



The Relational Data Model

One of the most important applications for computers is storing and managing information. The manner in which information is organized can have a profound effect on how easy it is to access and manage. Perhaps the simplest but most versatile way to organize information is to store it in tables.

The relational model is centered on this idea: the organization of data into collections of two-dimensional tables called “relations.” We can also think of the relational model as a generalization of the set data model that we discussed in Chapter 7, extending binary relations to relations of arbitrary arity.

Database

Originally, the relational data model was developed for *databases* — that is, information stored over a long period of time in a computer system — and for *database management systems*, the software that allows people to store, access, and modify this information. Databases still provide us with important motivation for understanding the relational data model. They are found today not only in their original, large-scale applications such as airline reservation systems or banking systems, but in desktop computers handling individual activities such as maintaining expense records, homework grades, and many other uses.

Other kinds of software besides database systems can make good use of tables of information as well, and the relational data model helps us design these tables and develop the data structures that we need to access them efficiently. For example, such tables are used by compilers to store information about the variables used in the program, keeping track of their data type and of the functions for which they are defined.



8.1 What This Chapter Is About

There are three intertwined themes in this chapter. First, we introduce you to the design of information structures using the relational model. We shall see that

- ◆ Tables of information, called “relations,” are a powerful and flexible way to represent information (Section 8.2).

- ◆ An important part of the design process is selecting “attributes,” or properties of the described objects, that can be kept together in a table, without introducing “redundancy,” a situation where a fact is repeated several times (Section 8.2).
- ◆ The columns of a table are named by attributes. The “key” for a table (or relation) is a set of attributes whose values uniquely determine the values of a whole row of the table. Knowing the key for a table helps us design data structures for the table (Section 8.3).
- ◆ Indexes are data structures that help us retrieve or change information in tables quickly. Judicious selection of indexes is essential if we want to operate on our tables efficiently (Sections 8.4, 8.5, and 8.6).

The second theme is the way data structures can speed access to information. We shall learn that

- ◆ Primary index structures, such as hash tables, arrange the various rows of a table in the memory of a computer. The right structure can enhance efficiency for many operations (Section 8.4).
- ◆ Secondary indexes provide additional structure and help perform other operations efficiently (Sections 8.5 and 8.6).

Our third theme is a very high-level way of expressing “queries,” that is, questions about the information in a collection of tables. The following points are made:

- ◆ Relational algebra is a powerful notation for expressing queries without giving details about how the operations are to be carried out (Section 8.7).
- ◆ The operators of relational algebra can be implemented using the data structures discussed in this chapter (Section 8.8).
- ◆ In order that we may get answers quickly to queries expressed in relational algebra, it is often necessary to “optimize” them, that is, to use algebraic laws to convert an expression into an equivalent expression with a faster evaluation strategy. We learn some of the techniques in Section 8.9.

◆◆◆ 8.2 Relations

Section 7.7 introduced the notion of a “relation” as a set of tuples. Each tuple of a relation is a list of components, and each relation has a fixed arity, which is the number of components each of its tuples has. While we studied primarily binary relations, that is, relations of arity 2, we indicated that relations of other arities were possible, and indeed can be quite useful.

The relational model uses a notion of “relation” that is closely related to this set-theoretic definition, but differs in some details. In the relational model, information is stored in tables such as the one shown in Fig. 8.1. This particular table represents data that might be stored in a registrar’s computer about courses, students who have taken them, and the grades they obtained.

The columns of the table are given names, called *attributes*. In Fig. 8.1, the attributes are Course, StudentId, and Grade.

Attribute

Course	StudentId	Grade
CS101	12345	A
CS101	67890	B
EE200	12345	C
EE200	22222	B+
CS101	33333	A-
PH100	67890	C+

Fig. 8.1. A table of information.

Relations as Sets Versus Relations as Tables

In the relational model, as in our discussion of set-theoretic relations in Section 7.7, a relation is a set of tuples. Thus the order in which the rows of a table are listed has no significance, and we can rearrange the rows in any way without changing the value of the table, just as we can rearrange the order of elements in a set without changing the value of the set.

The order of the components in each row of a table is significant, since different columns are named differently, and each component must represent an item of the kind indicated by the header of its column. In the relational model, however, we may permute the order of the columns along with the names of their headers and keep the relation the same. This aspect of database relations is different from set-theoretic relations, but rarely shall we reorder the columns of a table, and so we can keep the same terminology. In cases of doubt, the term “relation” in this chapter will always have the database meaning.

Tuple

Each row in the table is called a *tuple* and represents a basic fact. The first row, (CS101, 12345, A), represents the fact that the student with ID number 12345 got an A in the course CS101.

A table has two aspects:

1. The set of column names, and
2. The rows containing the information.

The term “relation” refers to the latter, that is, the set of rows. Each row represents a tuple of the relation, and the order in which the rows appear in the table is immaterial. No two rows of the same table may have identical values in all columns.

Relation scheme

Item (1), the set of column names (attributes) is called the *scheme* of the relation. The order in which the attributes appear in the scheme is immaterial, but we need to know the correspondence between the attributes and the columns of the table in order to write the tuples properly. Frequently, we shall use the scheme as the name of the relation. Thus the table in Fig. 8.1 will often be called the Course-StudentId-Grade relation. Alternatively, we could give the relation a name, like *CSG*.

Representing Relations

As sets, there are a variety of ways to represent relations by data structures. A table looks as though its rows should be structures, with fields corresponding to the column names. For example, the tuples in the relation of Fig. 8.1 could be represented by structures of the type

```
struct CSG {
    char Course[5];
    int StudentId;
    char Grade[2];
};
```

The table itself could be represented in any of a number of ways, such as

1. An array of structures of this type.
2. A linked list of structures of this type, with the addition of a `next` field to link the cells of the list.

Additionally, we can identify one or more attributes as the “domain” of the relation and regard the remainder of the attributes as the “range.” For instance, the relation of Fig. 8.1 could be viewed as a relation from domain `Course` to a range consisting of `StudentId-Grade` pairs. We could then store the relation in a hash table according to the scheme for binary relations that we discussed in Section 7.9. That is, we hash `Course` values, and the elements in buckets are `Course-StudentId-Grade` triples. We shall take up this issue of data structures for relations in more detail, starting in Section 8.4.

Databases

A collection of relations is called a *database*. The first thing we need to do when designing a database for some application is to decide on how the information to be stored should be arranged into tables. Design of a database, like all design problems, is a matter of business needs and judgment. In an example to follow, we shall expand our application of a registrar’s database involving courses, and thereby expose some of the principles of good database design.

Some of the most powerful operations on a database involve the use of several relations to represent coordinated types of data. By setting up appropriate data structures, we can jump from one relation to another efficiently, and thus obtain information from the database that we could not uncover from a single relation. The data structures and algorithms involved in “navigating” among relations will be discussed in Sections 8.6 and 8.8.

The set of schemes for the various relations in a database is called the *scheme* of the database. Notice the difference between the scheme for the database, which tells us something about how information is organized in the database, and the set of tuples in each relation, which is the actual information stored in the database.

**Database
scheme**

- ◆ **Example 8.1.** Let us supplement the relation of Fig. 8.1, which has scheme

$$\{\text{Course, StudentId, Grade}\}$$

with four other relations. Their schemes and intuitive meanings are:

1. {StudentId, Name, Address, Phone}. The student whose ID appears in the first component of a tuple has name, address, and phone number equal to the values appearing in the second, third, and fourth components, respectively.
2. {Course, Prerequisite}. The course named in the second component of a tuple is a prerequisite for the course named in the first component of that tuple.
3. {Course, Day, Hour}. The course named in the first component meets on the day specified by the second component, at the hour named in the third component.
4. {Course, Room}. The course named in the first component meets in the room indicated by the second component.

These four schemes, plus the scheme {Course, StudentId, Grade} mentioned earlier, form the database scheme for a running example in this chapter. We also need to offer an example of a possible “current value” for the database. Figure 8.1 gave an example for the Course-StudentId-Grade relation, and example relations for the other four schemes are shown in Fig. 8.2. Keep in mind that these relations are all much shorter than we would find in reality; we are just offering some sample tuples for each. ♦

Queries on a Database

We saw in Chapter 7 some of the most important operations performed on relations and functions; they were called *insert*, *delete*, and *lookup*, although their appropriate meanings differed, depending on whether we were dealing with a dictionary, a function, or a binary relation. There is a great variety of operations one can perform on database relations, especially on combinations of two or more relations, and we shall give a feel for this spectrum of operations in Section 8.7. For the moment, let us focus on the basic operations that we might perform on a single relation. These are a natural generalization of the operations discussed in the previous chapter.

1. *insert*(t, R). We add the tuple t to the relation R , if it is not already there. This operation is in the same spirit as *insert* for dictionaries or binary relations.
2. *delete*(X, R). Here, X is intended to be a specification of some tuples. It consists of components for each of the attributes of R , and each component can be either
 - a) A value, or
 - b) The symbol *, which means that any value is acceptable.

The effect of this operation is to delete all tuples that match the specification X . For example, if we cancel CS101, we want to delete all tuples of the

Course-Day-Hour

relation that have Course = “CS101.” We could express this condition by

$$\textit{delete}((\text{“CS101”}, *, *), \text{Course-Day-Hour})$$

That operation would delete the first three tuples of the relation in Fig. 8.2(c), because their first components each are the same value as the first component of the specification, and their second and third components all match *, as any values do.

StudentId	Name	Address	Phone
12345	C. Brown	12 Apple St.	555-1234
67890	L. Van Pelt	34 Pear Ave.	555-5678
22222	P. Patty	56 Grape Blvd.	555-9999

(a) StudentId-Name-Address-Phone

Course	Prerequisite
CS101	CS100
EE200	EE005
EE200	CS100
CS120	CS101
CS121	CS120
CS205	CS101
CS206	CS121
CS206	CS205

(b) Course-Prerequisite

Course	Day	Hour
CS101	M	9AM
CS101	W	9AM
CS101	F	9AM
EE200	Tu	10AM
EE200	W	1PM
EE200	Th	10AM

(c) Course-Day-Hour

Course	Room
CS101	Turing Aud.
EE200	25 Ohm Hall
PH100	Newton Lab.

(d) Course-Room

Fig. 8.2. Sample relations.

3. $lookup(X, R)$. The result of this operation is the set of tuples in R that match the specification X ; the latter is a symbolic tuple as described in the preceding item (2). For example, if we wanted to know for what courses CS101 is a prerequisite, we could ask

$$lookup((*, "CS101"), \text{Course-Prerequisite})$$

The result would be the set of two matching tuples

(CS120, CS101)
(CS205, CS101)

◆ **Example 8.2.** Here are some more examples of operations on our registrar's database:

- a) $lookup(("CS101", 12345, *), \text{Course-StudentId-Grade})$ finds the grade of the student with ID 12345 in CS101. Formally, the result is the one matching tuple, namely the first tuple in Fig. 8.1.
- b) $lookup(("CS205", "CS120"), \text{Course-Prerequisite})$ asks whether CS120 is a prerequisite of CS205. Formally, it produces as an answer either the single tuple ("CS205", "CS120") if that tuple is in the relation, or the empty set if not. For the particular relation of Fig. 8.2(b), the empty set is the answer.
- c) $delete(("CS101", *), \text{Course-Room})$ drops the first tuple from the relation of Fig. 8.2(d).
- d) $insert(("CS205", "CS120"), \text{Course-Prerequisite})$ makes CS120 a prerequisite of CS205.
- e) $insert(("CS205", "CS101"), \text{Course-Prerequisite})$ has no effect on the relation of Fig. 8.2(b), because the inserted tuple is already there. ◆

The Design of Data Structures for Relations

In much of the rest of this chapter, we are going to discuss the issue of how one selects a data structure for a relation. We have already seen some of the problem when we discussed the implementation of binary relations in Section 7.9. The relation Variety-Pollinizer was given a hash table on Variety as its data structure, and we observed that the structure was very useful for answering queries like

$$lookup(("Wickson", *), \text{Variety-Pollinizer})$$

because the value "Wickson" lets us find a specific bucket in which to search. But that structure was of no help answering queries like

$$lookup(*, "Wickson"), \text{Variety-Pollinizer})$$

because we would have to look in all buckets.

Whether a hash table on Variety is an adequate data structure depends on the expected mix of queries. If we expect the variety always to be specified, then a hash table is adequate, and if we expect the variety sometimes not to be specified, as in the preceding query, then we need to design a more powerful data structure.

The selection of a data structure is one of the essential design issues we tackle in this chapter. In the next section, we shall generalize the basic data structures for functions and relations from Sections 7.8 and 7.9, to allow for several attributes in either the domain or the range. These structures will be called "primary index structures." Then, in Section 8.5 we introduce "secondary index structures," which are additional structures that allow us to answer a greater variety of queries efficiently. At that point, we shall see how both the above queries, and others we might ask about the Variety-Pollinizer relation, can be answered efficiently, that is, in about as much time as it takes to list all the answers.

Design I: Selecting a Database Scheme

An important issue when we use the relational data model is how we select an appropriate database scheme. For instance, why did we separate information about courses into five relations, rather than have one table with scheme

$$\{\text{Course, StudentId, Grade, Prerequisite, Day, Hour, Room}\}$$

The intuitive reason is that

- ◆ If we combine into one relation scheme information of two independent types, we may be forced to repeat the same fact many times.

For example, prerequisite information about a course is independent of day and hour information. If we were to combine prerequisite and day-hour information, we would have to list the prerequisites for a course in conjunction with every meeting of the course, and vice versa. Then the data about EE200 found in Fig. 8.2(b) and (c), if put into a single relation with scheme

$$\{\text{Course, Prerequisite, Day, Hour}\}$$

would look like

Course	Prerequisite	Day	Hour
EE200	EE005	Tu	10AM
EE200	EE005	W	1PM
EE200	EE005	Th	10AM
EE200	CS100	Tu	10AM
EE200	CS100	W	1PM
EE200	CS100	Th	10AM

Notice that we take six tuples, with four components each, to do the work previously done by five tuples, with two or three components each.

