted.

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

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 Answer omitted.

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 Answer omitted.

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 statement creates Table A:

```
CREATE TABLE A (  a1      CHAR(10),
                  a2      CHAR(10),
                  PRIMARY KEY (a1) )
```

Tables B and C are created similarly to A.

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

We cannot capture the total participation constraint of A in R. This is because we cannot ensure that every key a1 appears in R without the use of checks.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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
```

4. (a) If DInfo.manager is updated, it could, in principle, be implemented automatically by updating the Dept relation to reflect a change in the manager of department DInfo.did. However, since SQL/92 does not allow an update on a view definition based on more than one base relation, this view update is not allowed.

   (b) If DInfo.totsals is updated, this change cannot be implemented automatically at all because it is not clear which of the employees' salary fields need to be changed.

   (c) Views are an important component of the security mechanisms provided by a relational DBMS. By defining views on the base relations, we can present needed information to a user while *hiding* other information that perhaps the user should not be given access to.
   As an example the chairman of a company might want his secretary to be able to look at the total salaries given to a department under him, but not at the individual salaries of the employees working in those departments. This view definition would be useful in that case and provides a layer of security that prevents the secretary from viewing or changing the salaries of the employees.

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 Answer omitted.

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 Answer omitted.

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:

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

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

■ DRC

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

■ 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

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

- DRC

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

- 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

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

- DRC

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

- 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

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

- TRC

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

- DRC

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

- 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

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

■ 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 Answer omitted.

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. Print the level and the average age of students for that level, for each level.
8. Print the level and the average age of students for that level, for all levels except JR.
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.num 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 Answer omitted.

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 earn 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.aname
FROM   Employee E
WHERE  E.salary < ( SELECT MIN (F.price)
                    FROM   Flights F
                    WHERE  F.from = 'LA' 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, Employes 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 = 'Boeing'
```

6.
```
SELECT A.aid
FROM   Aircraft A
WHERE  A.cruisingrange > ( SELECT MIN (F.distance)
                           FROM   Flights F
                           WHERE  F.from = 'L.A.' 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.eid = C.aid) )
```

8.
```
SELECT DISTINCT E.ename
FROM   Employees E, Certified C, Aircraft A
WHERE  C.eid = E.eid
AND    C.aid = A.aid
AND    A.cruisingrange > 3000
AND    E.eid NOT IN ( SELECT C1.eid
       FROM Certified C1, Aircraft A1
       WHERE C1.aid = A1.aid
       AND A1.aname = '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 = 'NY' AND
                            AND F0.arrives < 1800 )
                   UNION
                   ( SELECT F0.flno
                     FROM   Flights F0, Flights F1
                     WHERE  F0.from = 'Madison' AND F0.to <> 'NY' AND
                     AND    F0.to = F1.from AND F1.to = 'NY'
                            F1.departs > F0.arrives AND
                            F1.arrives < 1800 )
                     UNION
                     (      SELECT F0.flno
                            FROM Flights F0, Flights F1, Flights F2
                            WHERE F0.from = 'Madison'
                            WHERE F0.to = F1.from
                            AND F1.to = F2.from
                            AND F2.to = 'NY'
                            AND F0.to <> 'NY'
                            AND F1.to <> 'NY'
                            AND F1.departs > F0.arrives
                            AND F2.departs > F1.arrives
                            AND F2.arrives < 1800 ))
```

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 *name*s of managers who manage the departments with the largest budgets.

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

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 Answer omitted.

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:
   - The *Sailors* relation is empty, or

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

(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

- 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:

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

6

DATABASE APPLICATION DEVELOPMENT

Answers not available yet.

7

INTERNET APPLICATIONS

Answers not available yet.

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 Not yet available.

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?

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*

7. Consider Alternatives (1), (2) and (3) for 'data entries' in an index, as discussed in Section XXX. Are all of them suitable for secondary indexes? Explain.

Answer 8.2 Answer omitted.

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. At this time,using index is 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).
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 Answer omitted.

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 Not yet available.

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 Answer omitted.

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 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.

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.

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 Answer omitted.

Exercise 8.9 What main conclusions can you draw from the discussion of the five basic file organizations discussed in Section **??**? 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 Answer omitted.

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).

3. We should create a hash index on the *eid* field of Emp (d) since then we can do a filescan on Dept and hash each manager id into Emp and check to see if the salary is greater than 12,000 dollars. None of the indexes offered lend themselves to index-only scans, so it doesn't matter if they are allowed or not.

4. For this query we should create a dense clustered B+ tree (c) index on $\langle did, sal \rangle$ fields of the Emp relation, so we can do an index only scan. (An unclustered index would be sufficient, but is not included in the list of index choices for this question.) If index-only scans are not allowed, we should then create a clustered sparse B+ tree index on the *did* field of Emp (a). This index will be just as efficient as an index on $\langle did, sal \rangle$ since both will involve accessing the data records in *did* order. However, since we now have only one attribute in the search key, it will be updated less often.

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 Answer omitted.

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 Answer omitted.

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? 51,200?
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 and 51200 are.
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

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

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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.

