ABOUT THE BOOK

This book lives up to its title by describing the fundamental concepts behind database systems. *Database System Concepts* shows how to solve many of the problems encountered in designing and using a database system. Readers are introduced to the entity/relationship and relational models first, followed by the network and hierarchical models. Several chapters are devoted to the physical organization of databases, index techniques, and query processing, and the latter part of the book delves into advanced areas, including coverage of distributed databases, database security, and artificial intelligence.

ABOUT THE AUTHORS

Henry F. Korth is Assistant Professor of Computer Sciences at the University of Texas at Austin. Prior to joining the University of Texas faculty, Dr. Korth was a staff member of the IBM T.J. Watson Research Center in New York, where he was involved in the design and implementation of a distributed office automation system. His writings on database systems have appeared in several ACM and IEEE publications.

Abraham Silberschatz is Professor of Computer Sciences at the University of Texas at Austin, where he specializes in the area of concurrent processing. His research interests include operating systems, database systems, distributed systems, and programming languages. Dr. Silberschatz is the holder of the First David Bruton Jr. Centennial Professorship in Computer Sciences, and coauthor of the best-selling *Operating System Concepts* textbook.

BEST AVAILABLE COPY

McGraw-Hill Book Company
Serving the Need for Knowledge®
1221 Avenue of the Americas
New York, N.Y. 10020

ISBN 0-07-044752-7

KORTH
SILBERSCHATZ

DATABASE SYSTEM CONCEPTS

DATABASE SYSTEM CONCEPTS

HENRY F. KORTH
ABRAHAM SILBERSCHATZ

DATABASE SYSTEM CONCEPTS

HENRY F. KORTH
ABRAHAM SILBERSCHATZ
University of Texas at Austin

McGraw-Hill Book Company
New York St. Louis San Francisco Auckland Bogotá Hamburg
London Madrid Mexico Montreal New Delhi
Paris Panama São Paulo Singapore Sydney Tokyo Toronto

McGraw-Hill Advanced Computer Science Series

Davis and Lenat: Knowledge-Based Systems in Artificial Intelligence
Kogge: The Architecture of Pipelined Computers
Lindsay, Buchanan, Feigenbaum, and Lederberg: Applications of Artificial Intelligence for Organic Chemistry: The Dendral Project
Nilsson: Problem-Solving Methods in Artificial Intelligence
Wulf, Levin, and Harbison: HYDRA/C.mmp: An Experimental Computer System

DATABASE SYSTEM CONCEPTS

67890 DODO 89

ISBN 0-07-044752-7

The editor was Kaye Pace; the production supervisor was Joe Campanella; the cover was designed by Anne Canevari Green. Project supervision was done by Caliber Design Planning, Inc.

In memory of my father Joseph Silberschatz,
and my grandparents Stepha and Aaron Rosenblum.

Avi Silberschatz

To my parents

Hank Korth

2.7 Every weak entity set can be converted to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entities?

2.8 Suppose that you design an E-R diagram in which the same entity set appears several times. Why is this a bad practice that should be avoided whenever possible?

2.9 When designing an E-R diagram for a particular enterprise, there exists several alternative designs.

a. What criteria should you consider in deciding on the appropriate choice?

b. Come up with several alternative E-R diagrams to represent an enterprise. List the merits of each alternative and argue in favor of one of the alternatives.

2.10 Explain the difference between generalization and specialization.

Bibliographic Notes

The entity relationship data model was introduced by Chen [1976]. Discussions concerning the applicability of the E-R approach to database design are offered by Chen [1977], Sakai [1980], and Ng [1981]. Modeling techniques based on the E-R approach are covered by Schiffner and Scheuermann [1979], Lusk et al. [1980], Casanova [1984], and Wang [1984].

Various data manipulation languages for the E-R model have been proposed. These include CABLE [Shoshani 1978], GERM [Benneworth et al. 1981], and GORDAS [ElMasri and Wiederhold 1983]. A graphical query language for the E-R database was proposed by Zhang and Mendelzon [1983].

The concepts of generalization, specialization, and aggregation were introduced by Smith and Smith [1977]. Lenzerini and Santucci [1983] have used these concepts in defining cardinality constraints in the E-R model.