- ◆ Conversely, do not separate attributes when they represent connected information.

For example, we cannot replace the Course-Day-Hour relation by two relations, one with scheme Course-Day and the other with scheme Course-Hour. For then, we could only tell that EE200 meets Tuesday, Wednesday, and Thursday, and that it has meetings at 10AM and 1PM, but we could not tell when it met on each of its three days.

EXERCISES

8.2.1: Give appropriate structure declarations for the tuples of the relations of Fig. 8.2(a) through (d).

8.2.2*: What is an appropriate database scheme for

- a) A telephone directory, including all the information normally found in a directory, such as area codes.

- b) A dictionary of the English language, including all the information normally found in the dictionary, such as word origins and parts of speech.
- c) A calendar, including information normally found on a calendar such as holidays, good for the years 1 through 4000.

◆ 8.3 Keys

Many database relations can be considered functions from one set of attributes to the remaining attributes. For example, we might choose to view the

Course-StudentId-Grade

relation as a function whose domain is Course-StudentId pairs and whose range is Grade. Because functions have somewhat simpler data structures than general relations, it helps if we know a set of attributes that can serve as the domain of a function. Such a set of attributes is called a “key.”

More formally, a *key* for a relation is a set of one or more attributes such that under no circumstances will the relation have two tuples whose values agree in each column headed by a key attribute. Frequently, there are several different sets of attributes that could serve as a key for a relation, but we normally pick one and refer to it as “the key.”

Finding Keys

Because keys can be used as the domain of a function, they play an important role in the next section when we discuss primary index structures. In general, we cannot deduce or prove that a set of attributes forms a key; rather, we need to examine carefully our assumptions about the application being modeled and how they are reflected in the database scheme we are designing. Only then can we know whether it is appropriate to use a given set of attributes as a key. There follows a sequence of examples that illustrate some of the issues.

◆ **Example 8.3.** Consider the relation StudentId-Name-Address-Phone of Fig. 8.2(a). Evidently, the intent is that each tuple gives information about a different student. We do not expect to find two tuples with the same ID number, because the whole purpose of such a number is to give each student a unique identifier.

If we have two tuples with identical student ID numbers in the same relation, then one of two things has gone wrong.

1. If the two tuples are identical in all components, then we have violated our assumption that a relation is a set, because no element can appear more than once in a set.
2. If the two tuples have identical ID numbers but disagree in at least one of the Name, Address, or Phone columns, then there is something wrong with the data. Either we have two different students with the same ID (if the tuples differ in Name), or we have mistakenly recorded two different addresses and/or phone numbers for the same student.

Thus it is reasonable to take the `StudentId` attribute by itself as a key for the `StudentId-Name-Address-Phone` relation.

However, in declaring `StudentId` a key, we have made a critical assumption, enunciated in item (2) preceding, that we never want to store two names, addresses, or phone numbers for one student. But we could just as well have decided otherwise, for example, that we want to store for each student both a home address and a campus address. If so, we would probably be better off designing the relation to have five attributes, with `Address` replaced by `HomeAddress` and `LocalAddress`, rather than have two tuples for each student, with all but the `Address` component the same. If we did use two tuples — differing in their `Address` components only — then `StudentId` would no longer be a key but $\{\text{StudentId}, \text{Address}\}$ would be a key. ♦

- ♦ **Example 8.4.** Examining the `Course-StudentId-Grade` relation of Fig. 8.1, we might imagine that `Grade` was a key, since we see no two tuples with the same grade. However, this reasoning is fallacious. In this little example of six tuples, no two tuples hold the same grade; but in a typical `Course-StudentId-Grade` relation, which would have thousands or tens of thousands of tuples, surely there would be many grades appearing more than once.

Most probably, the intent of the designers of the database is that `Course` and `StudentId` together form a key. That is, assuming students cannot take the same course twice, we could not have two different grades assigned to the same student in the same course; hence, there could not be two different tuples that agreed in both `Course` and `StudentId`. Since we would expect to find many tuples with the same `Course` component and many tuples with the same `StudentId` component, neither `Course` nor `StudentId` by itself would be a key.

However, our assumption that students can get only one grade in any course is another design decision that could be questioned, depending on the policy of the school. Perhaps when course content changes sufficiently, a student may reregister for the course. If that were the case, we would not declare $\{\text{Course}, \text{StudentId}\}$ to be a key for the `Course-StudentId-Grade` relation; rather, the set of all three attributes would be the only key. (Note that the set of all attributes for a relation can always be used as a key, since two identical tuples cannot appear in a relation.) In fact, it would be better to add a fourth attribute, `Date`, to indicate when a course was taken. Then we could handle the situation where a student took the same course twice and got the same grade each time. ♦

- ♦ **Example 8.5.** In the `Course-Prerequisite` relation of Fig. 8.2(b), neither attribute by itself is a key, but the two attributes together form a key. ♦
- ♦ **Example 8.6.** In the `Course-Day-Hour` relation of Fig. 8.2(c), all three attributes form the only reasonable key. Perhaps `Course` and `Day` alone could be declared a key, but then it would be impossible to store the fact that a course met twice in one day (e.g., for a lecture and a lab). ♦

Design II: Selecting a Key

Determining a key for a relation is an important aspect of database design; it is used when we select a primary index structure in Section 8.4.

◆ You can't tell the key by looking at an example value for the relation.

That is, appearances can be deceptive, as in the matter of Grade for the Course-StudentId-Grade relation of Fig. 8.1, which we discuss in Example 8.4.

◆ There is no one "right" key selection; what is a key depends on assumptions made about the types of data the relations will hold.

◆ **Example 8.7.** Finally, consider the Course-Room relation of Fig. 8.2(d). We believe that Course is a key; that is, no course meets in two or more different rooms. If that were not the case, then we should have combined the Course-Room relation with the Course-Day-Hour relation, so we could tell which meetings of a course were held in which rooms. ◆

EXERCISES

8.3.1*: Suppose we want to store home and local addresses and also home and local phones for students in the StudentId-Name-Address-Phone relation.

- a) What would then be the most suitable key for the relation?
- b) This change causes redundancy; for example, the name of a student could be repeated four times as his or her two addresses and two phones are combined in all possible ways in different tuples. We suggested in Example 8.3 that one solution is to use separate attributes for the different addresses and different phones. What would the relation scheme be then? What would be the most suitable key for this relation?
- c) Another approach to handling redundancy, which we suggested in Section 8.2, is to split the relation into two relations, with different schemes, that together hold all the information of the original. Into what relations should we split StudentId-Name-Address-Phone, if we are going to allow multiple addresses and phones for one student? What would be the most suitable keys for these relations? *Hint:* A critical issue is whether addresses and phones are independent. That is, would you expect a phone number to ring in all addresses belonging to one student (in which case address and phone are independent), or are phones associated with single addresses?

8.3.2*: The Department of Motor Vehicles keeps a database with the following kinds of information.

1. The name of a driver (Name).
2. The address of a driver (Addr).
3. The license number of a driver (LicenseNo).
4. The serial number of an automobile (SerialNo).
5. The manufacturer of an automobile (Manf).
6. The model name of an automobile (Model).

7. The registration (license plate) number of an automobile (RegNo).

The DMV wants to associate with each driver the relevant information: address, driver's license, and autos owned. It wants to associate with each auto the relevant information: owner(s), serial number, manufacturer, model, and registration. We assume that you are familiar with the basics of operation of the DMV; for example, it strives not to issue the same license plate to two cars. You may not know (but it is a fact) that no two autos, even with different manufacturers, will be given the same serial number.

- a) Select a database scheme — that is, a collection of relation schemes — each consisting of a set of the attributes 1 through 7 listed above. You must allow any of the desired connections to be found from the data stored in these relations, and you must avoid redundancy; that is, your scheme should not require you to store the same fact repeatedly.
- b) Suggest what attributes, if any, could serve as keys for your relations from part (a).



8.4 Primary Storage Structures for Relations

In Sections 7.8 and 7.9 we saw how certain operations on functions and binary relations were speeded up by storing pairs according to their domain value. In terms of the general *insert*, *delete*, and *lookup* operations that we defined in Section 8.2, the operations that are helped are those where the domain value is specified. Recalling the Variety-Pollinizer relation from Section 7.9 again, if we regard Variety as the domain of the relation, then we favor operations that specify a variety but we do not care whether a pollinizer is specified.

Here are some structures we might use to represent a relation.

1. A binary search tree, with a “less than” relation on domain values to guide the placement of tuples, can serve to facilitate operations in which a domain value is specified.
2. An array used as a characteristic vector, with domain values as the array index, can sometimes serve.
3. A hash table in which we hash domain values to find buckets will serve.
4. In principle, a linked list of tuples is a candidate structure. We shall ignore this possibility, since it does not facilitate operations of any sort.

The same structures work when the relation is not binary. In place of a single attribute for the domain, we may have a combination of k attributes, which we call the *domain attributes* or just the “domain” when it is clear we are referring to a set of attributes. Then, domain values are k -tuples, with one component for each attribute of the domain. The *range attributes* are all those attributes other than the domain attributes. The range values may also have several components, one for each attribute of the range.

In general, we have to pick which attributes we want for the domain. The easiest case occurs when there is one or a small number of attributes that serve as a key for the relation. Then it is common to choose the key attribute(s) as the

**Domain and
range attributes**

domain and the rest as the range. In cases where there is no key (except the set of all attributes, which is not a useful key), we may pick any set of attributes as the domain. For example, we might consider typical operations that we expect to perform on the relation and pick for the domain an attribute we expect will be specified frequently. We shall see some concrete examples shortly.

Once we have selected a domain, we can select any of the four data structures just named to represent the relation, or indeed we could select another structure. However, it is common to choose a hash table based on domain values as the index, and we shall generally do so here.

Primary index

The chosen structure is said to be the *primary index structure* for the relation. The adjective “primary” refers to the fact that the location of tuples is determined by this structure. An *index* is a data structure that helps find tuples, given a value for one or more components of the desired tuples. In the next section, we shall discuss “secondary” indexes, which help answer queries but do not affect the location of the data.

```
typedef struct TUPLE *TUPLELIST;
struct TUPLE {
    int StudentId;
    char Name[30];
    char Address[50];
    char Phone[8];
    TUPLELIST next;
};
typedef TUPLELIST HASHTABLE[1009];
```

Fig. 8.3. Types for a hash table as primary index structure.

- ◆ **Example 8.8.** Let us consider the StudentId-Name-Address-Phone relation, which has key StudentId. This attribute will serve as our domain, and the other three attributes will form the range. We may thus see the relation as a function from StudentId to Name-Address-Phone triples.

As with all functions, we select a hash function that takes a domain value as argument and produces a bucket number as result. In this case, the hash function takes student ID numbers, which are integers, as arguments. We shall choose the number of buckets, B , to be 1009,¹ and the hash function to be

$$h(x) = x \% 1009$$

This hash function maps ID's to integers in the range 0 to 1008.

An array of 1009 bucket headers takes us to a list of structures. The structures on the list for bucket i each represent a tuple whose StudentId component is an integer whose remainder, when divided by 1009, is i . For the StudentId-Name-Address-Phone relation, the declarations in Fig. 8.3 are suitable for the structures

¹ 1009 is a convenient prime around 1000. We might choose about 1000 buckets if there were several thousand students in our database, so that the average number of tuples in a bucket would be small.

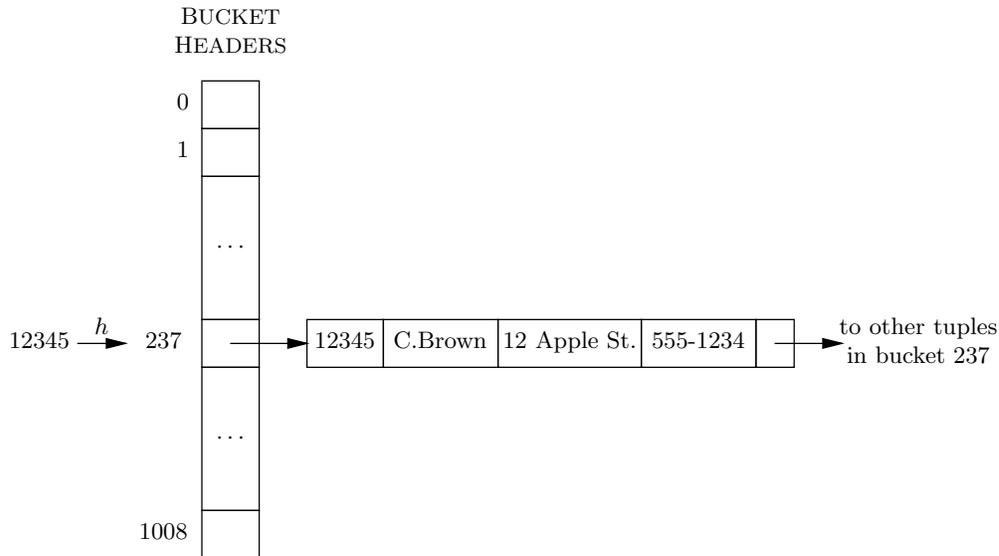


Fig. 8.4. Hash table representing StudentId-Name-Address-Phone relation.

in the linked lists of the buckets and for the bucket header array. Figure 8.4 suggests what the hash table would look like. ♦

- ♦ **Example 8.9.** For a more complicated example, consider the Course-StudentId-Grade relation. We could use as a primary structure a hash table whose hash function took as argument both the course and the student (i.e., both attributes of the key for this relation). Such a hash function might take the characters of the course name, treat them as integers, add those integers to the student ID number, and divide by 1009, taking the remainder.

That data structure would be useful if all we ever did was look up grades, given a course and a student ID — that is, we performed operations like

$$\text{lookup}((\text{"CS101"}, 12345, *), \text{Course-StudentId-Grade})$$

However, it is not useful for operations such as

1. Finding all the students taking CS101, or
2. Finding all the courses being taken by the student whose ID is 12345.

In either case, we would not be able to compute a value for the hash function. For example, given only the course, we do not have a student ID to add to the sum of the characters converted to integers, and thus have no value to divide by 1009 to get the bucket number.