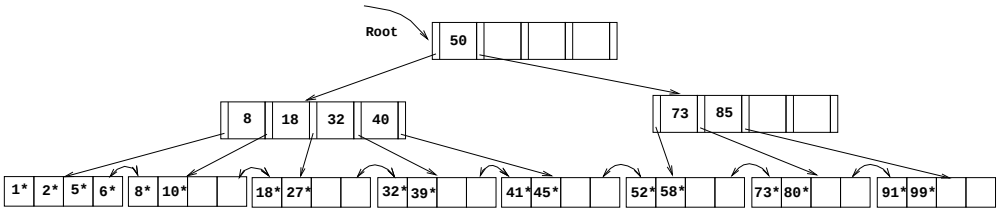


Figure 10.1 Tree for Exercise 10.1

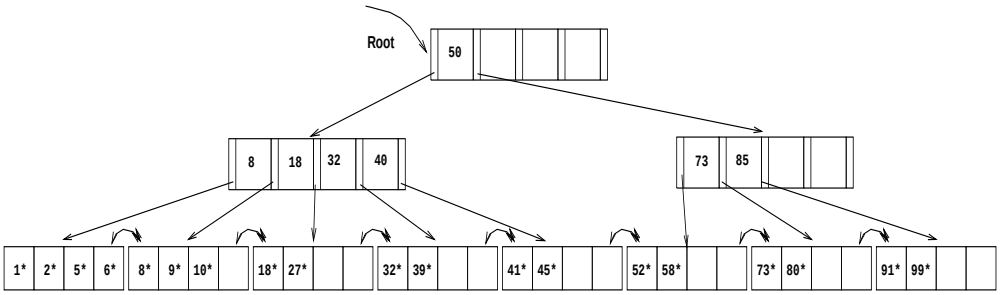


Figure 10.2

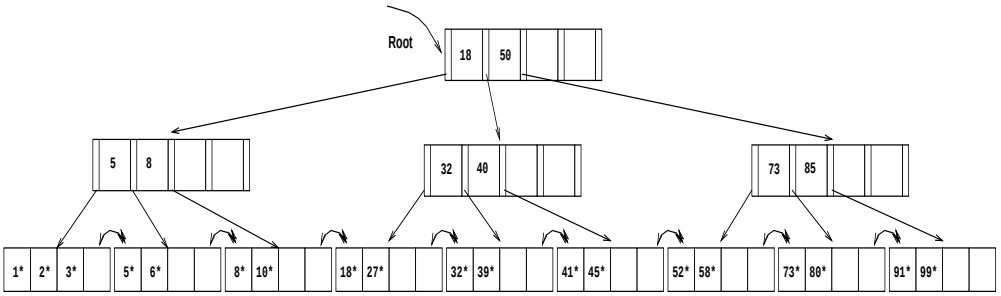


Figure 10.3

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

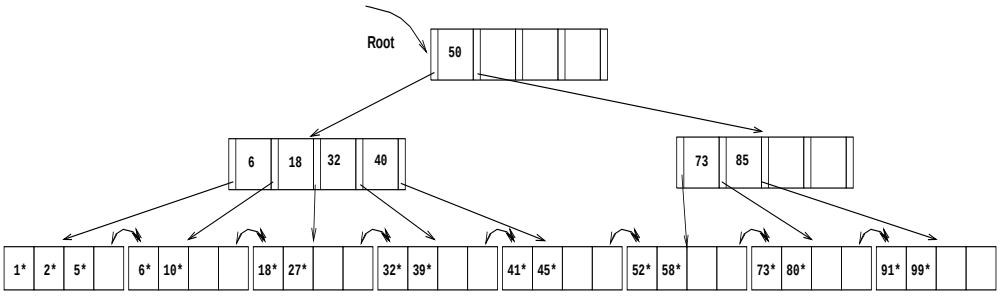


Figure 10.4

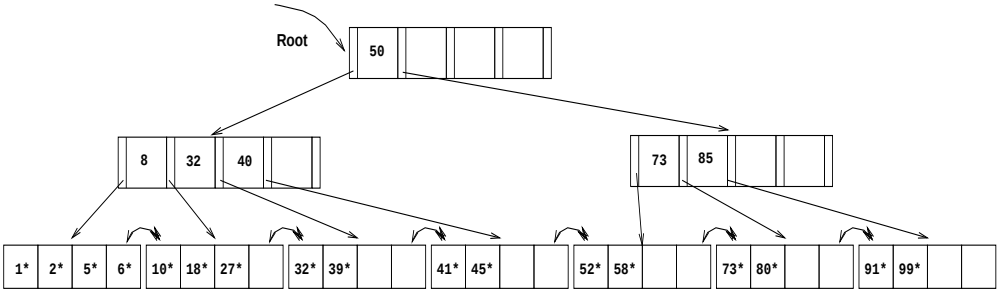


Figure 10.5

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.