Basic textbook discussions are offered by Tsichritzis and Lochovsky [1982] and by Chen [1983].

3 Relational Model

From a historical perspective, the relational data model is relatively new. The first database systems were based on either the hierarchical model (see Chapter 5) or the network model (see Chapter 4). Those two older models are tied more closely to the underlying implementation of the database than is the relational model.

The relational data model represents the database as a collection of tables. Although tables are a simple, intuitive notion, there is a direct correspondence between the concept of a table and the mathematical concept of a relation.

In the years following the introduction of the relational model, a substantial theory has developed for relational databases. This theory assists in the design of relational databases and in the efficient processing of user requests for information from the database. We shall study this theory in Chapter 6, after we have introduced all the major data models.

3.1 Structure of Relational Databases

A relational database consists of a collection of *tables*, each of which is assigned a unique name. Each table has a structure similar to that presented in Chapter 2, where we represented E-R databases by tables. A row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In what follows, we introduce the concept of relation.

In this chapter, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a banking enterprise. They differ slightly from the tables that were used in Chapter 2 in order to simplify our presentation. We shall discuss appropriate relational structures in great detail in Chapter 6.

Consider the *deposit* table of Figure 3.1. It has four attributes: *branch-name, account-number, customer-name, balance*. For each attribute, there is a set of permitted values, called the *domain* of that attribute. For the

branch-name	*account-number*	*customer-name*	*balance*
Downtown	101	Johnson	500
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Round Hill	305	Turner	350
Perryridge	201	Williams	900
Redwood	222	Lindsay	700
Brighton	217	Green	750

Figure 3.1 The *deposit* relation.

attribute *branch-name*, for example, the domain would be the set of all branch names. Let D_1 denote this set and let D_2 denote the set of all account-numbers, D_3 the set of all customer names, and D_4 the set of all balances. As we saw in Chapter 2, any row of *deposit* must consist of a 4-tuple (v_1, v_2, v_3, v_4) where v_1 is a branch name (that is, v_1 is in domain D_1), v_2 is an account number (that is, v_2 is in domain D_2), v_3 is a customer name (that is, v_3 is in domain D_3), and v_4 is a balance (that is, v_4 is in domain D_4). In general, *deposit* will contain only a subset of the set of all possible rows. Therefore *deposit* is a subset of:

$$\mathop{\times}_{i=1}^{4} D_i$$

In general, a table of n columns must be a subset of

$$\mathop{\times}_{i=1}^{n} D_i$$

Mathematicians define a *relation* to be a subset of a cartesian product of a list of domains. This corresponds almost exactly with our definition of table. The only difference is that we have assigned names to attributes, while mathematicians rely on numeric "names," using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the attribute whose domain appears second, etc. Because tables are essentially relations, we shall use the mathematical terms *relation* and *tuple* in place of the terms *table* and *row*.

In the *deposit* relation of Figure 3.1, there are seven tuples. Let the *tuple variable* t refer to the first tuple of the relation. We use the notation t[*branch-name*] to denote the value of t on the *branch-name* attribute. Thus, t[*branch-name*] = "Downtown". Similarly, t[*account-number*] denotes the value of t on the *account-number* attribute, etc. Alternatively, we may write

$t[1]$ to denote the value of tuple t on the first attribute (*branch-name*), $t[2]$ for *account-number*, etc. Since a relation is a set of tuples, we use the mathematical notation of $t \in r$ to denote that tuple t is in relation r.

When we talk about a database, we must differentiate between the *database schema*, that is, the logical design of the database, and a *database instance*, which is the data in the database at a given instant in time.

The concept of a relation *scheme* corresponds to the programming language notion of type definition. A variable of a given type has a particular value at a given instant in time. Thus, a variable in programming languages corresponds to the concept of an *instance* of a relation.

It is convenient to give a name to a relation scheme, just as we give names to type definitions in programming languages. We adopt the convention of using lowercase names for relations and names beginning with an uppercase letter for relation schemes. Following this notation, we use *Deposit-scheme* to denote the relation scheme for relation *deposit*. Thus,

Deposit-scheme = (*branch-name, account-number, customer-name, balance*)

In general, a relation scheme is a list of attributes and their corresponding domains. We denote the fact that *deposit* is a relation on scheme *Deposit* by

deposit (*Deposit-scheme*)

We shall not, in general, be concerned about the precise definition of the domain of each attribute until we discuss file systems in Chapter 7. However, when we do wish to define our domains, we use the notation

(*branch-name* : string, *account-number* : integer,
customer-name : string, *balance* : integer)

to define the relation scheme for the relation *deposit*.