However, suppose it is quite common to ask queries like, “Who is taking CS101?” that is,

$$\text{lookup}((\text{"CS101"}, *, *), \text{Course-StudentId-Grade})$$

Design III: Choosing a Primary Index

- ◆ It is often useful to make the key for a relation scheme be the domain of a function and the remaining attributes be the range.

Then, the relation can be implemented as if it were a function, using a primary index such as a hash table, with the hash function based on the key attributes.

- ◆ However, if the most common type of query specifies values for an attribute or a set of attributes that do not form a key, we may prefer to use this set of attributes as the domain, with the other attributes as the range.

We may then implement this relation as a binary relation (e.g., by a hash table). The only problem is that the division of the tuples into buckets may not be as even as we would expect were the domain a key.

- ◆ The choice of domain for the primary index structure probably has the greatest influence over the speed with which we can execute “typical” queries.
-
-

We might find it more efficient to use a primary structure based only on the value of the Course component. That is, we may regard our relation as a binary relation in the set-theoretic sense, with domain equal to Course and range the StudentId-Grade pairs.

For instance, suppose we convert the characters of the course name to integers, sum them, divide by 197, and take the remainder. Then the tuples of the

Course-StudentId-Grade

relation would be divided by this hash function into 197 buckets, numbered 0 through 196. However, if CS101 has 100 students, then there would be at least 100 structures in its bucket, regardless of how many buckets we chose for our hash table; that is the disadvantage of using something other than a key on which to base our primary index structure. There could even be more than 100 structures, if some other course were hashed to the same bucket as CS101.

On the other hand, we still get help when we want to find the students in a given course. If the number of courses is significantly more than 197, then on the average, we shall have to search something like $1/197$ of the entire

Course-StudentId-Grade

relation, which is a great saving. Moreover, we get some help when performing operations like looking up a particular student’s grade in a particular course, or inserting or deleting a Course-StudentId-Grade tuple. In each case, we can use the Course value to restrict our search to one of the 197 buckets of the hash table. The only sort of operation for which no help is provided is one in which no course is specified. For example, to find the courses taken by student 12345, we must search all the buckets. Such a query can be made more efficient only if we use a secondary index structure, as discussed in the next section. ◆

Insert, Delete, and Lookup Operations

The way in which we use a primary index structure to perform the operations *insert*, *delete*, and *lookup* should be obvious, given our discussion of the same subject for

binary relations in Chapter 7. To review the ideas, let us focus on a hash table as the primary index structure. If the operation specifies a value for the domain, then we hash this value to find a bucket.

1. To insert a tuple t , we examine the bucket to check that t is not already there, and we create a new cell on the bucket's list for t if it is not.
2. To delete tuples that match a specification X , we find the domain value from X , hash to find the proper bucket, and run down the list for this bucket, deleting each tuple that matches the specification X .
3. To lookup tuples according to a specification X , we again find the domain value from X and hash that value to find the proper bucket. We run down the list for that bucket, producing as an answer each tuple on the list that matches the specification X .

If the operation does not specify the domain value, we are not so fortunate. An *insert* operation always specifies the inserted tuple completely, but a *delete* or *lookup* might not. In those cases, we must search all the bucket lists for matching tuples and delete or list them, respectively.

EXERCISES

8.4.1: The DMV database of Exercise 8.3.2 should be designed to handle the following sorts of queries, all of which may be assumed to occur with significant frequency.

1. What is the address of a given driver?
2. What is the license number of a given driver?
3. What is the name of the driver with a given license number?
4. What is the name of the driver who owns a given automobile, identified by its registration number?
5. What are the serial number, manufacturer, and model of the automobile with a given registration number?
6. Who owns the automobile with a given registration number?

Suggest appropriate primary index structures for the relations you designed in Exercise 8.3.2, using a hash table in each case. State your assumptions about how many drivers and automobiles there are. Tell how many buckets you suggest, as well as what the domain attribute(s) are. How many of these types of queries can you answer efficiently, that is, in average time $O(1)$ independent of the size of the relations?

8.4.2: The primary structure for the Course-Day-Hour relation of Fig. 8.2(c) might depend on the typical operations we intended to perform. Suggest an appropriate hash table, including both the attributes in the domain and the number of buckets if the typical queries are of each of the following forms. You may make reasonable assumptions about how many courses and different class periods there are. In each case, a specified value like "CS101" is intended to represent a "typical" value; in this case, we would mean that Course is specified to be some particular course.

- a) *lookup*((“CS101”, “M”, *), Course-Day-Hour).
- b) *lookup*(*, “M”, “9AM”), Course-Day-Hour).
- c) *delete*((“CS101”, *, *), Course-Day-Hour).
- d) Half of type (a) and half of type (b).
- e) Half of type (a) and half of type (c).
- f) Half of type (b) and half of type (c).

❖❖❖ 8.5 Secondary Index Structures

Suppose we store the StudentId-Name-Address-Phone relation in a hash table, where the hash function is based on the key StudentId, as in Fig. 8.4. This primary index structure helps us answer queries in which the student ID number is specified. However, perhaps we wish to ask questions in terms of students’ names, rather than impersonal — and probably unknown — ID’s. For example, we might ask, “What is the phone number of the student named C. Brown?” Now, our primary index structure gives no help. We must go to each bucket and examine the lists of records until we find one whose Name field has value “C. Brown.”

To answer such a query rapidly, we need an additional data structure that takes us from a name to the tuple or tuples with that name in the Name component of the tuple.² A data structure that helps us find tuples — given a value for a certain attribute or attributes — but is not used to position the tuples within the overall structure, is called a *secondary index*.

What we want for our secondary index is a binary relation whose

1. Domain is Name.
2. Range is the set of pointers to tuples of the StudentId-Name-Address-Phone relation.

In general, a secondary index on attribute A of relation R is a set of pairs (v, p) , where

- a) v is a value for attribute A , and
- b) p is a pointer to one of the tuples, in the primary index structure for relation R , whose A -component has the value v .

The secondary index has one such pair for each tuple with the value v in attribute A .

We may use any of the data structures for binary relations for storing secondary indexes. Usually, we would expect to use a hash table on the value of the attribute A . As long as the number of buckets is no greater than the number of different values of attribute A , we can normally expect good performance — that is, $O(n/B)$ time, on the average — to find one pair (v, p) in the hash table, given a desired value of v . (Here, n is the number of pairs and B is the number of buckets.) To show that other structures are possible for secondary (or primary) indexes, in the next example we shall use a binary search tree as a secondary index.

² Remember that Name is not a key for the StudentId-Name-Address-Phone relation, despite the fact that in the sample relation of Fig. 8.2(a), there are no tuples that have the same Name value. For example, if Linus goes to the same college as Lucy, we could find two tuples with Name equal to “L. Van Pelt,” but with different student ID’s.

◆ **Example 8.10.** Let us develop a data structure for the

StudentId-Name-Address-Phone

relation of Fig. 8.2(a) that uses a hash table on StudentId as a primary index and a binary search tree as a secondary index for attribute Name. To simplify the presentation, we shall use a hash table with only two buckets for the primary structure, and the hash function we use is the remainder when the student ID is divided by 2. That is, the even ID's go into bucket 0 and the odd ID's into bucket 1.

```
typedef struct TUPLE *TUPLELIST;
struct TUPLE {
    int StudentId;
    char Name[30];
    char Address[50];
    char Phone[8];
    TUPLELIST next;
};

typedef TUPLELIST HASHTABLE[2];

typedef struct NODE *TREE;
struct NODE {
    char Name[30];
    TUPLELIST toTuple; /* really a pointer to a tuple */
    TREE lc;
    TREE rc;
};
```

Fig. 8.5. Types for a primary and a secondary index.

For the secondary index, we shall use a binary search tree, whose nodes store elements that are pairs consisting of the name of a student and a pointer to a tuple. The tuples themselves are stored as records, which are linked in a list to form one of the buckets of the hash table, and so the pointers to tuples are really pointers to records. Thus we need the structures of Fig. 8.5. The types `TUPLE` and `HASHTABLE` are the same as in Fig. 8.3, except that we are now using two buckets rather than 1009 buckets.

The type `NODE` is a binary tree node with two fields, `Name` and `toTuple`, representing the element at the node — that is, a student's name — and a pointer to a record where the tuple for that student is kept. The remaining two fields, `lc` and `rc`, are intended to be pointers to the left and right children of the node. We shall use alphabetic order on the last names of students as the “less than” order with which we compare elements at the nodes of the tree. The secondary index itself is a variable of type `TREE` — that is, a pointer to a node — and it takes us to the root of the binary search tree.

An example of the entire structure is shown in Fig. 8.6. To save space, the Address and Phone components of tuples are not shown. The *Li*'s indicate the memory locations at which the records of the primary index structure are stored.

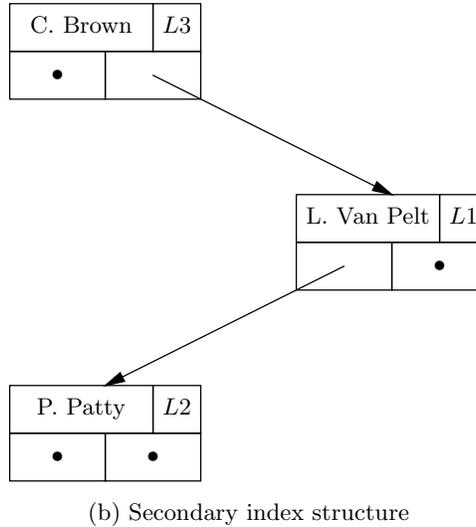
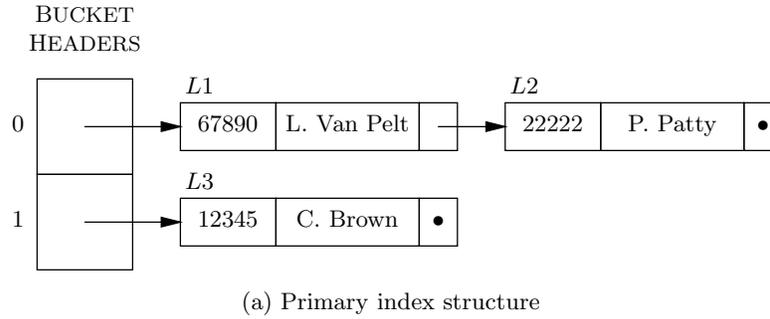


Fig. 8.6. Example of primary and secondary index structures.

Now, if we want to answer a query like “What is P. Patty’s phone number,” we go to the root of the secondary index, look up the node with `Name` field “P. Patty,” and follow the pointer in the `toTuple` field (shown as *L2* in Fig. 8.6). That gets us to the record for P. Patty, and from that record we can consult the `Phone` field and produce the answer to the query. ♦

Secondary Indexes on a Nonkey Field

It appeared that the attribute `Name` on which we built a secondary index in Example 8.10 was a key, because no name occurs more than once. As we know, however, it is possible that two students will have the same name, and so `Name` really is not a key. Nonkeyness, as we discussed in Section 7.9, does not affect the hash table data structure, although it may cause tuples to be distributed less evenly among buckets than we might expect.

A binary search tree is another matter, because that data structure does not handle two elements, neither of which is “less than” the other, as would be the case if we had two pairs with the same name and different pointers. A simple fix to the structure of Fig. 8.5 is to use the field `toTuple` as the header of a linked list of

Design IV: When Should We Create a Secondary Index?

The existence of secondary indexes generally makes it easier to look up a tuple, given the values of one or more of its components. However,

- ◆ Each secondary index we create costs time when we insert or delete information in the relation.
- ◆ Thus it makes sense to build a secondary index on only those attributes that we are likely to need for looking up data.

For example, if we never intend to find a student given the phone number alone, then it is wasteful to create a secondary index on the Phone attribute of the

StudentId-Name-Address-Phone

relation.

pointers to tuples, one pointer for each tuple with a given value in the Name field. For instance, if there were several P. Patty's, the bottom node in Fig. 8.6(b) would have, in place of $L2$, the header of a linked list. The elements of that list would be the pointers to the various tuples that had Name attribute equal to "P. Patty."

Updating Secondary Index Structures

When there are one or more secondary indexes for a relation, the insertion and deletion of tuples becomes more difficult. In addition to updating the primary index structure as outlined in Section 8.4, we may need to update each of the secondary index structures as well. The following methods can be used to update a secondary index structure for A when a tuple involving attribute A is inserted or deleted.

1. *Insertion.* If we insert a new tuple with value v in the component for attribute A , we must create a pair (v, p) , where p points to the new record in the primary structure. Then, we insert the pair (v, p) into the secondary index.
2. *Deletion.* When we delete a tuple that has value v in the component for A , we must first remember a pointer — call it p — to the tuple we have just deleted. Then, we go into the secondary index structure and examine all the pairs with first component v , until we find the one with second component p . That pair is then deleted from the secondary index structure.

EXERCISES

8.5.1: Show how to modify the binary search tree structure of Fig. 8.5 to allow for the possibility that there are several tuples in the StudentId-Name-Address-Phone relation that have the same student name. Write a C function that takes a name and lists all the tuples of the relation that have that name for the Name attribute.

8.5.2:** Suppose that we have decided to store the

StudentId-Name-Address-Phone

relation with a primary index on StudentId. We may also decide to create some secondary indexes. Suppose that all lookups will specify only one attribute, either Name, Address, or Phone. Assume that 75% of all lookups specify Name, 20% specify Address, and 5% specify Phone. Suppose that the cost of an insertion or a deletion is 1 time unit, plus 1/2 time unit for each secondary index we choose to build (e.g., the cost is 2.5 time units if we build all three secondary indexes). Let the cost of a lookup be 1 unit if we specify an attribute for which there is a secondary index, and 10 units if there is no secondary index on the specified attribute. Let a be the fraction of operations that are insertions or deletions of tuples with all attributes specified; the remaining fraction $1 - a$ of the operations are lookups specifying one of the attributes, according to the probabilities we assumed [e.g., $.75(1 - a)$ of all operations are lookups given a Name value]. If our goal is to minimize the average time of an operation, which secondary indexes should we create if the value of parameter a is (a) .01 (b) .1 (c) .5 (d) .9 (e) .99?