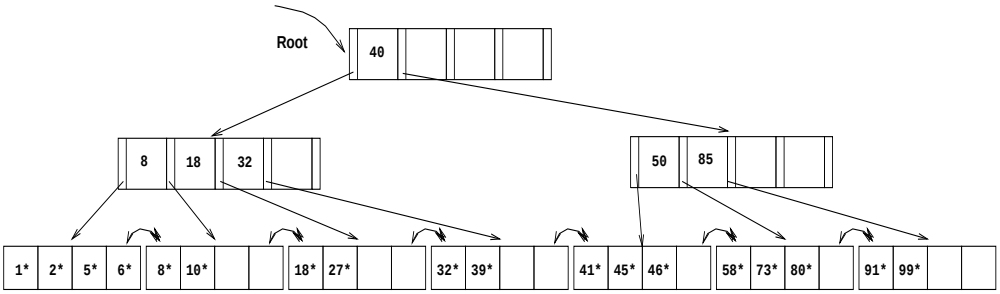


Figure 10.6

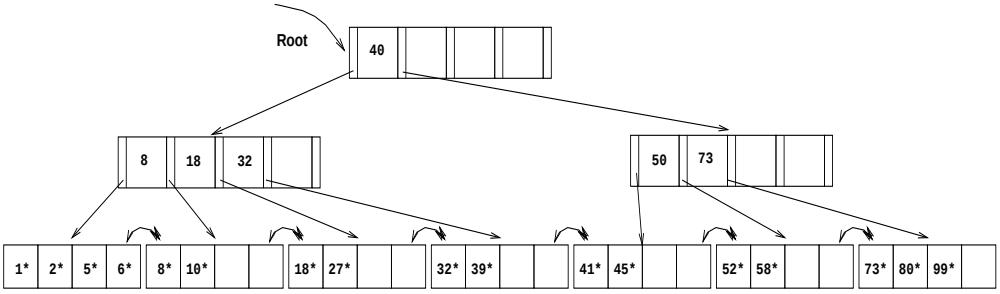


Figure 10.7

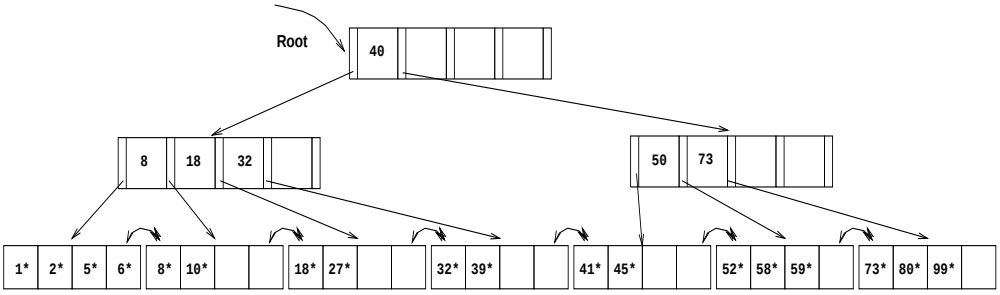


Figure 10.8

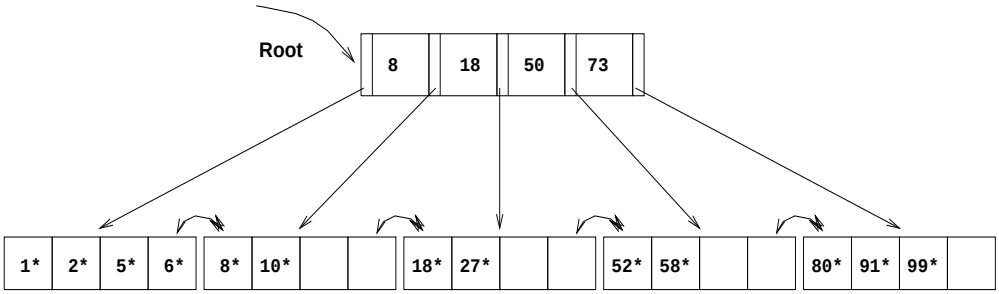


Figure 10.9

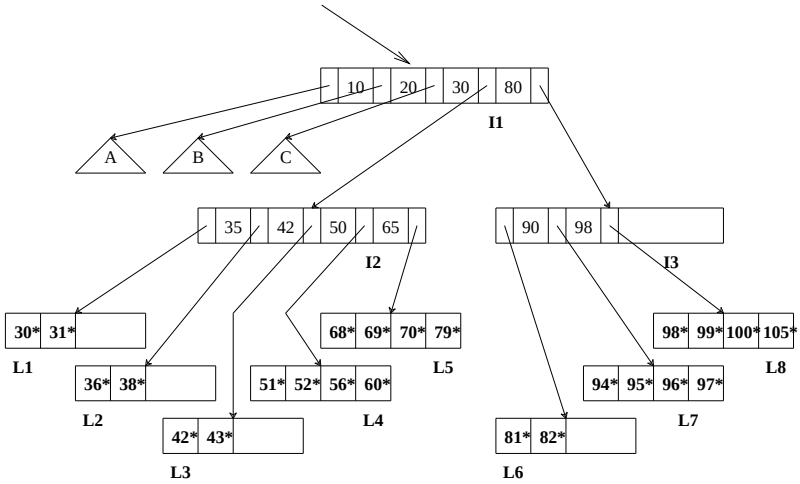


Figure 10.10 Tree for Exercise 10.2

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."

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 Answer omitted.

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 leave 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

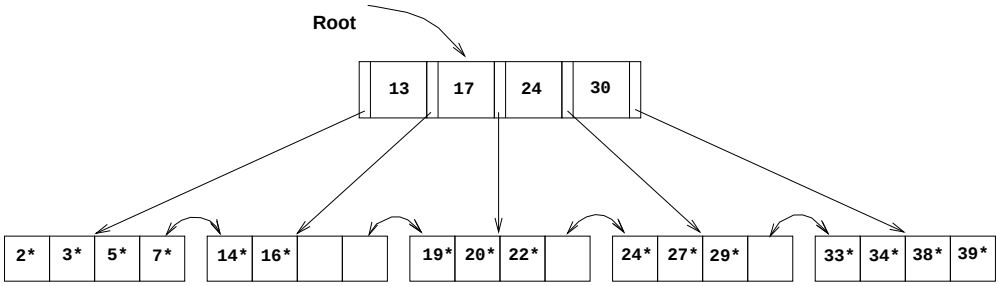


Figure 10.11 Tree for Exercise 10.5

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.
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 Answer omitted.

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

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

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 Answer omitted.

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 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.

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 10.12 An Instance of the Students Relation

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 Answer omitted.