As another example, consider the *customer* relation of Figure 3.2. The scheme for that relation is

Customer-scheme = (*customer-name, street, customer-city*)

Note that the attribute *customer-name* appears in both relation schemes. This is not a coincidence. Rather, the use of common attributes in relation schemes is one way of relating tuples of distinct relations. For example, suppose we wish to find the cities where depositors of the Perryridge branch live. We would look first at the *deposit* relation to find all depositors of the Perryridge branch. Then, for each such customer, we look in the *customer* relation to find the city he or she lives in. Using the terminology

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

Figure 3.2 The *customer* relation.

of the entity-relationship model, we would say that the attribute *customer-name* represents the same entity set in both relations.

It would appear that, for our banking example, we could have just one relation scheme rather than several. That is, it may be easier for a user to think in terms of one relation scheme rather than several. Suppose we used only one relation for our example, with scheme

Account-info-scheme = (*branch-name, account-number, customer-name, balance, street, customer-city*)

Observe that if a customer has several accounts, we must list her or his address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and was avoided by our use of two relations. If a customer has one or more accounts, but has not provided an address, we cannot construct a tuple on *Account-info-scheme*, since the values for the *street* and *customer-city* are not known. To represent incomplete tuples, we must use *null values*. Thus, in the above example, the values for *street* and *customer-city* must be null. By using two relations, one on *Customer-scheme* and one on *Deposit-scheme*, we can represent customers whose address is unknown, without using null values. We simply use a tuple on *Deposit-scheme* to represent the information about the account, and create no tuple on *Customer-scheme* until the address information becomes available. In Chapter 6, we shall study criteria to help us decide when one set of relation schemes is better than another. For now, we shall assume the relation schemes are given.

For the purpose of this chapter, we assume that the relation schemes for our banking enterprise are as follows:

Branch-scheme = (*branch-name, assets, branch-city*)
Customer-scheme = (*customer-name, street, customer-city*)
Deposit-scheme = (*branch-name, account-number, customer-name, balance*)
Borrow-scheme = (*branch-name, loan-number, customer-name, amount*)

We have already seen an example of a *deposit* relation and a *customer* relation. Figure 3.3 shows a sample *borrow* (*Borrow-scheme*) relation.

The notion of a *superkey, candidate key*, and *primary key*, as discussed in Chapter 2, is applicable also to the relational model. For example, in *Branch-scheme*, {*branch-name*} and {*branch-name, branch-city*} are both superkeys. {*branch-name, branch-city*} is not a candidate key because {*branch-name*} $\subseteq$ {*branch-name, branch-city*} and {*branch-name*} itself is a superkey. {*branch-name*}, however, *is* a candidate key, which for our purpose will also serve as a primary key. The attribute *branch-city* is not a superkey since two branches in the same city may have different names (and different asset figures). The primary key for the *customer-scheme* is *customer-name*. We are not using the *social-security* number, as was done in Chapter 2, in order to have smaller relation schemes in our running example of a bank database. We expect that in a real world database the *social-security* attribute would serve as a primary key.

Let R be a relation scheme. If we say that a subset K of R is a *superkey* for R, we are restricting consideration to relations $r(R)$ in which no two distinct tuples have the same values on all attributes in K. That is, if t_1 and t_2 are in r and $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

branch-name	*loan-number*	*customer-name*	*amount*
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200

Figure 3.3 The *borrow* relation.

3.2 Formal Query Languages

A *query language* is a language in which a user requests information from the database. These languages are typically higher-level languages than standard programming languages. Query languages can be categorized as being either *procedural* or *nonprocedural*. In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a nonprocedural language, the user describes the information desired without giving a specific procedure for obtaining that information.