8.5.3: Suppose that the DMV wants to be able to answer the following types of queries efficiently, that is, much faster than by searching entire relations.

- i)* Given a driver's name, find the driver's license(s) issued to people with that name.
- ii)* Given a driver's license number, find the name of the driver.
- iii)* Given a driver's license number, find the registration numbers of the auto(s) owned by this driver.
- iv)* Given an address, find all the drivers' names at that address.
- v)* Given a registration number (i.e., a license plate), find the driver's license(s) of the owner(s) of the auto.

Suggest a suitable data structure for your relations from Exercise 8.3.2 that will allow all these queries to be answered efficiently. It is sufficient to suppose that each index will be built from a hash table and tell what the primary and secondary indexes are for each relation. Explain how you would then answer each type of query.

8.5.4*: Suppose that it is desired to find efficiently the pointers in a given secondary index that point to a particular tuple t in the primary index structure. Suggest a data structure that allows us to find these pointers in time proportional to the number of pointers found. What operations are made more time-consuming because of this additional structure?



8.6 Navigation among Relations

Until now, we have considered only operations involving a single relation, such as finding a tuple given values for one or more of its components. The power of the relational model can be seen best when we consider operations that require us to “navigate,” or jump from one relation to another. For example, we could answer the query “What grade did the student with ID 12345 get in CS101?” by working entirely within the Course-StudentId-Grade relation. But it would be more natural to ask, “What grade did C. Brown get in CS101?” That query cannot be answered within the Course-StudentId-Grade relation alone, because that relation uses student ID's, rather than names.

To answer the query, we must first consult the StudentId-Name-Address-Phone relation and translate the name “C. Brown” into a student ID (or ID’s, since it is possible that there are two or more students with the same name and different ID’s). Then, for each such ID, we search the Course-StudentId-Grade relation for tuples with this ID and with course component equal to “CS101.” From each such tuple we can read the grade of some student named C. Brown in course CS101. Figure 8.7 suggests how this query connects given values to the relations and to the desired answers.

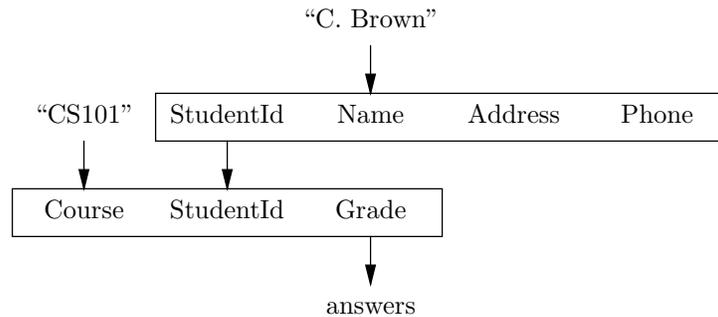


Fig. 8.7. Diagram of the query “What grade did C. Brown get in CS101?”

If there are no indexes we can use, then answering this query can be quite time-consuming. Suppose that there are n tuples in the

StudentId-Name-Address-Phone

relation and m tuples in the Course-StudentId-Grade relation. Also assume that there are k students with the name “C. Brown.” A sketch of the algorithm for finding the grades of this student or students in CS101, assuming there are no indexes we can use, is shown in Fig. 8.8.

- (1) **for** each tuple t in StudentId-Name-Address-Phone **do**
- (2) **if** t has “C. Brown” in its Name component **then begin**
- (3) let i be the StudentId component of tuple t ;
- (4) **for** each tuple s in Course-StudentId-Grade **do**
- (5) **if** s has Course component “CS101” and StudentId component i **then**
- (6) **print** the Grade component of tuple s ;
- (6) **end**
- (1) **end**

Fig. 8.8. Finding the grade of C. Brown in CS101.

Let us determine the running time of the program in Fig. 8.8. Starting from the inside out, the print statement of line (6) takes $O(1)$ time. The conditional statement of lines (5) and (6) also takes $O(1)$ time, since the test of line (5) is an $O(1)$ -time test. Since we assume that there are m tuples in the relation

Course-StudentId-Grade

the loop of lines (4) through (6) is iterated m times and thus takes $O(m)$ time in total. Since line (3) takes $O(1)$ time, the block of lines (3) to (6) takes $O(m)$ time.

Now consider the if-statement of lines (2) to (6). Since the test of line (2) takes $O(1)$ time, the entire if-statement takes $O(1)$ time if the condition is false and $O(m)$ time if it is true. However, we have assumed that the condition is true for k tuples and false for the rest; that is, there are k tuples t for which the name component is “C. Brown.” Since there is so much difference between the times taken when the condition is true and when it is false, we should be careful how we analyze the for-loop of lines (1) to (6). That is, instead of counting the number of times around the loop and multiplying by the greatest time the body can take, we shall consider separately the two outcomes of the test at line (2).

First, we go around the loop n times, because that is the number of different values of t . For the k tuples t on which the test at line (2) is true, we take $O(m)$ time each, or a total of $O(km)$ time. For the remaining $n - k$ tuples for which the test is false, we take $O(1)$ time per tuple, or $O(n - k)$ total. Since k is presumably much less than n , we can take $O(n)$ as a simpler but tight upper bound instead of $O(n - k)$. Thus the cost of the entire program is $O(n + km)$. In the likely case where $k = 1$, when there is only one student with the given name, the time required, $O(n + m)$, is proportional to the sum of the sizes of the two relations involved. If k is greater than 1, the time is greater still.

Speeding Navigation by Using Indexes

With the right indexes, we can answer the same query in $O(k)$ average time — that is, $O(1)$ time if k , the number of students with the name C. Brown, is 1. That makes sense, since all we must do is examine $2k$ tuples, k from each of the two relations. The indexes allow us to focus on the needed tuples in $O(1)$ average time for each tuple, if a hash table with the right number of buckets is used. If we have an index on Name for the StudentId-Name-Address-Phone relation, and an index on the combination of Course and StudentId for the Course-StudentId-Grade relation, then the algorithm for finding the grade of C. Brown in CS101 is as sketched in Fig. 8.9.

- (1) using the index on Name, find each tuple in the StudentId-Name-Address-Grade relation that has Name component “C. Brown”;
- (2) **for** each tuple t found in (1) **do begin**
- (3) let i be the StudentId component of tuple t ;
- (4) using the index on Course and StudentId in the Course-StudentId-Grade relation, find the tuple s with Course component “CS101” and StudentId component i ;
- (5) **print** the Grade component of tuple s ;
- end**

Fig. 8.9. Finding the grade of C. Brown in CS101 using indexes.

Let us assume that the index on Name is a hash table with about n buckets, used as a secondary index. Since n is the number of tuples in the

StudentId-Name-Address-Grade

relation, the buckets have $O(1)$ tuples each, on the average. Finding the bucket for Name value “C. Brown” takes $O(1)$ time. If there are k tuples with this name, it will take $O(k)$ time to find these tuples in the bucket and $O(1)$ time to skip over possible other tuples in the bucket. Thus line (1) of Fig. 8.9 takes $O(k)$ time on the average.

The loop of lines (2) through (5) is executed k times. Let us suppose we store the k tuples t that were found at line (1) in a linked list. Then the cost of going around the loop by finding the next tuple t or discovering that there are no more tuples is $O(1)$, as are the costs of lines (3) and (5). We claim that line (4) can also be executed in $O(1)$ time, and therefore the loop of lines (2) to (5) takes $O(k)$ time.

We analyze line (4) as follows. Line (4) requires the lookup of a single tuple, given its key value. Let us suppose that the Course-StudentId-Grade relation has a primary index on its key, {Course, StudentId}, and that this index is a hash table with about m buckets. Then the average number of tuples per bucket is $O(1)$, and therefore line (4) of Fig. 8.9 takes $O(1)$ time. We conclude that the body of the loop of lines (2) through (5) takes $O(1)$ average time, and thus the entire program of Fig. 8.9 takes $O(k)$ average time. That is, the cost is proportional to the number of students with the particular name we query about, regardless of the size of the relations involved.

Navigating over Many Relations

The same techniques that let us navigate efficiently from one relation to another also allow navigation involving many relations. For example, suppose we wanted to know, “Where is C. Brown 9AM Monday mornings?” Assuming that he is in some class, we can find the answer to this query by finding the courses C. Brown is taking, seeing whether any of them meet 9AM Mondays, and, if so, finding the room in which the course meets. Figure 8.10 suggests the navigation through relations from the given value “C. Brown” to the answer.

The following plan assumes that there is a unique student named C. Brown; if there is more than one, then we can get the rooms in which one or more of them are found at 9AM Mondays. It also assumes that this student has not registered for conflicting courses; that is, he is taking at most one course that meets at 9AM on Mondays.

1. Find the student ID for C. Brown, using the StudentId-Name-Address-Phone relation for C. Brown. Let this ID number be i .
2. Look up in the Course-StudentId-Grade relation all tuples with StudentId component i . Let $\{c_1, \dots, c_k\}$ be the set of Course values in these tuples.
3. In the Course-Day-Hour relation, look for tuples with Course component c_i , that is, one of the courses found in step (2). There should be at most one that has both “M” in the Day component and “9AM” in the Hour component.
4. If a course c is found in step (3), then look up in the Course-Room relation the room in which course c meets. That is where C. Brown will be found on Mondays at 9AM, assuming that he hasn’t decided to take a long weekend.

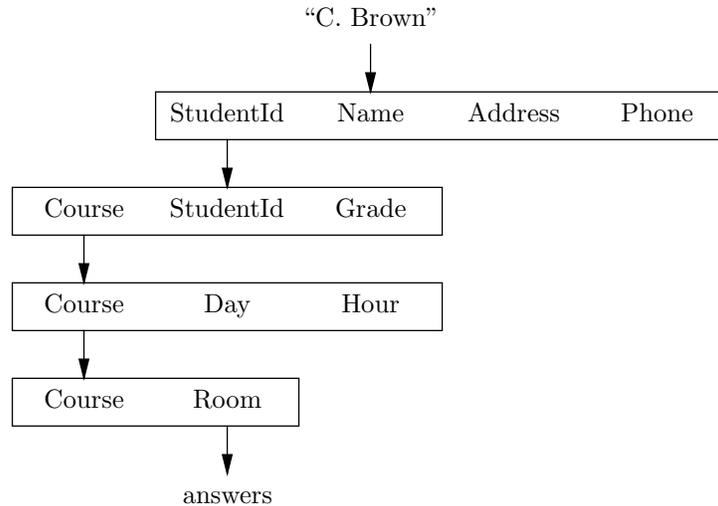


Fig. 8.10. Diagram of the query “Where is C. Brown at 9AM on Mondays?”

If we do not have indexes, then the best we can hope for is that we can execute this plan in time proportional to the sum of the sizes of the four relations involved. However, there are a number of indexes we can take advantage of.

- a) In step (1), we can use an index on the Name component of the StudentId-Name-Address-Phone relation to get the student ID of C. Brown in $O(1)$ average time.
- b) In step (2), we can take advantage of an index on the StudentId component of Course-StudentId-Grade to get in $O(k)$ time all the courses C. Brown is taking, if he is taking k courses.
- c) In step (3), we can take advantage of an index on Course in the Course-Day-Hour relation to find all the meetings of the k courses from step (2) in average time proportional to the sum of the numbers of meetings of these courses. If we assume that no course meets more than five times a week, then there are at most $5k$ tuples, and we can find them in $O(k)$ average time. If there is no index on Course for this relation, but there is an index on Day and/or Hour, we can take some advantage of such an index, although we may look at far more than $O(k)$ tuples, depending on how many courses there are that meet on Monday or that meet at 9AM on some day.
- d) In step (4), we can take advantage of an index on Course for the Course-Room relation. In that case, we can retrieve the desired room in $O(1)$ average time.

We conclude that, with all the right indexes, we can answer this very complicated query in $O(k)$ average time. Since k , the number of courses taken by C. Brown, can be assumed small — say, 5 or so — this amount of time is normally quite small,

Summary: Fast Access to Relations

It is useful to review how our capability to get answers from relations has grown. We began in Section 7.8 by using a hash table, or another structure such as a binary search tree or a (generalized) characteristic vector, to implement functions, which in the context of this chapter are binary relations whose domain is a key. Then, in Section 7.9, we saw that these ideas worked even when the domain was not a key, as long as the relation was binary.

In Section 8.4, we saw that there was no requirement that the relation be binary; we could regard all attributes that are part of the key as a single “domain” set, and all the other attributes as a single “range” set. Further, we saw in Section 8.4 that the domain did not have to be a key.

In Section 8.5 we learned that we could use more than one index structure on a relation to allow fast access based on attributes that are not part of the domain, and in Section 8.6 we saw that it is possible to use a combination of indexes on several relations to perform complex retrievals of information in time proportional to the number of tuples we actually visit.

and in particular is independent of the sizes of any of the relations involved.

EXERCISES

8.6.1: Suppose that the Course-StudentId-Grade relation in Fig. 8.9 did not have an index on Course-StudentId pairs, but rather had an index on Course alone. How would that affect the running time of Fig. 8.9? What if the index were only on StudentId?

8.6.2: Discuss how the following queries can be answered efficiently. In each case, state what assumptions you make about the number of elements in intermediate sets (e.g., the number of courses taken by C. Brown), and also state what indexes you assume exist.

- a) Find all the prerequisites of the courses taken by C. Brown.
- b) Find the phone numbers of all the students taking a course that meets in Turing Aud.
- c) Find the prerequisites of the prerequisites of CS206.

8.6.3: Assuming no indexes, how much time would each of the queries in Exercise 8.6.2 take, as a function of the sizes of the relations involved, assuming straightforward iterations over all tuples, as in the examples of this section?