Exercise 10.11 Suppose that duplicates are handled using the approach without overflow pages discussed in Section **??**. 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 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 Answer omitted.

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.

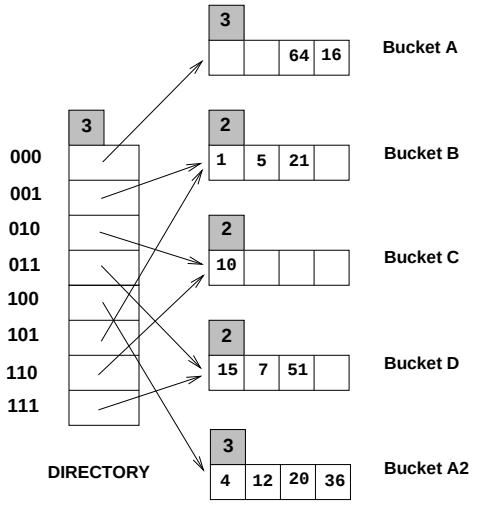


Figure 11.1 Figure for Exercise 11.1

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.

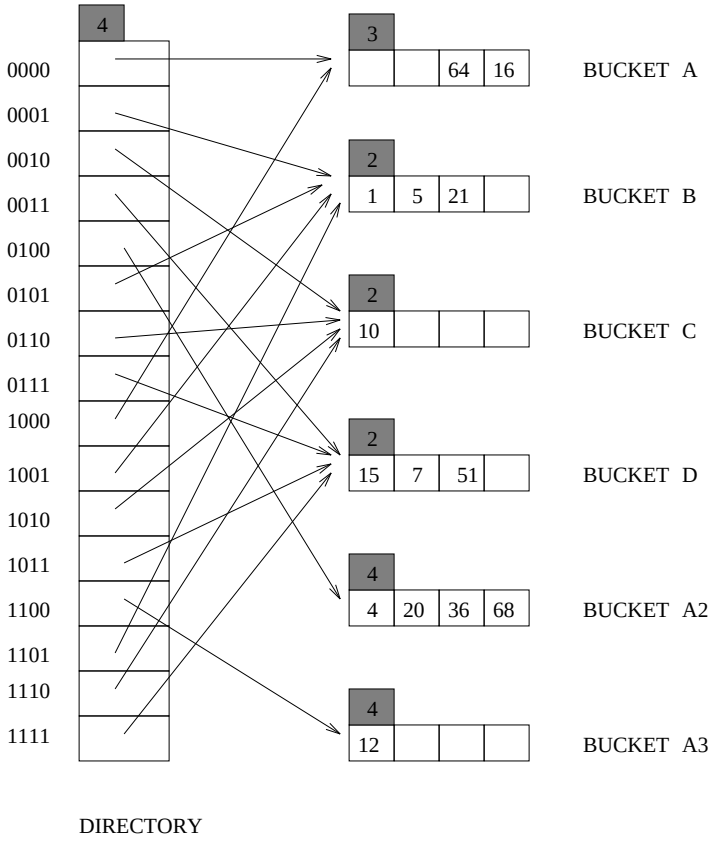

4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
DIRECTORY
3
64
16
BUCKET A
2
1
5
21
BUCKET B
2
10
BUCKET C
2
15
7
51
BUCKET D
4
4
20
36
68
BUCKET A2
4
12
BUCKET A3

Figure 11.2

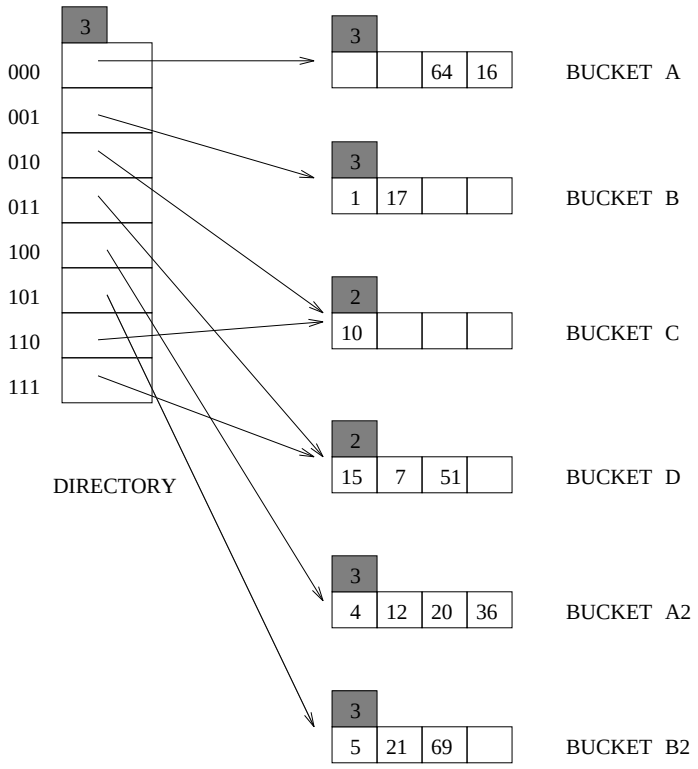

3
000
001
010
011
100
101
110
111
DIRECTORY
3
64
16
BUCKET A
3
1
17
BUCKET B
2
10
BUCKET C
2
15
7
51
BUCKET D
3
4
12
20
36
BUCKET A2
3
5
21
69
BUCKET B2