Most commercial relational database systems offer a query language that includes elements of both the procedural and the nonprocedural approaches. We shall study several commercial languages later in this chapter. First, we look at two "pure" languages: one procedural and one nonprocedural. These "pure" languages lack the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

3.2.1 The Relational Algebra

The relational algebra is a procedural query language. There are five fundamental operations in the relational algebra. These operations are: *select*, *project*, *cartesian-product*, *union*, and *set-difference*. All of these operations produce a new relation as their result.

In addition to the five fundamental operations, we shall introduce several other operations, namely, *set intersection*, *theta join*, *natural join*, and *division*. These operations will be defined in terms of the fundamental operations.

Fundamental operations

The select and project operations are called *unary* operations, since they operate on one relation. The other three relations operate on pairs of relations and are, therefore, called *binary* operations.

The *select* operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is given in parentheses

branch-name	*loan-number*	*customer-name*	*amount*
Perryridge	15	Hayes	1500
Perryridge	25	Glenn	2500

Figure 3.4 Result of $\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}}(borrow)$.

following the σ. Thus, to select those tuples of the *borrow* relation where the branch is "Perryridge," we write

$$\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}}(borrow)$$

If the borrow relation is as shown in Figure 3.3, then the relation that results from the above query is as shown in Figure 3.4. We may find all tuples in which the amount borrowed is more than $1200 by writing

$$\sigma_{amount\ >\ 1200}(borrow)$$

In general, we allow comparisons using $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ in the selection predicate. Furthermore, several predicates may be combined into a larger predicate using the connectives *and* ($\wedge$) and *or* ($\vee$). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write

$$\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}\ \wedge\ amount\ >\ 1200}(borrow)$$

The selection predicate may include comparisons between two attributes. To illustrate this, we consider the relation scheme

$$Client\text{-}scheme = (customer\text{-}name,\ employee\text{-}name)$$

indicating that the employee is the "personal banker" of the customer. The relation *client* (*Client-scheme*) is shown in Figure 3.5. We may find all those customers who have the same name as their personal banker by writing

$$\sigma_{customer\text{-}name\ =\ employee\text{-}name}(client)$$

If the *client* relation is as given in Figure 3.5, the answer is the relation shown in Figure 3.6.

In the above example, we obtained a relation (Figure 3.6) on (*customer-name, employee-name*) in which $t[customer\text{-}name] = t[employee\text{-}name]$ for all tuples t. It seems redundant to list the person's name twice. We would

customer-name	*employee-name*
Turner	Johnson
Hayes	Jones
Johnson	Johnson

Figure 3.5 The *client* relation.

prefer a one attribute relation on (*customer-name*) which lists all those who have the same name as their personal banker. The *project* operation allows us to produce this relation. The project operation is a unary operation that copies its argument relation, with certain columns left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows Π in parentheses.

Suppose we want a relation showing customers and the branches from which they borrow, but do not care about the amount of the loan, nor the loan number. We may write

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\,(borrow)$$

Let us revisit the query "Find those customers who have the same name as their personal banker." We write

$$\Pi_{customer\text{-}name}\,(\sigma_{customer\text{-}name\ =\ employee\text{-}name}\,(client))$$

Notice that instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

The operations we have discussed up to this point allow us to extract information from only one relation at at time. We have not yet been able to combine information from several relations. One operation that allows us to do that is the *cartesian product* operation, denoted by a cross ($\times$). This operation is a binary operation. We shall use infix notation for binary operations and, thus, write the cartesian product of relations r_1 and r_2 as $r_1 \times r_2$. We saw the definition of cartesian product earlier in this chapter (recall that a relation is defined to be a subset of a cartesian product of a set of domains). From that definition we should already have some intuition about the definition of the relational algebra operation $\times$. However, we face the problem of choosing the attribute names for the relation that results from a cartesian product.