8.7 An Algebra of Relations

In Section 8.6 we saw that a query involving several relations can be quite complicated. It is useful to express such queries in language that is much “higher-level” than C, in the sense that the query expresses what we want (e.g., all tuples with Course component equal to “CS101”) without having to deal with issues such as

lookup in indexes, as a C program would. For this purpose, a language called *relational algebra* has been developed.

Like any algebra, relational algebra allows us to rephrase queries by applying algebraic laws. Since complicated queries often have many different sequences of steps whereby their answer can be obtained from the stored data, and since different algebraic expressions represent different sequences of steps, relational algebra provides an excellent example of algebra as a design theory. In fact, the improvement in efficiency made possible by transforming expressions of relational algebra is arguably the most striking example of the power of algebra that we find in computer science. The ability to “optimize” queries by algebraic transformation is the subject of Section 8.9.

Operands of Relational Algebra

In relational algebra, the operands are relations. As in other algebras, operands can be either constants — specific relations in this case — or variables representing unknown relations. However, whether a variable or a constant, each operand has a specific scheme (list of attributes naming its columns). Thus a constant argument might be shown as

Constant arguments

<i>A</i>	<i>B</i>	<i>C</i>
0	1	2
0	3	4
5	2	3

This relation has scheme $\{A, B, C\}$, and it has three tuples, $(0, 1, 2)$, $(0, 3, 4)$, and $(5, 2, 3)$.

Variable arguments

A variable argument might be represented by $R(A, B, C)$, which denotes a relation called R , whose columns are named A , B , and C but whose set of tuples is unknown. If the scheme $\{A, B, C\}$ for R is understood or irrelevant, we can just use R as the operand.

Set Operators of Relational Algebra

The first three operators we shall use are common set operations: union, intersection, and set difference, which were discussed in Section 7.3. We place one requirement on the operands of these operators: the schemes of the two operands must be the same. The scheme of the result is then naturally taken to be the scheme of either argument.

Union, intersection, and difference

◆ **Example 8.11.** Let R and S be the relations of Fig. 8.11(a) and (b), respectively. Note that both relations have the scheme $\{A, B\}$. The union operator produces a relation with each tuple that appears in either R or S , or both. Note that since relations are sets, they can never have two or more copies of the same tuple, even though a tuple appears in both R and S , as does the tuple $(0, 1)$ in this example. The relation $R \cup S$ is shown in Fig. 8.11(c).

The intersection operator produces the relation that has those tuples appearing in both operands. Thus the relation $R \cap S$ has only the tuple $(0, 1)$, as shown in Fig. 8.11(d). The set difference produces a relation with those tuples in the first relation that are not also in the second. The relation $R - S$, shown in Fig. 8.11(e),

A	B	A	B
0	1	0	1
2	3	4	5

(a) R (b) S

A	B	A	B	A	B
0	1	A	B	A	B
2	3	0	1	2	3
4	5				

(c) $R \cup S$ (d) $R \cap S$ (e) $R - S$

Fig. 8.11. Examples of operations of relational algebra.

has the tuple $(2, 3)$ of R , because that tuple is not in S , but does not have the tuple $(0, 1)$ of R , because that tuple is also in S . ♦

The Selection Operator

The other operators of relational algebra are designed to perform the kinds of actions we have studied in this chapter. For example, we have frequently wanted to extract from a relation tuples meeting certain conditions, such as all tuples from the

Course-StudentId-Grade

relation that have Course component “CS101.” For this purpose, we use the *selection* operator. This operator takes a single relation as operand, but also has a conditional expression as a “parameter.” We write the selection operator $\sigma_C(R)$, where σ (Greek lower-case sigma) is the symbol for selection, C is the condition, and R is the relation operand. The condition C is allowed to have operands that are attributes from the scheme of R , as well as constants. The operators allowed in C are the usual ones for C conditional expressions, that is, arithmetic comparisons and the logical connectives.

The result of this operation is a relation whose scheme is the same as that of R . Into this relation we put every tuple t of R such that condition C becomes true when we substitute for each attribute A the component of tuple t in the column for A .

- ♦ **Example 8.12.** Let CSG stand for the Course-StudentId-Grade relation of Fig. 8.1. If we want those tuples that have Course component “CS101,” we can write the expression

$$\sigma_{\text{Course}=\text{“CS101”}}(CSG)$$

The result of this expression is a relation with the same scheme as CSG , that is, $\{\text{Course}, \text{StudentId}, \text{Grade}\}$, and the set of tuples shown in Fig. 8.12. That is, the condition becomes true only for those tuples where the Course component is “CS101.” For then, when we substitute “CS101” for Course, the condition becomes “CS101” = “CS101.” If the tuple has any other value, such as “EE200”, in the Course component, we get an expression like “EE200” = “CS101,” which is false. ♦

Course	StudentId	Grade
CS101	12345	A
CS101	67890	B
CS101	33333	A-

Fig. 8.12. Result of expression $\sigma_{\text{Course}=\text{“CS101”}}(CSG)$.

The Projection Operator

Whereas the selection operator makes a copy of the relation with some rows deleted, we often want to make a copy in which some columns are eliminated. For that purpose we have the *projection operator*, represented by the symbol π . Like selection, the projection operator takes a single relation as argument, and it also takes a parameter, which is a list of attributes, chosen from the scheme of the relation that is the argument.

If R is a relation with set of attributes $\{A_1, \dots, A_k\}$, and (B_1, \dots, B_n) is a list of some of the A 's, then $\pi_{B_1, \dots, B_n}(R)$, the *projection of R onto attributes B_1, \dots, B_n* , is the set of tuples formed as follows. Take each tuple t in R , and extract its components in attributes B_1, \dots, B_n ; say these components are b_1, \dots, b_n , respectively. Then add the tuple (b_1, \dots, b_n) to the relation $\pi_{B_1, \dots, B_n}(R)$. Note that two or more tuples of R may have the same components in all of B_1, \dots, B_n . If so, only one copy of the projection of those tuples goes into $\pi_{B_1, \dots, B_n}(R)$, since that relation, like all relations, cannot have more than one copy of any tuple.

- ♦ **Example 8.13.** Suppose we wanted to see only the student ID's for the students who are taking CS101. We could apply the same selection as in Example 8.12, which gives us all the tuples for CS101 in the CSG relation, but we then must project out the course and grade; that is, we project onto StudentId alone. The expression that performs both operations is

$$\pi_{\text{StudentId}}(\sigma_{\text{Course}=\text{“CS101”}}(CSG))$$

The result of this expression is the relation of Fig. 8.12 projected onto its StudentId component — that is, the unary relation of Fig. 8.13. ♦

StudentId
12345
67890
33333

Fig. 8.13. Students taking CS101.

Joining Relations

Finally, we need a way to express the idea that two relations are connected, so that we can navigate from one to the other. For this purpose, we use the *join* operator, which we denote \bowtie .³ Suppose we have two relations R and S , with sets of attributes (schemes) $\{A_1, \dots, A_n\}$ and $\{B_1, \dots, B_m\}$, respectively. We pick one attribute from each set — say, A_i and B_j — and these attributes become parameters of the join operation with arguments R and S .

The join of R and S , written $R \bowtie_{A_i=B_j} S$, is formed by taking each tuple r from R and each tuple s from S and comparing them. If the component of r for A_i equals the component of s for B_j , then we form one tuple from r and s ; otherwise, no tuple is created from the pairing of r and s . We form a tuple from r and s by taking the components of r and following them by all the components of s , but omitting the component for B_j , which is the same as the A_i component of r anyway.

The relation $R \bowtie_{A_i=B_j} S$ is the set of tuples formed in this manner. Note that there could be no tuples in this relation, if no value appearing in the A_i column of R also appeared in the B_j column of S . At the other extreme, every tuple of R could have the same value in the A_i component, and this component could also appear in the B_j component of every tuple in S . Then, the number of tuples in the join would be the product of the number of tuples in R and the number in S , since every pair of tuples would match. Generally, the truth lies somewhere between these extremes; each tuple of R pairs with some but not all of the tuples of S .

The scheme of the joined relation is

$$\{A_1, \dots, A_n, B_1, \dots, B_{j-1}, B_{j+1}, \dots, B_m\}$$

that is, the set of all attributes of R and S except for B_j . However, there could be two occurrences of the same name on this list, if one of the A 's was the same as one of the B 's (other than B_j , which is not an attribute of the join). If that is the case, we shall insist that one of the pair of identical attributes be renamed.

◆ **Example 8.14.** Suppose we want to perform some operation connecting the

Course-Day-Hour

relation (which we abbreviate to CDH), and the Course-Room relation (CR). For instance, we might want to know at what times each room is occupied by some course. To answer that query, we must pair each tuple from CR with each tuple from CDH , provided that the Course components of the two tuples are the same —

³ The “join” that we describe here is less general than that normally found in relational algebra but will serve to get the flavor of the operator without going into all the complexities of the subject.

that is, if the tuples are talking about the same course. Thus if we join CR with CDH , requiring equality of the two Course attributes, we shall get a relation with scheme

$$\{\text{Course, Room, Day, Hour}\}$$

that contains each tuple (c, r, d, h) such that (c, r) is a tuple of CR and (c, d, h) is a tuple of CDH . The expression defining this relation is

$$CR \underset{\text{Course=Course}}{\bowtie} CDH$$

and the value of the relation produced by this expression, assuming that the relations have the tuples found in Fig. 8.2, is as shown in Fig. 8.14.

Course	Room	Day	Hour
CS101	Turing Aud.	M	9AM
CS101	Turing Aud.	W	9AM
CS101	Turing Aud.	F	9AM
EE200	25 Ohm Hall	Tu	10AM
EE200	25 Ohm Hall	W	1PM
EE200	25 Ohm Hall	Th	10AM

Fig. 8.14. Join of CR and CDH on $\text{Course} = \text{Course}$.

To see how the relation of Fig. 8.14 is constructed, consider the first tuple of CR , which is (CS101, Turing Aud.). We examine the tuples of CDH for those that have the same Course value, that is, “CS101.” In Fig. 8.2(c), we find that the first three tuples match, and from each of them, we construct one of the first three tuples of Fig. 8.14. For example, the first tuple of CDH , which is (CS101, M, 9AM), joins with tuple (CS101, Turing Aud.) to create the first tuple of Fig. 8.14. Notice how that tuple agrees with each of the two tuples from which it is constructed.

Similarly, the second tuple of CR , (EE200, 25 Ohm Hall), shares a common Course component with each of the last three tuples of CDH . These three pairings give rise to the last three tuples of Fig. 8.14. The last tuple of CR ,

(PH100, Newton Lab.)

does not have the same Course component as any tuple of CDH . Thus that tuple does not contribute anything at all to the join. ♦

Natural Join

When we join two relations R and S , it is common that the attributes we equate have the same name. If, in addition, R and S have no other attribute names in common, then we can omit the parameter of the join and simply write $R \bowtie S$. Such a join is called a *natural join*.

For instance, the join in Example 8.14 is a natural join. The equated attributes are both called Course, and the remaining attributes of CR and CDH all have distinct names. Thus we could have written this join simply as $CR \bowtie CDH$.

Expression Trees for Relational Algebra Expressions

Just as we draw expression trees for arithmetic expressions, we can represent a relational algebra expression as a tree. The leaves are labeled by operands, that is, by specific relations or variables representing relations. Each interior node is labeled by an operator, including the parameter of the operator if it is a selection, projection, or join (except a natural join, which needs no parameter). The children of each interior node N are the node or nodes representing the operands to which the operator at node N is applied.

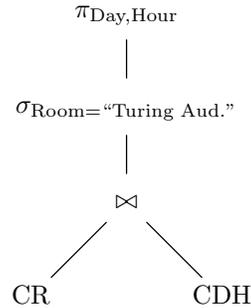


Fig. 8.15. Expression tree in relational algebra.

◆ **Example 8.15.** Building on Example 8.14, suppose we wanted to see not the entire relation $CR \bowtie CDH$, but just the Day-Hour pairs during which Turing Aud. is occupied by some course. Then we need to take the relation of Fig. 8.14 and

1. Select for those tuples having Room component “Turing Aud.,” and
2. Project onto attributes Day and Hour.

The expression that performs the join, selection, and projection, in that order, is

$$\pi_{\text{Day, Hour}}(\sigma_{\text{Room}=\text{“Turing Aud.”}}(CR \bowtie CDH))$$

Alternatively, we could display this expression as the tree shown in Fig. 8.15. The relation computed at the join node appeared in Fig. 8.14. The relation for the selection node is the first three tuples in Fig. 8.14, because these have “Turing Aud.” in their Room component. The relation for the root of the expression is shown in Fig. 8.16, that is, the Day and Hour components of the latter three tuples. ◆

Day	Hour
M	9AM
W	9AM
F	9AM

Fig. 8.16. Result of expression in Fig. 8.15.

SQL, a Language Based on Relational Algebra

Many modern database systems use a language called SQL (Structured Query Language) for expressing queries. While a complete guide to that language is beyond the scope of this book, we can give the reader a feel for SQL with a few examples.

```
SELECT StudentId
FROM CSG
WHERE Course = "CS101"
```

is SQL's way of expressing the query of Example 8.13, that is,

$$\pi_{\text{StudentId}}(\sigma_{\text{Course}=\text{"CS101"}}(\text{CSG}))$$

The **FROM**-clause indicates the relation to which the query is applied. The **WHERE**-clause gives the condition of the selection, and the **SELECT**-clause gives the list of attributes onto which the answer is projected. (It is unfortunate that the keyword **SELECT** in SQL corresponds not to the relational algebra operator called "selection" but to the operator called "projection.")

For a more complicated example, we can express the query of Example 8.15, which is $\pi_{\text{Day,Hour}}(\sigma_{\text{Room}=\text{"Turing Aud."}}(\text{CR} \bowtie \text{CDH}))$, by the SQL program

```
SELECT Day, Hour
FROM CR, CDH
WHERE CR.Course = CDH.Course AND Room = "Turing Aud."
```

Here, the **FROM**-clause tells us we are going to join the two relations *CR* and *CDH*. The first part of the **WHERE**-clause is the join condition; it says that the *Course* attribute of *CR* must equal the *Course* attribute in *CDH*. The second part of the **WHERE**-clause is the selection condition. The **SELECT**-clause gives us the attributes in the projection.