Figure 11.3

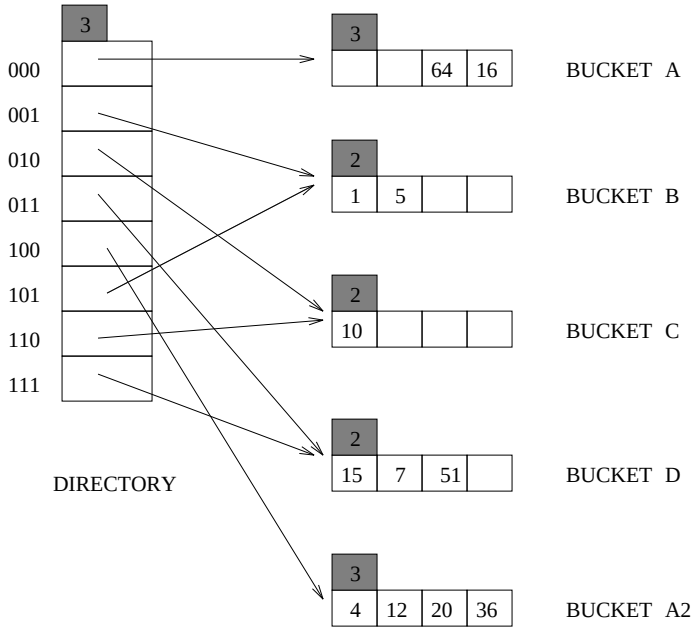


Figure 11.4

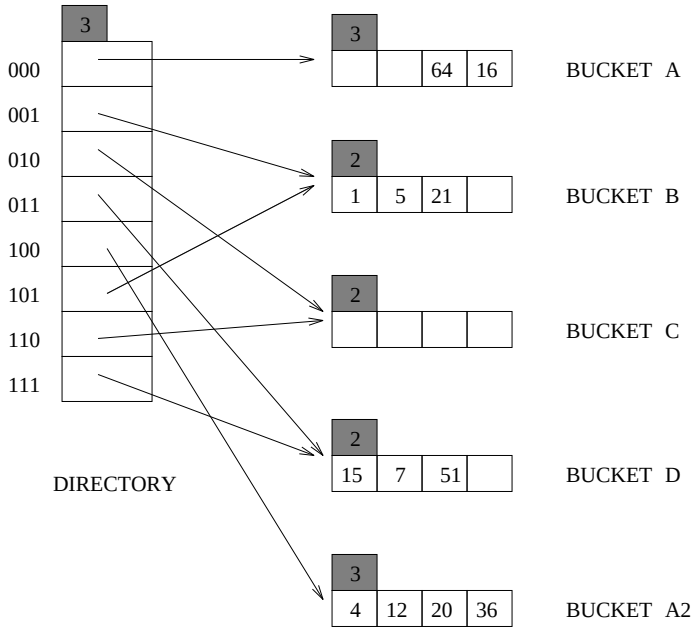


Figure 11.5

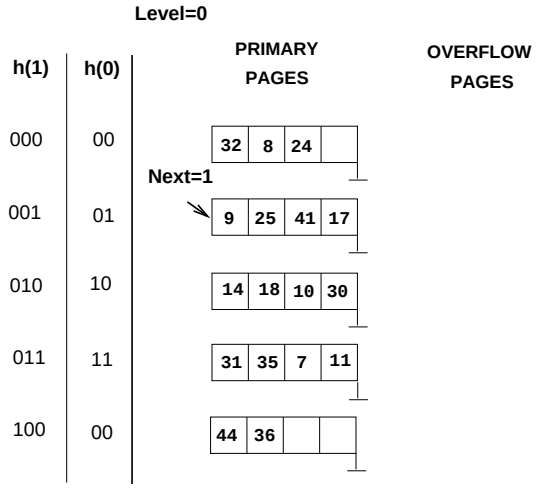


Figure 11.6 Figure for Exercise 11.2

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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:

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 Answer omitted.

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.
   (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?

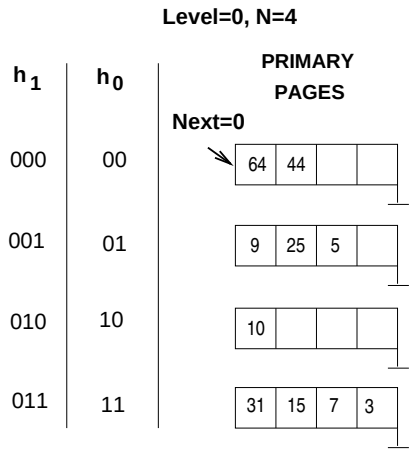


Figure 11.7 Figure for Exercise 11.9

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 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.