EXERCISES

8.7.1: Express the queries of Exercise 8.4.2(a, b, c) in relational algebra. Assume that what is wanted as an answer is the complete tuple(s).

8.7.2: Repeat Exercise 8.7.1, assuming that what is wanted is only the components that have a * in the specification.

8.7.3: Express the queries of Exercise 8.6.2(a, b, c) in relational algebra. Note that in part (c), you will have to rename some attributes in order to take the join of a relation with itself.

8.7.4: Express the query "Where is C. Brown at 9AM on Monday?" in relational algebra. The discussion at the end of Section 8.6 should indicate the joins necessary to answer this query.

8.7.5: Draw expression trees for the queries of Exercise 8.7.2(a) through (c), Exercise 8.7.3(a) through (c), and Exercise 8.7.4.



8.8 Implementing Relational Algebra Operations

Using the right data structures and algorithms for relational algebra operations can speed up database queries. In this section, we shall consider some of the simpler and more common strategies for implementing relational algebra operations.

Implementing Union, Intersection, and Difference

The three basic set operations can be implemented in the same way for relations as for sets. We can take the union of two sets or relations by sorting both sets and merging, as discussed in Section 7.4. The intersection and difference can be taken by a similar technique. If the relations have n tuples each, it takes $O(n \log n)$ time to sort them and $O(n)$ time to do the merging, or $O(n \log n)$ total.

However, there are several other ways we could take the union of relations R and S , and these are sometimes more efficient. First, we might not worry about eliminating the duplicate copy of a tuple that appears in both R and S . We could construct $R \cup S$ by making a copy of R , say, as a linked list, and then appending all the tuples of S , without checking whether a tuple of S is also in R . This operation can be done in time proportional to the sum of the sizes of R and S . The drawback is that the result is not, strictly speaking, the union, since it can have duplicate tuples. However, perhaps the presence of duplicates does not matter, because they are expected to be rare. Or, we might find it more convenient to eliminate the duplicates at a later stage, such as by sorting after taking the union of several more relations.

Another option is to use an index. For example, suppose that S has an index on attribute A , and that this attribute is a key for S . Then we can take the union $R \cup S$ by starting with the tuples of S , and examining each tuple t of R , in its turn. We find the value of t in its component A — let us call it a — and use the index to look up the tuple of S that has the value a in its A -component. If this tuple in S is identical to t , then we do not add t to the union a second time; but if there is no tuple with key value a in S , or if the tuple with this key value differs from t , then we add t to the union.

If the index gives us $O(1)$ average time to look up a tuple, given its key value, then this method takes average time proportional to the sum of the sizes of R and S . Moreover, the resulting relation will have no duplicates, as long as neither R nor S has duplicates.

Implementing Projection

In principle, when we perform a projection, we have no choice but to run through every tuple and make a copy that omits the components corresponding to attributes not on the projection list. Indexes do not help us at all. Moreover, after we compute the projection of each tuple, we may find that we are left with many duplicates.

For example, suppose we have a relation R with scheme (A, B, C) and we compute $\pi_{A,B}(R)$. Even though R cannot have tuples that agree on all of A , B , and C , it may have many tuples with the same values for attributes A and B but different values for C . Then all these tuples will yield the same tuple in the projection.

Thus, after we compute a projection such as $S = \pi_L(R)$, for some relation R and list of attributes L , we must eliminate duplicates. For example, we could sort S and then run through the tuples in the sorted order. Any tuple that is the same

as the previous tuple in the order will be eliminated. Another way to eliminate duplicates is to treat the relation S as an ordinary set. Each time we generate a tuple by projecting a tuple of R onto the attributes in the list L , we insert it into the set. As with all insertions into a set, if the element inserted is already there, we do nothing. A structure such as a hash table will serve adequately to represent the set S of tuples generated by the projection.

To sort the relation S before eliminating duplicates requires $O(n \log n)$ time if there are n tuples in the relation R . If we instead hash tuples of S as we generate them and we use a number of buckets proportional to n , then the entire projection will take $O(n)$ time, on the average. Thus hashing is normally slightly better than sorting.

Implementing Selection

When we perform a selection $S = \sigma_C(R)$ and there are no indexes on R , then we have no choice but to run through all the tuples of R to apply the condition C . Regardless of how we perform the selection, we know that there can be no duplicates in the result S , as long as R has no duplicates.

However, if there are indexes on R , then we can often take advantage of one of them to home in on the tuples that meet the condition C , and we can thus avoid looking at most or all of the tuples that do not meet condition C . The simplest situation occurs when condition C is of the form $A = b$, where A is an attribute of R and b is a constant. If R has an index on A , then we can retrieve all the tuples that meet this condition by looking up b in the index.

If condition C is the logical AND of several conditions, then we can use any one of them to look up tuples using an index, and then check the retrieved tuples to see which ones meet the remaining conditions. For example, suppose condition C is

$$(A = a) \text{ AND } (B = b)$$

Then we have the choice of using an index on A or an index on B , if either or both exists. Suppose that there is an index on B , and either there is no index on A or we prefer to use the index on B . Then we get all the tuples of R that have the value b in their B component. Each of these tuples that has a in the A component belongs in the relation S , the result of the selection; other retrieved tuples do not. The time taken for the selection is proportional to the number of tuples with B value b , which generally lies somewhere between the number of tuples in R and the number of tuples in the answer, S .

Implementing Join

Suppose we want to take the natural join of relation R with scheme $\{A, B\}$ and relation S with scheme $\{B, C\}$. Suppose also that the join is the natural join, with equality between the B attributes of the two relations.⁴ How we perform this join depends on what indexes on attribute B we can find. The issues are similar to those discussed in Section 8.6, when we considered how to navigate among relations, because the join is the essence of navigation.

There is an obvious and slow way to compute the join, called *nested-loop join*. We compare every tuple of one relation with every tuple of the other relation, as

Nested-loop join

⁴ We show for each relation only one attribute (A and C , respectively) that is not involved in the join, but the ideas mentioned here clearly carry over to relations with many attributes.

```

for each tuple  $r$  in  $R$  do
  for each tuple  $s$  in  $S$  do
    if  $r$  and  $s$  agree on their  $B$  attributes then
      print the tuple agreeing with  $r$  and  $s$ 
      on attributes  $A$ ,  $B$ , and  $C$ ;

```

Index-join However, there are several more efficient ways to take a join. One is an *index-join*. Suppose S has an index on B . Then we can visit every tuple t of R and find its B component — say, b . We look up b in the index for S , and thus obtain all the tuples that match t in B -values.

Similarly, if R has an index on B , we can run through the tuples of S . For each such tuple, we look up the corresponding tuples of R by using the B index of R . If both R and S have indexes on B , we can choose either one to use. As we shall see, it makes a difference in the amount of time the join takes.

Sort-join If there are no indexes on B , we can still do better than a nested-loop join by a technique called *sort-join*. We begin by merging the tuples of R and S , but reorganizing them so that the B components are first in all tuples, and tuples have an extra component that is either R (when the tuple comes from relation R) or S (when the tuple comes from S). That is, a tuple (a, b) from R becomes (b, a, R) , while tuple (b, c) from S becomes (b, c, S) .

We sort the merged list of tuples on the first, or b , component. Now, all the tuples of both relations that join because of a common B value are consecutive in the ordering, although tuples from the two relations may be mixed.⁵ We visit the tuples with each given B value in turn, by going down the sorted list. When we come to the tuples with B value b , we can pair all the tuples from R with those from S . Since these tuples all have the same B value, they all join, and the time taken to produce the tuples of the joined relation is proportional to the number of tuples produced, except in the case that there are no tuples from R or no tuples from S . In the latter case, we must still take time proportional to the number of tuples with B value b , just to examine each once and skip over them on the sorted list.

◆ **Example 8.16.** Suppose we want to join the relation CDH from Fig. 8.2(c) with the relation CR from Fig. 8.2(d). Here, Course plays the role of attribute B , Day and Hour together play the role of A , and Room is C . The six tuples from CDH and the three from CR are first padded with the name of the relation. No reordering of components is necessary, because Course is first in both relations. When we compare tuples, we first compare the Course components, using lexicographic order to determine which course name comes first in the order. If there is a tie, that is, if the course names are the same, we compare the last components, where we take CDH to precede CR . If there is still a tie, we can allow either tuple to precede the other.

Then one sorted order of the tuples will be as shown in Fig. 8.17. Note that this list is not a relation, because it has tuples of varying lengths. However, it does group the tuples for CS101 and the tuples for EE200, so that we can easily take the

⁵ We could arrange while sorting that the last component — that is, the relation name — be taken into account, so that a tuple with a given B value from relation R is deemed to precede a tuple with the same B value from S . Then, the tuples with a common B value would appear with the tuples from R first, and then the tuples from S .

CS101	M	9AM	<i>CDH</i>
CS101	W	9AM	<i>CDH</i>
CS101	F	9AM	<i>CDH</i>
CS101	Turing Aud.	<i>CR</i>	
EE200	Tu	10AM	<i>CDH</i>
EE200	W	1PM	<i>CDH</i>
EE200	F	10AM	<i>CDH</i>
EE200	25 Ohm Hall	<i>CR</i>	
PH100	Newton Lab.	<i>CR</i>	

Fig. 8.17. Sorted list of tuples from *CDH* and *CR*.

join of these groups of tuples. ♦

Comparison of Join Methods

Suppose we join the relation R , with scheme $\{A, B\}$, and the relation S , with scheme $\{B, C\}$, and let R and S have r tuples and s tuples, respectively. Also, let the number of tuples in the join be m . Remember that m could be as large as rs , if each tuple of R joins with each tuple of S (because they all have the same B value), but m could also be as small as 0, if no tuple of R has a B value in common with any tuple of S . Finally, let us assume that we can look up any value in any index in $O(1)$ time, on the average, as we could if the index were a hash table with a sufficiently large number of buckets.

Every method for joining will take at least $O(m)$ time, just to produce the output. However, some methods will take more. If we use nested-loop join, it takes time rs to perform the comparisons. Since $m \leq rs$, we can neglect the time to produce the output and say that the cost of pairing all tuples is $O(rs)$.

On the other hand, we could sort the relations. If we use an algorithm like merge sort to sort the combined list of $r + s$ tuples, the time required is

$$O((r + s) \log(r + s))$$

Building the output tuples from adjacent tuples in the sorted list will take $O(r + s)$ time to examine the list, plus $O(m)$ time to produce the output. The time to sort dominates the $O(r + s)$ term, but the $O(m)$ cost to produce the output can be greater or less than the sorting time. Thus we must include both terms in the running time of the algorithm that joins by sorting; this running time is thus

$$O(m + (r + s) \log(r + s))$$

Since m is never greater than rs , and $(r + s) \log(r + s)$ is greater than rs only in some unusual cases (for example, when r or s is 0), we conclude that sort-join is generally faster than nested-loop join.

Now suppose we have an index on B in the relation S . It takes $O(r)$ time to look at each tuple of R and look up its B value in the index. To this time we must add the $O(m)$ cost of retrieving the matching tuples for the various B values and of producing the output tuples. Since m can be greater than or less than r , the expression for the cost of this index-join is $O(m + r)$. Similarly, if there is an index on B for relation R , we can perform the index-join in $O(m + s)$ time. Since both r and s are smaller than $(r + s) \log(r + s)$, except in some unusual situations such

as $r + s \leq 1$, the running time of index-join is smaller than that of sort-join. Of course, we need an index on one of the attributes involved in the join, if we are to do an index-join, while a sort-join can be done on any relations.

EXERCISES

8.8.1: Suppose that the StudentId-Name-Address-Phone relation (*SNAP*) of Fig. 8.2(a) is stored with a primary index on StudentId (the key) and a secondary index on Phone. How would you compute most efficiently the answer to the query $\sigma_C(SNAP)$ if C were

- a) StudentId = 12345 AND Address \neq "45 Kumquat Blvd"?
- b) Name = "C. Brown" AND Phone = 555-1357?
- c) Name = "C. Brown" OR Phone = 555-1357?

8.8.2: Show how to sort-join the relations *CSG* from Fig. 8.1 and *SNAP* from Fig. 8.2(a) by sorting the merged list of tuples as in Example 8.16. Assume the natural join, or equality on the StudentId components, is wanted. Show the result of the sort, analogous to Fig. 8.17, and give the tuples in the result of the join.

8.8.3*: Suppose that we join relations R and S , each with n tuples, and the result has $O(n^{3/2})$ tuples. Write formulas for the big-oh running time, as a function of n , for the following techniques for taking the join:

- a) Nested-loop join
- b) Sort-join
- c) Index-join, using an index on the join attribute of R
- d) Index-join, using an index on the join attribute of S

8.8.4*: We proposed taking the union of two relations by using an index on an attribute A that was a key for one of the relations. Is the method a reasonable way to take a union if the attribute A that has an index is not a key?

8.8.5*: Suppose we want to compute (a) $R \cap S$ (b) $R - S$ using an index on attribute A for one of R and S . Can we obtain running time close to the sum of the sizes of the two relations?

8.8.6: If we project a relation R onto a set of attributes that contains a key for R , do we need to eliminate duplicates? Why?



8.9 Algebraic Laws for Relations

As with other algebras, by transforming expressions we often have the opportunity to "optimize" expressions. That is, we can take an expression that is expensive to evaluate and turn it into an equivalent expression whose evaluation has a lower cost. While transformations to arithmetic or logical expressions sometimes save a few operations, the right transformations applied to expressions of relational algebra can save orders of magnitude in the time it takes to evaluate the expression. Because of the tremendous difference between the running times of optimized and unoptimized relational algebra expressions, our ability to optimize such expressions is essential if programmers are going to program in very high-level languages, like the language SQL that we mentioned in Section 8.7.

Laws Involving Union, Intersection, and Difference

Section 7.3 covered the principal algebraic laws for union, intersection, and difference of sets. They apply to relations as a special case, although the reader should bear in mind the requirement of the relational model that the schemes of the relations involved in these operations be the same.

Laws Involving Join

Commutativity of join

In one sense, the join operator is commutative, and in another sense it is not. Suppose we take the natural join $R \bowtie S$, where R has attributes A and B while S has attributes B and C . Then the columns of the scheme for $R \bowtie S$ are A , B , and C , in that order. If we take $S \bowtie R$ instead, we get essentially the same tuples, but the order of the columns is B , C , and then A . Thus if we insist that order of columns matters, join is not commutative. However, if we accept that a relation, with its columns permuted along with their column names, is really the same relation, then we can consider the join commutative; that point of view will be adopted here.

Nonassociativity of join

The join operator does not always obey the associative law. For example, suppose relations R , S , and T have schemes $\{A, B\}$, $\{B, C\}$, and $\{A, D\}$, respectively. Suppose we take the natural join $(R \bowtie S) \bowtie T$, where we first equate the B components of R and S and then equate the A component of the result with the A component of relation T . If we associate from the right instead of the left, we get $R \bowtie (S \bowtie T)$. Relations S and T have schemes $\{B, C\}$ and $\{A, D\}$, respectively. There is no pair of attributes we can choose to equate that will achieve the same effect as the natural join.

However, there are some conditions under which the associative law holds for \bowtie . We leave it as an exercise for the reader to show that

$$((R \bowtie_{A=B} S) \bowtie_{C=D} T) \equiv (R \bowtie_{A=B} (S \bowtie_{C=D} T))$$

whenever A is an attribute of R , B and C are different attributes of S , and D is an attribute of T .

Laws Involving Selection

The most useful laws of relational algebra involve the selection operator. If the selection condition requires that a specified component have a certain value, as is often the case in practice, then the relation that is the result of the selection will tend to have many fewer tuples than the relation to which the selection is applied. Since operations in general take less time if they are applied to smaller relations, it is extremely advantageous to apply a selection as soon as we can. In algebraic terms, we apply selections early by using a law that lets a selection pass down the expression tree, below other operators.

An example of such a law is

$$(\sigma_C(R \bowtie S)) \equiv (\sigma_C(R) \bowtie S)$$

which holds provided that all attributes mentioned in condition C are attributes of relation R . Similarly, if all attributes mentioned by C are attributes of S , we can push the selection down to S , using the law

$$(\sigma_C(R \bowtie S)) \equiv (R \bowtie \sigma_C(S))$$

Pushing selections

Either law is referred to as *selection pushing*.

When we have a complex condition in a selection, sometimes we can push part of it one way and part of it the other way, through a join. In order to split a selection into two or more parts, we need the law

$$\sigma_{C \text{ AND } D}(R) \equiv \sigma_C(\sigma_D(R))$$

Selection splitting

Notice that we can only split a condition into two parts — C and D here — if the parts are connected by AND. Intuitively, when we select for the AND of two conditions C and D , we can either examine each tuple of the relation R and see whether the tuple satisfies both C and D , or we can examine all the tuples of R , selecting those that satisfy D , and then examine the tuples that satisfy D to see which of them satisfy C . We call this law *selection splitting*.

Another necessary law is the commutativity of selections. If we apply two selections to a relation, it does not matter in what order we apply the selections; the selected tuples will still be the same. Formally, we may write

Commutativity of selection

$$\sigma_C(\sigma_D(R)) \equiv \sigma_D(\sigma_C(R))$$

for any conditions C and D .

◆ **Example 8.17.** Let us take up the complex query that we first considered in Section 8.6: “Where is C. Brown 9 AM on Mondays?” This query involves navigating over the four relations

1. CSG (Course-StudentId-Grade),
2. $SNAP$ (StudentId-Name-Address-Phone),
3. CDH (Course-Day-Hour), and
4. CR (Course-Room).

To get an algebraic expression for the query, we can start by taking the natural join of all four relations. That is, we connect CSG and $SNAP$ by equating the StudentId components. Think of this operation as extending each

Course-StudentId-Grade

tuple by adding components for the name, address, and phone of the student mentioned in the tuple. Of course, we wouldn’t want to store data this way, because it forces us to repeat the information about each student once for each course the student takes. However, we are not storing this data, but just designing an expression to compute it.

To the result of $CSG \bowtie SNAP$ we join CDH , by equating on the Course components. That join has the effect of taking each CSG tuple (already extended by the student information), making one copy for each meeting of the course, and extending each tuple by one of the possible Day and Hour values. Finally, we join the result of $(CSG \bowtie SNAP) \bowtie CDH$ with the CR relation, equating Course components, which has the effect of extending each tuple by adding a component with the room in which the course meets. The resulting relation has scheme

{Course, StudentId, Grade, Name, Address, Phone, Day, Hour, Room}

and the meaning of a tuple $(c, s, g, n, a, p, d, h, r)$ is that

1. Student s took course c and got grade g .
2. The name of the student with ID number s is n , his or her address is a , and phone is p .
3. The course c meets in room r , and one meeting of the course is on day d at hour h .

To this set of tuples, we must apply the selection that restricts our consideration to the relevant tuples, namely, those in which the Name component is “C. Brown,” the Day component is “M,” and the Hour component is “9AM.” There will be at most one such tuple, on the assumption that C. Brown is taking at most one course meeting at 9AM on Mondays. Since the answer we want is the Room component of this tuple, we finish our expression by projecting onto Room. The expression tree for our query is shown in Fig. 8.18. It consists of the four-way join, followed by the selection, and then the projection.

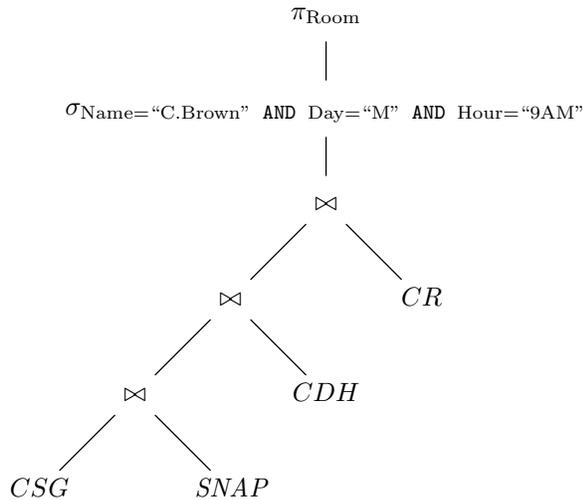


Fig. 8.18. Initial expression to determine where C. Brown is at 9AM, Mondays.

If we were to evaluate the expression of Fig. 8.18 as written, we would construct an enormous relation by joining CSG , $SNAP$, CDH , and CR , and then restrict it to a single tuple and project that tuple to a single component. Remember from Section 8.6 that it is not necessary to build such a big relation; we can “push the selection down the tree” to restrict the relations involved in the join, thus limiting greatly the sizes of the relations we do construct.

Our first step is shown in Fig. 8.19(a). Notice that the selection involves only attributes Name, Day, and Hour. None of these come from the right operand of the top join in Fig. 8.18; they all come from the left side, which is the join of CSG , $SNAP$, and CDH . Thus we may push the selection below the top join and have it apply to the left operand only, as we see in Fig. 8.19(a).

Now we cannot push the selection further, because one of the attributes involved, Name, comes from the left operand of the middle join in Fig. 8.19(a), while the other attributes, Day and Hour, come from the right operand, the relation CDH . Thus we must split the condition in the selection, which is the AND of three

conditions. We could split into three selections, but in this example it suffices to split the condition $\text{Name} = \text{“C. Brown”}$ off from the other two. The result of the split is shown in Fig. 8.19(b).

Now, the selection involving Day and Hour can be pushed down to the right operand of the middle join, since the right operand, the relation CDH , has both attributes Day and Hour . Then the other selection, involving Name , can be pushed to the left operand of the middle join, since that operand, $CSG \bowtie SNAP$, has Name as an attribute. These two changes yield the expression tree shown in Fig. 8.19(c).

Finally, the selection on Name involves an attribute of $SNAP$, and so we can push this selection to the right operand of the bottom join. This change is shown in Fig. 8.19(d).

Now we have an expression that gives us almost the same plan as we developed in Section 8.6 for this query. We begin at the bottom of the expression in Fig. 8.19(d) by finding the student ID(s) for the student(s) named “C. Brown.” By joining the tuples of $SNAP$ that have $\text{Name} = \text{“C. Brown”}$ with the CSG relation, we get the courses taken by C. Brown. When we apply the second selection to relation CDH , we get the courses that meet at 9AM on Mondays. The middle join in Fig. 8.19(d) thus gives us tuples with a course that both is taken by C. Brown and meets at 9AM Mondays. The top join gets the rooms in which those courses meet, and the projection gives us these rooms as answer.

The major difference between this plan and the plan of Section 8.6 is that the latter projects away useless components of tuples, while the plan here carries them along until the end. Thus to complete our optimization of expressions of relational algebra, we need laws that push projections down the tree. These laws are not all the same as the laws for selection, as we shall see in the next subsection. ♦

Laws Involving Projection

First, whereas selections can be pushed below a union, an intersection, or a set difference (provided that we push the selection to *both* operands), projections push below unions only. That is, the law

$$(\pi_L(R \cup S)) \equiv (\pi_L(R) \cup \pi_L(S))$$

holds. However, it is not true that $\pi_L(R \cap S)$ is necessarily the same as

$$\pi_L(R) \cap \pi_L(S)$$

For example, suppose that R and S are relations with scheme $\{A, B\}$, R contains only the tuple (a, b) , and S contains only the tuple (a, c) . Then $\pi_A(R) \cap \pi_A(S)$ contains the (one-component) tuple (a) , while $\pi_A(R \cap S)$ does not (because $R \cap S$ is empty). Thus we have a situation in which

$$(\pi_A(R \cap S)) \neq (\pi_A(R) \cap \pi_A(S))$$

It is possible to push a projection below a join. In general, we need a projection operator for each operand of the join. If we have an expression $\pi_L(R \bowtie_{A=B} S)$, then the attributes of R that we need are those appearing in the list of attributes L , and the attribute A , which is the attribute from R upon which the join is based. Similarly, from S we need those attributes that are on the list L , and we also need the join attribute B , regardless of whether it is or is not on L . Formally, the law for pushing projections below joins is

**Projection
pushing**

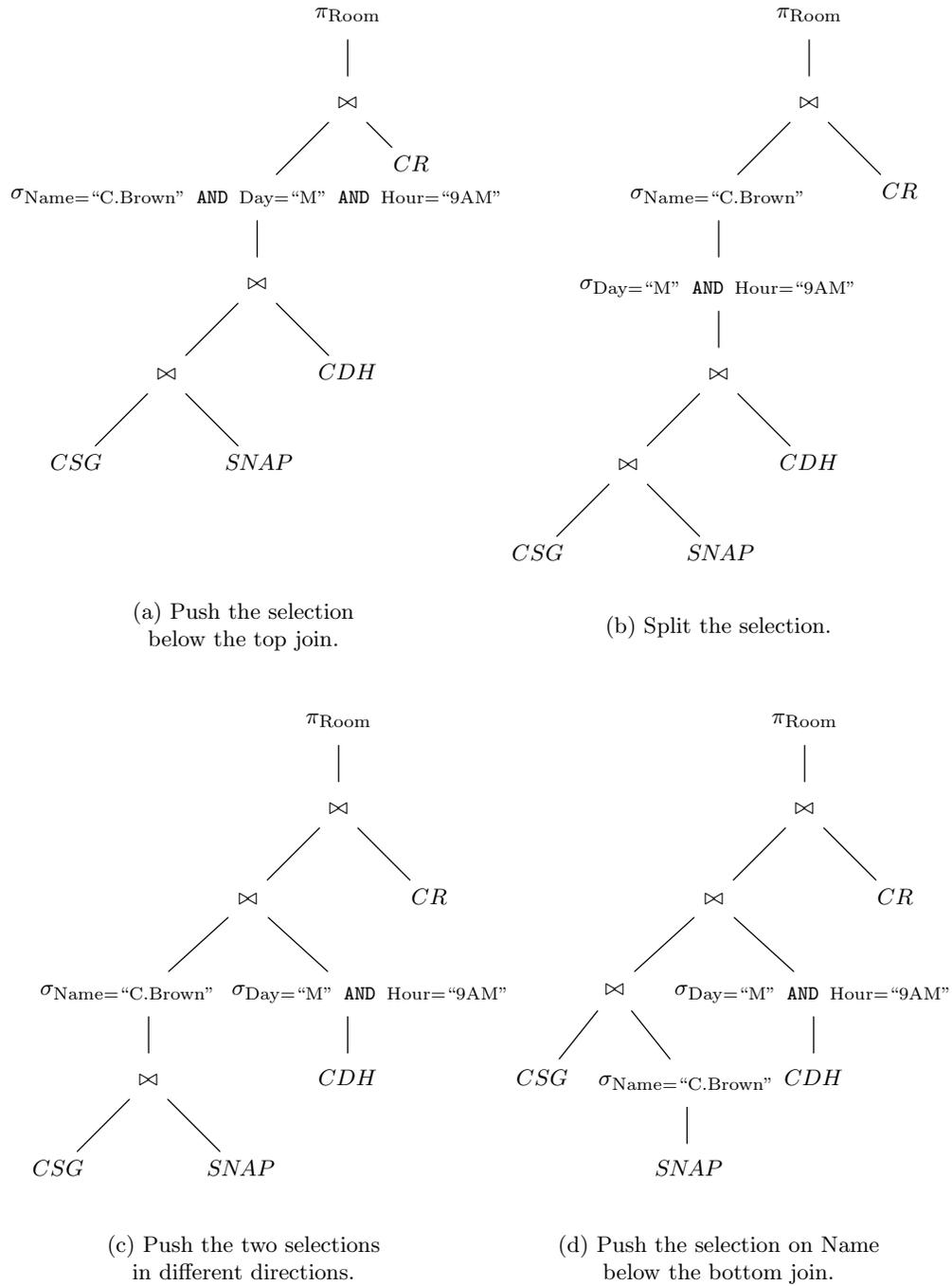


Fig. 8.19. Pushing the selection down.

$$(\pi_L(R \bowtie_{A=B} S)) \equiv (\pi_L(\pi_M(R) \bowtie_{A=B} \pi_N(S)))$$

where

1. List M consists of those attributes of L that are in the scheme for R , followed by attribute A if it is not on L , and
2. List N is the attributes of L that are in the scheme of S , followed by B if that attribute is not on list L .