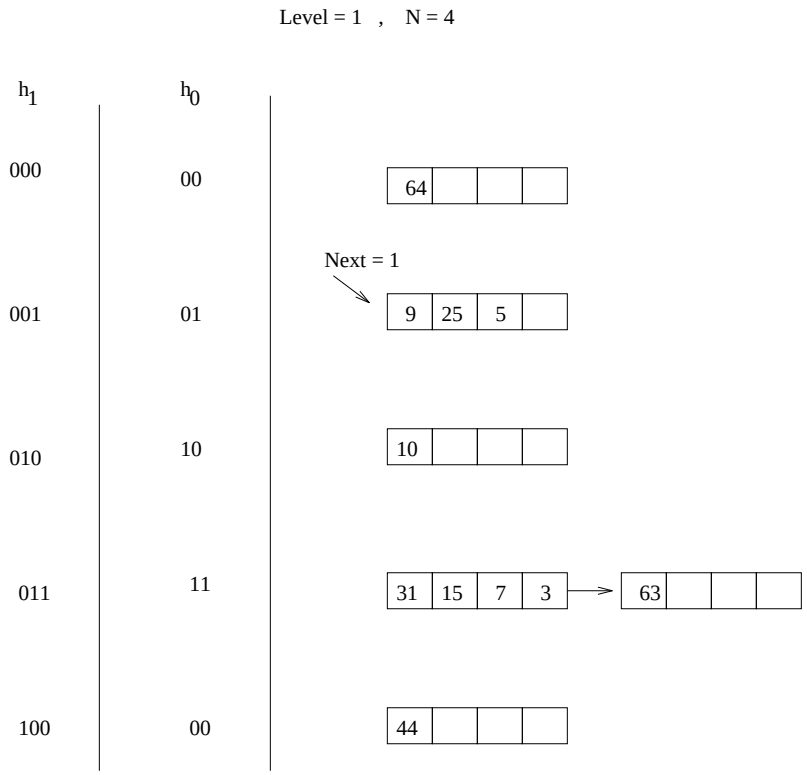

Level = 1 , N = 4
h1
h0
000
00
64
Next = 1
001
01
9
25
5
010
10
10
011
11
31
15
7
3
63
100
00
44

Figure 11.8

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 Answer omitted.

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.
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.

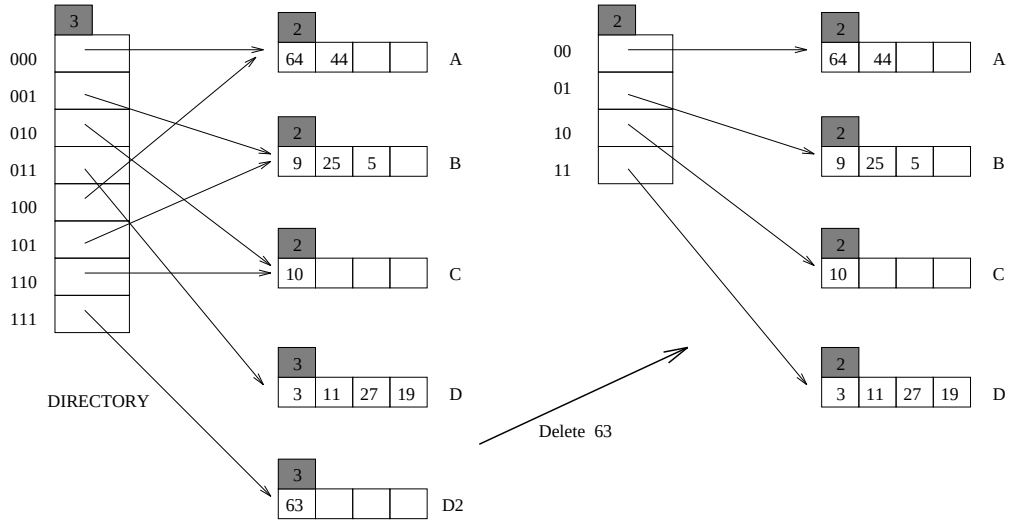

3
000
001
010
011
100
101
110
111
DIRECTORY
2
64
44
A
2
9
25
5
B
2
10
C
3
3
11
27
19
D
3
63
D2
Delete 63
2
00
01
10
11
2
64
44
A
2
9
25
5
B
2
10
C
2
3
11
27
19
D