Note that the useful way in which to apply this *projection pushing* law is from left to right, even though we thereby introduce two additional projections and do not get rid of any. The reason is that it is usually beneficial to project out what attributes we can as early as possible, that is, as far down the tree as we can. We still may have to do the projection onto the list L after the join, in the situation where the join attribute A is not on the list L (recall that the other join attribute, B from S , will not appear in the join anyway).

Sometimes, the lists M and/or N consist of all attributes of R or S , respectively. If so, there is no point in performing the projection, since it has no effect, except perhaps a pointless permutation of the columns of the relation. Thus we shall use the following law.

$$\pi_L(R) \equiv R$$

provided that list L consists of all the attributes in the scheme for R . Note that this law takes the point of view that relations are not changed by permutations of their columns.

There is also a situation in which we do not want to bother projecting. Suppose we have a subexpression $\pi_L(R)$ that is part of a larger expression, and let R be a single relation (rather than an expression involving one or more occurrences of operators). Suppose also that above this subexpression in the expression tree is another projection. To perform the projection on R now requires us to examine the entire relation, regardless of the existence of indexes. If we instead carry along the attributes of R not on the list L , until the next opportunity to project out those attributes, we are frequently able to save a significant amount of time.

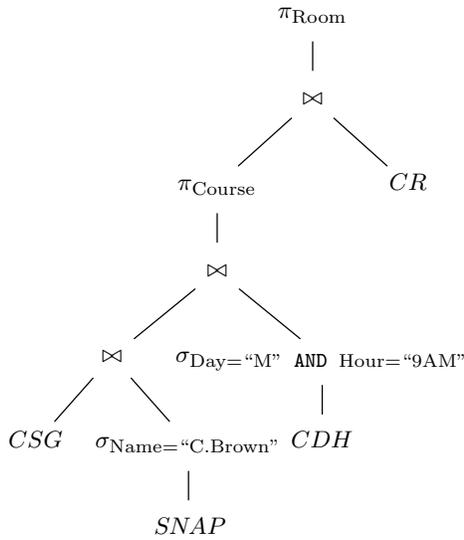
For instance, we shall, in the next example, discuss a subexpression

$$\pi_{\text{Course, StudentId}}(CSG)$$

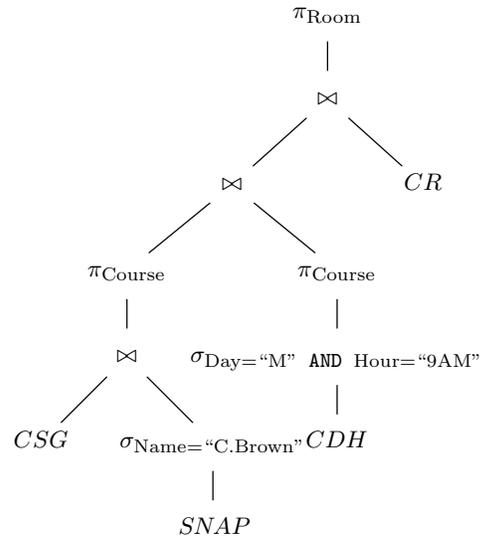
which has the effect of getting rid of grades. Since our entire expression, which is for the query of Example 8.17, eventually focuses on a few tuples of the CSG relation, we are much better off projecting out grades later; by so doing, we avoid ever examining the entire CSG relation.

◆ **Example 8.18.** Let us proceed from Fig. 8.19(d) to push projections down. The projection at the root is first pushed below the top join. The projection list consists of only Room, and the join attribute on both sides of the join is Course. Thus on the left we project onto Course alone, since Room is not an attribute of the expression on the left. The right operand of the join is projected onto both Course and Room. Since these are all the attributes of the operand CR , we can omit the projection. The resulting expression is shown in Fig. 8.20(a).

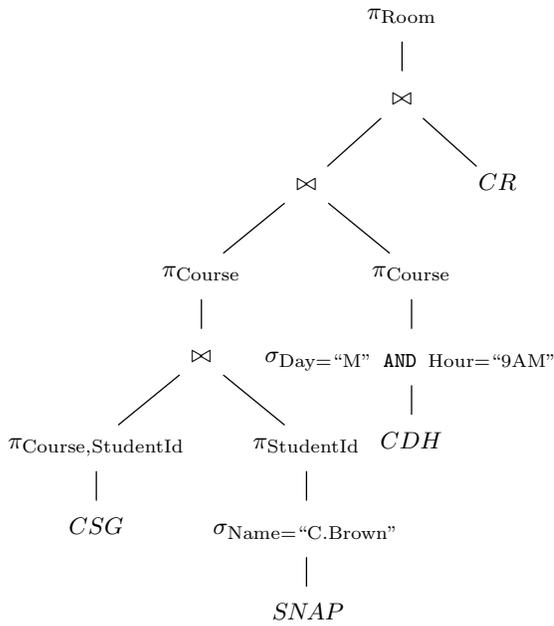
Now, we can push the projection onto Course below the middle join. Since



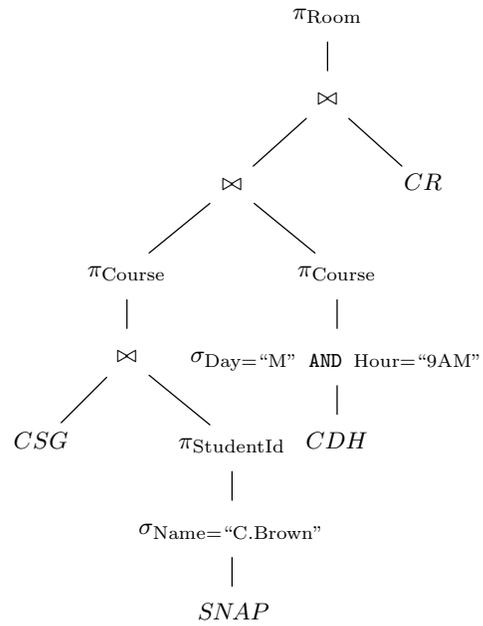
(a) Push the projection below the top join.



(b) Push the projection below the middle join.



(c) Push the projection below the bottom join.



(d) Remove the step that projects out the grade from CSG.

Fig. 8.20. Pushing the projection down.

Course is also the join attribute on both sides, we introduce two operators π_{Course} below the middle join. Since the result of the middle join then has only attribute Course, we no longer need the projection above that join; the new expression is shown in Fig. 8.20(b). Note that this join, involving two relations whose tuples have only the one component Course, is effectively an intersection of sets. That makes sense — it intersects the set of courses *C. Brown* is taking with the set of courses that meet at 9AM Mondays.

At this point, we need to push π_{Course} below the bottom join. The join attribute is StudentId on both sides, and so the projection list on the left is (Course, StudentId) and the list on the right is just StudentId (because Course is not an attribute of the expression on the right). The expression that results is shown in Fig. 8.20(c).

Finally, as we mentioned just before the example, it is advantageous here not to project Grade out of the *CSG* relation immediately. Above that projection we meet the operator π_{Course} , which will get rid of the grades anyway. If we instead use the expression of Fig. 8.20(d), we have essentially the plan of Section 8.6 for this query. That is, the expression $\pi_{\text{StudentId}}(\sigma_{\text{Name}=\text{“C.Brown”}}(\textit{SNAP}))$ gives us the student ID(s) for students named “C. Brown,” and the first join followed by projection π_{Course} gives us the courses taken by those students. If there is an index on Name for relation *SNAP* and there is an index on StudentId for relation *CSG*, then these operations are performed quickly.

The subexpression $\pi_{\text{Course}}(\sigma_{\text{Day}=\text{“M” AND Hour}=\text{“9AM”}}(\textit{CDH}))$ has as its value the courses that meet at 9AM Mondays, and the middle join intersects these sets to give us the courses taken by a student named “C. Brown” that meet at 9AM Mondays. Finally, the top join followed by projection looks up these courses in the *CR* relation (a fast operation if there is an index on Course), and produces the associated rooms as answer. ♦

EXERCISES

8.9.1*: Prove that

$$((R \bowtie_{A=B} S) \bowtie_{C=D} T) \equiv (R \bowtie_{A=B} (S \bowtie_{C=D} T))$$

whenever *A* is an attribute of *R*, *B* and *C* are different attributes of *S*, and *D* is an attribute of *T*. Why is it important that $B \neq C$? *Hint:* Remember that certain attributes disappear when a join is taken.

8.9.2*: Prove that

$$((R \bowtie_{A=B} S) \bowtie_{A=C} T) \equiv (R \bowtie_{A=B} (S \bowtie_{B=C} T))$$

whenever *A* is an attribute of *R*, *B* is an attribute of *S*, and *C* is an attribute of *T*.

8.9.3: Take each of your relational algebra queries from Exercise 8.7.3 and push selections and projections down as far as you can.

8.9.4: Let us make the following gross simplifications regarding the number of tuples that appear in relations that are the result of the operations of relational algebra.

- i)* Each operand relation has 1000 tuples.
- ii)* When we join relations with n and m tuples, respectively, the resulting relation has $mn/100$ tuples.
- iii)* When we perform a selection whose condition is the AND of k conditions, each of which equates an attribute to a constant value, we divide the size of the relation by 10^k .
- iv)* When we perform a projection, the size of the relation does not change.

Further, let us estimate the cost of evaluating an expression by the sum of the sizes of the relations computed for each interior node. Give the costs of each of the expressions in Figs. 8.18, 8.19(a) through (d), and 8.20(a) through (d).

8.9.5*: Prove the selection-pushing law

$$(\sigma_C(R \bowtie S)) \equiv ((\sigma_C(R)) \bowtie S)$$

Hint: To prove the equality of two sets, it is often easiest to show that each is a subset of the other, as discussed in Section 7.3.

8.9.6*: Prove the laws

- a) $(\sigma_C(R \cap S)) \equiv (\sigma_C(R) \cap \sigma_C(S))$
- b) $(\sigma_C(R \cup S)) \equiv (\sigma_C(R) \cup \sigma_C(S))$
- c) $(\sigma_C(R - S)) \equiv (\sigma_C(R) - \sigma_C(S))$

8.9.7*: Give an example to show that the law

$$(\pi_L(R - S)) \equiv (\pi_L(R) - \pi_L(S))$$

does not hold.

8.9.8:** It is sometimes possible to push a selection down both ways through a join, using the “equivalence”

$$\sigma_C(R \bowtie S) \equiv (\sigma_C(R) \bowtie \sigma_C(S)) \tag{8.1}$$

- a) Under what circumstances is Equation (8.1) truly an equivalence?
- b) If (8.1) is valid, when would it be better to use this law, rather than push the selection down only to R or only to S ?

◆◆ 8.10 Summary of Chapter 8

You should remember the following points from this chapter.

- ◆ Two-dimensional tables, called relations, are a versatile way to store information.
- ◆ Rows of a relation are called “tuples,” and the columns are named by “attributes.”
- ◆ A “primary index” represents the tuples of a relation as data structures and distributes them in such a way that operations using values in certain attributes — the “domain” for the index — are facilitated.

- ◆ A “key” for a relation is a set of attributes that uniquely determine values for the other attributes of the relation. Often, a primary index uses a key for its domain.
- ◆ “Secondary indexes” are data structures that facilitate operations that specify a particular attribute, usually one not part of the domain for the primary index.
- ◆ Relational algebra is a high-level notation for specifying queries about one or more relations. Its principal operations are union, intersection, difference, selection, projection, and join.
- ◆ There are a number of ways to implement joins more efficiently than the obvious “nested-loop join,” which pairs each tuple of one relation with each tuple of the other. Index-join and sort-join run in time that is close to what it takes to look at the two relations involved and produce the result of the join.
- ◆ Optimization of expressions in relational algebra can make significant improvements in the running time for evaluation of expressions and is therefore essential if languages based on relational algebra are to be used in practice to express queries.
- ◆ A number of ways to improve the running time of a given expression are known. Pushing down selections is often the most profitable.

◆◆◆ 8.11 Bibliographic Notes for Chapter 8

Further study of database systems, especially those based on the relational model, can be found in Ullman [1988].

The paper by Codd [1970] is generally regarded as the origin of the relational data model, although there were a number of earlier works that contained some of the ideas. The first implementations of systems using this model were INGRES (Stonebraker et al. [1976]) at Berkeley and System R (Astrahan et al. [1976]) at IBM. The latter is the origin of the language SQL sampled in Section 8.7 and found in many database management systems today; see Chamberlin et al. [1976]. The relational model is also found in the UNIX command `awk` (Aho, Kernighan, and Weinberger [1988]).

Aho, A. V., B. W. Kernighan, and P. J. Weinberger [1988]. *The AWK programming Language*, Addison-Wesley, Reading, Mass.

Astrahan, M. M., et al. [1976]. “System R: a relational approach to data management,” *ACM Trans. on Database Systems* **1**:2, pp. 97–137.

Chamberlin, D. D., et al. [1976]. “SEQUENCE 2: a unified approach to data definition, manipulation, and control,” *IBM J. Research and Development* **20**:6, pp. 560–575.

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

Stonebraker, M., E. Wong, P. Kreps, and G. Held [1976]. “The design and implementation of INGRES,” *ACM Trans. on Database Systems* **1**:3, pp. 189–222.

Ullman, J. D. [1988]. *Principles of Database and Knowledge-Base Systems* (two volumes) Computer Science Press, New York.