Figure 11.9

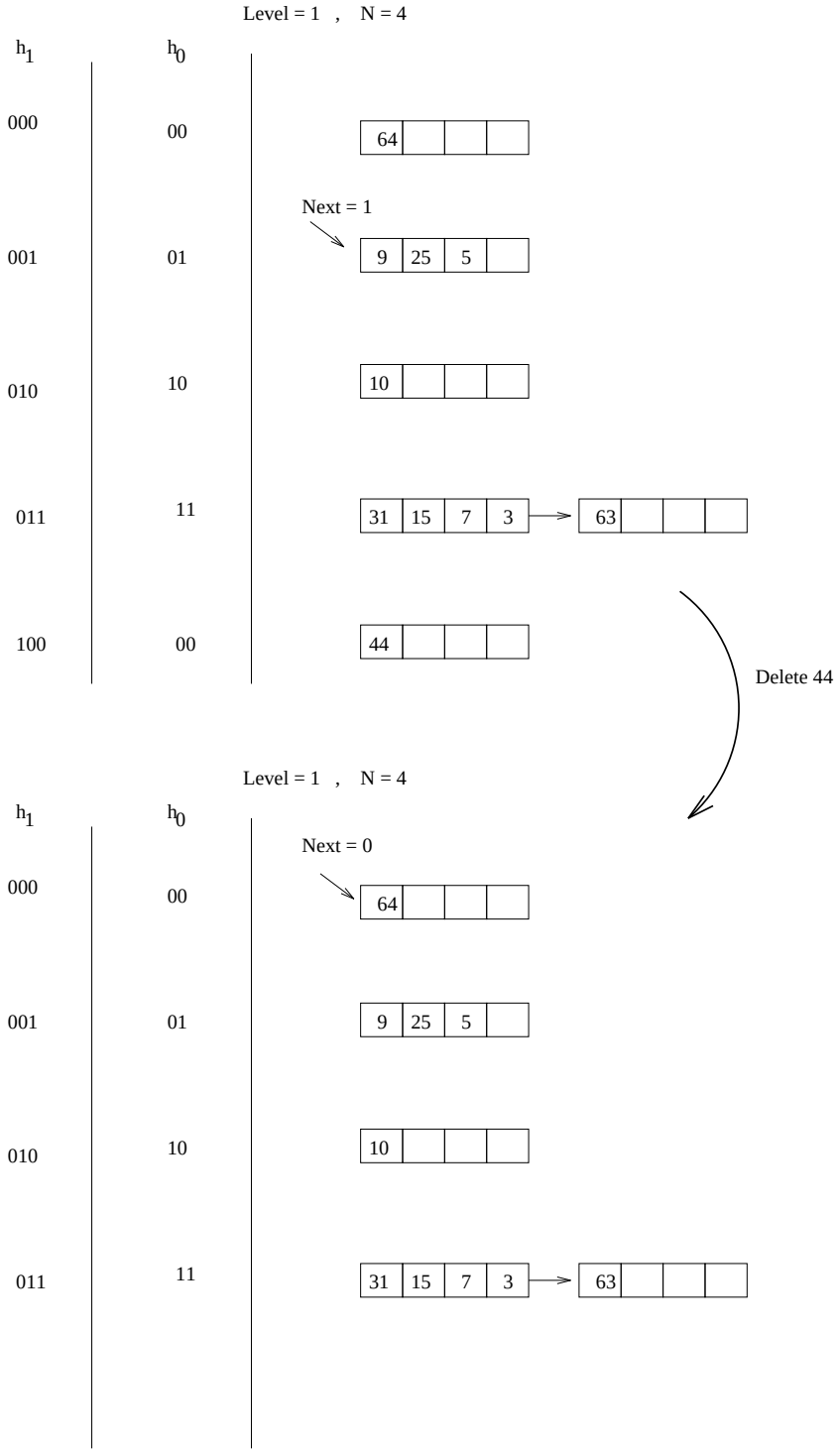

Level = 1 , N = 4
h1
h0
000
00
64
Next = 1
001
01
9
25
5
010
10
10
011
11
31
15
7
3
63
100
00
44
Delete 44
Level = 1 , N = 4
h1
h0
Next = 0
000
00
64
001
01
9
25
5
010
10
10
011
11
31
15
7
3
63

Figure 11.10

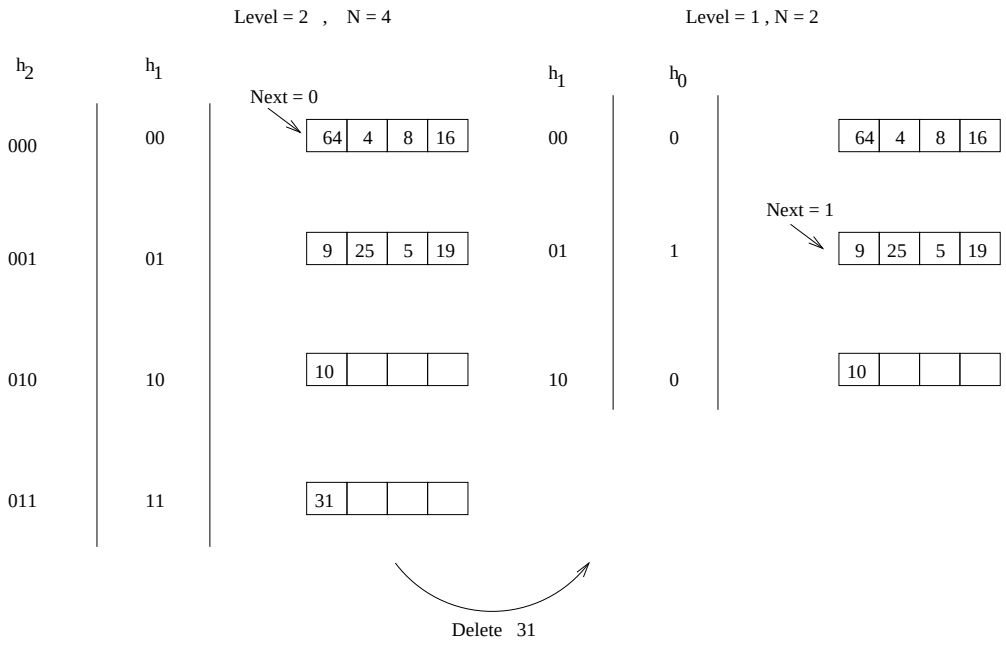

Level = 2 , N = 4
Level = 1 , N = 2
h2
h1
h1
h0
Next = 0
Next = 1
000
001
010
011
00
01
10
11
64 4 8 16
9 25 5 19
10
31
00
01
10
0
1
0
64 4 8 16
9 25 5 19
10
Delete 31

Figure 11.11

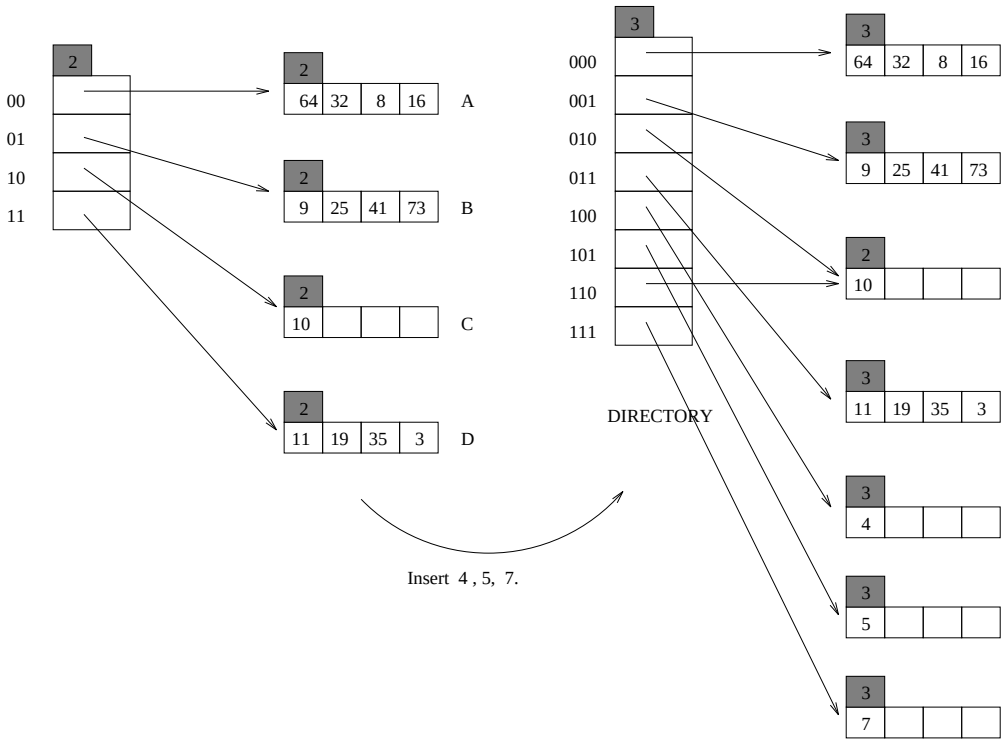

2
00
01
10
11
2
64 32 8 16
A
2
9 25 41 73
B
2
10
C
2
11 19 35 3
D
Insert 4 , 5, 7.
3
000
001
010
011
100
101
110
111
DIRECTORY
3
64 32 8 16
3
9 25 41 73
2
10
3
11 19 35 3
3
4
3
5
3
7

Figure 11.12

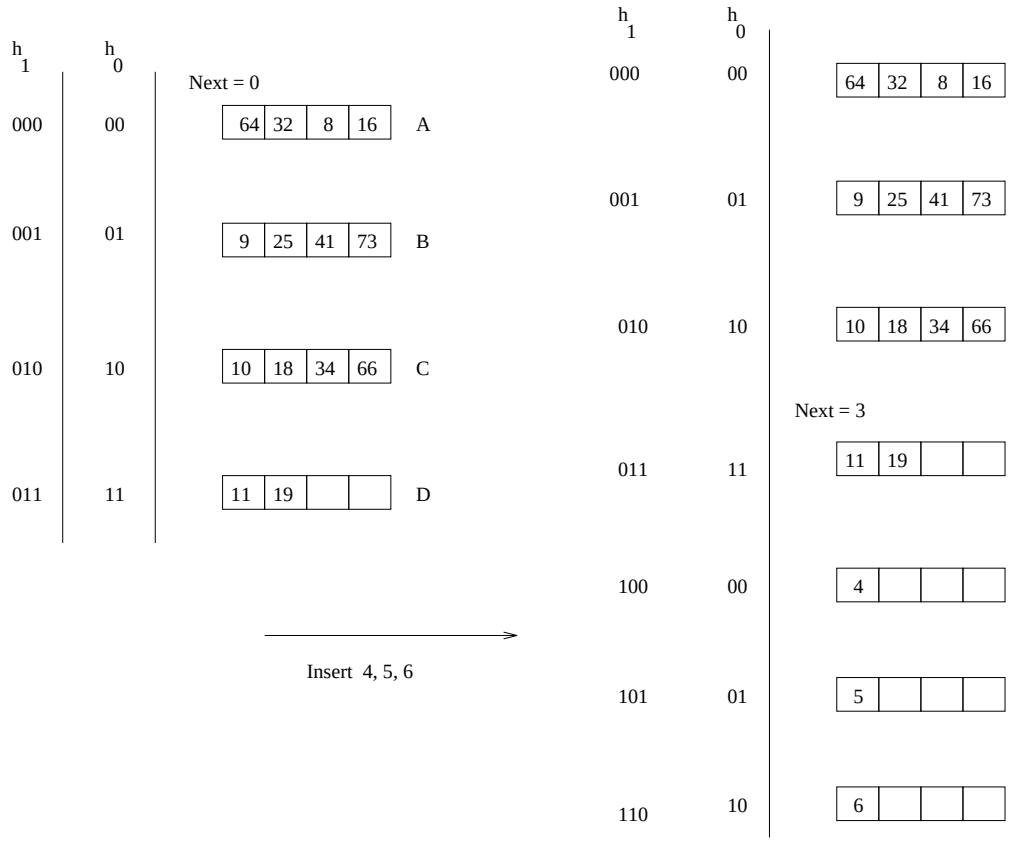

h1
h0
Next = 0
000
00
64 32 8 16
A
001
01
9 25 41 73
B
010
10
10 18 34 66
C
011
11
11 19
D
Insert 4, 5, 6
h1
h0
000
00
64 32 8 16
001
01
9 25 41 73
010
10
10 18 34 66
Next = 3
011
11
11 19
100
00
4
101
01
5
110
10
6

Figure 11.13

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 1. Answer not available yet.

2. Anwswer not available yet.
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.

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 Answer omitted.

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 Answer omitted.

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 $\langle$ Sailors.sid $\rangle$, 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 $\langle$ Sailors.sid $\rangle$, 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 Answer not available yet.

Exercise 12.6 Consider the two join methods described in Section **??**. 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 **??**. 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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 Answer omitted.

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 14.1 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 Answer omitted.

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.