Suppose we want to find all clients of bank employee Johnson, as well as the cities in which the clients live. We need the information in both the *client* relation and the *customer* relation in order to do so. Figure 3.7 shows the relation r = *client* $\times$ *customer*. The relation scheme for r is

customer-name	*employee-name*
Johnson	Johnson

Figure 3.6 Result of $\sigma_{customer\text{-}name\ =\ employee\text{-}name}\,(client)$.

client. *customer-name*	*client.* *employee-name*	*customer.* *customer-name*	*customer.* *street*	*customer.* *customer-city*
Turner	Johnson	Jones	Main	Harrison
Turner	Johnson	Smith	North	Rye
Turner	Johnson	Hayes	Main	Harrison
Turner	Johnson	Curry	North	Rye
Turner	Johnson	Lindsay	Park	Pittsfield
Turner	Johnson	Turner	Putnam	Stamford
Turner	Johnson	Williams	Nassau	Princeton
Turner	Johnson	Adams	Spring	Pittsfield
Turner	Johnson	Johnson	Alma	Palo Alto
Turner	Johnson	Glenn	Sand Hill	Woodside
Turner	Johnson	Brooks	Senator	Brooklyn
Turner	Johnson	Green	Walnut	Stamford
Hayes	Jones	Jones	Main	Harrison
Hayes	Jones	Smith	North	Rye
Hayes	Jones	Hayes	Main	Harrison
Hayes	Jones	Curry	North	Rye
Hayes	Jones	Lindsay	Park	Pittsfield
Hayes	Jones	Turner	Putnam	Stamford
Hayes	Jones	Williams	Nassau	Princeton
Hayes	Jones	Adams	Spring	Pittsfield
Hayes	Jones	Johnson	Alma	Palo Alto
Hayes	Jones	Glenn	Sand Hill	Woodside
Hayes	Jones	Brooks	Senator	Brooklyn
Hayes	Jones	Green	Walnut	Stamford
Johnson	Johnson	Jones	Main	Harrison
Johnson	Johnson	Smith	North	Rye
Johnson	Johnson	Hayes	Main	Harrison
Johnson	Johnson	Curry	North	Rye
Johnson	Johnson	Lindsay	Park	Pittsfield
Johnson	Johnson	Turner	Putnam	Stamford
Johnson	Johnson	Williams	Nassau	Princeton
Johnson	Johnson	Adams	Spring	Pittsfield
Johnson	Johnson	Johnson	Alma	Palo Alto
Johnson	Johnson	Glenn	Sand Hill	Woodside
Johnson	Johnson	Brooks	Senator	Brooklyn
Johnson	Johnson	Green	Walnut	Stamford

Figure 3.7 Result of *client* $\times$ *customer*.

(*client.customer-name, client.employee-name, customer.customer-name, customer.street, customer.customer-city*)

That is, we simply list all the attributes of both relations, and attach the name of the relation from which the attribute originally came. We need to attach the relation name to distinguish *client.customer-name* from *customer.customer-name*.

Now that we know the relation scheme for $r = client \times customer$, what tuples appear in r? As you may have suspected, we construct a tuple of r out of each possible pair of tuples: one from the *client* relation and one from the *customer* relation. Thus r is a large relation, as can be seen from Figure 3.7.

Assume we have n_1 tuples in *client* and n_2 tuples in *customer*. Then there are $n_1 n_2$ ways of choosing a pair of tuples: one tuple from each relation, so there are $n_1 n_2$ tuples in r. In particular, note that it may be the case for some tuples t in r that $t[client.customer\text{-}name] \neq t[customer.customer\text{-}name]$.

In general, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then $r_1 \times r_2$ is a relation whose scheme is the concatenation of R_1 and R_2. Relation R contains all tuples t for which there is a tuple t_1 in r_1, and t_2 in r_2 for which $t[R_1] = t_1[R_1]$ and $t[R_2] = t_2[R_2]$.

Returning to the query "Find all clients of Johnson and the city in which they live," we consider the relation $r = client \times customer$. If we write

$$\sigma_{client.employee\text{-}name\ =\ \text{"Johnson"}}\ (client \times customer)$$

then the result relation is as shown in Figure 3.8. We have a relation pertaining only to employee Johnson. However, the *customer.customer-name* column may contain customers of employees other than Johnson (if you don't see why, look at the definition of cartesian product again). Note that the *client.customer-name* column contains only customers of Johnson. Since the cartesian product operation associates *every* tuple of *customer* with every tuple of *client*, we know that some tuple in $client \times customer$ has the address of the employee's customer. This occurs in those cases where it happens that *client.customer-name* = *customer.customer-name*. So if we write

$$\sigma_{client.customer\text{-}name\ =\ customer.customer\text{-}name}\ (\sigma_{client.employee\text{-}name\ =\ \text{"Johnson"}}\ (client \times customer))$$

we get only those tuples of $client \times customer$ that:

- Pertain to Johnson.
- Have the street and city of the customer of Johnson.

client. customer-name	*client. employee-name*	*customer. customer-name*	*customer. street*	*customer. customer-city*
Turner	Johnson	Jones	Main	Harrison
Turner	Johnson	Smith	North	Rye
Turner	Johnson	Hayes	Main	Harrison
Turner	Johnson	Curry	North	Rye
Turner	Johnson	Lindsay	Park	Pittsfield
Turner	Johnson	Turner	Putnam	Stamford
Turner	Johnson	Williams	Nassau	Princeton
Turner	Johnson	Adams	Spring	Pittsfield
Turner	Johnson	Johnson	Alma	Palo Alto
Turner	Johnson	Glenn	Sand Hill	Woodside
Turner	Johnson	Brooks	Senator	Brooklyn
Turner	Johnson	Green	Walnut	Stamford
Johnson	Johnson	Jones	Main	Harrison
Johnson	Johnson	Smith	North	Rye
Johnson	Johnson	Hayes	Main	Harrison
Johnson	Johnson	Curry	North	Rye
Johnson	Johnson	Lindsay	Park	Pittsfield
Johnson	Johnson	Turner	Putnam	Stamford
Johnson	Johnson	Williams	Nassau	Princeton
Johnson	Johnson	Adams	Spring	Pittsfield
Johnson	Johnson	Johnson	Alma	Palo Alto
Johnson	Johnson	Glenn	Sand Hill	Woodside
Johnson	Johnson	Brooks	Senator	Brooklyn
Johnson	Johnson	Green	Walnut	Stamford

Figure 3.8 Result of $\sigma_{client.employee\text{-}name\ =\ \text{"Johnson"}}\ (client \times customer)$.

Finally, since we want only *customer-name* and *customer-city*, we do a projection

$$\Pi_{client.customer\text{-}name,\ customer.customer\text{-}city}\ (\sigma_{client.customer\text{-}name\ =\ customer.customer\text{-}name}\ (\sigma_{client.employee\text{-}name\ =\ \text{"Johnson"}}\ (client \times customer)))$$

The result of this expression is the correct answer to our query.

Let us now consider a query that might be posed by a bank's advertising department: "Find all customers of the Perryridge branch." That is, find everyone who has a loan, an account, or both. To answer this query, we need the information in the *borrow* relation (Figure 3.3) and the *deposit* relation (Figure 3.1). We know how to find all customers with a loan at the Perryridge branch:

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrow))$$

We know also how to find all customers with an account at the Perryridge branch:

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(deposit))$$

To answer the query, we need the *union* of these two sets, that is, all customers appearing in either or both of the two relations. This is accomplished by the binary operation union, denoted, as in set theory, by $\cup$. So the expression the advertising department needs in our example is

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrow))$$
$$\cup\ \Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(deposit))$$

The result relation for this query appears in Figure 3.9. Notice that there are three tuples in the result even though the Perryridge branch has two borrowers and two depositors. This is due to the fact that Hayes is both a borrower and a depositor of the Perryridge branch. Since relations are sets, duplicate values are eliminated.

Observe that, in our example, we took the union of two sets, both of which consisted of *customer-name* values. In general, we must ensure that unions are taken between *compatible* relations. For example, it would not make sense to take the union of the *borrow* relation and the *customer* relation. The former is a relation of four attributes and the latter of three. Furthermore, consider a union of a set of customer names and a set of cities. Such a union would not make sense in most situations. Therefore, for a union operation $r \cup s$ to be legal, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.

customer-name
Hayes
Glenn
Williams

Figure 3.9 Names of all customers of the Perryridge branch.

2. The domains of the ith attribute of r and the ith attribute of s must be the same.

The *set-difference* operator, denoted by $-$, allows us to find tuples that are in one relation, but not in another. The expression $r - s$ results in a relation containing those tuples in r but not in s.

We can find all customers of the Perryridge branch who have an account there but do not have a loan there by writing:

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(deposit))$$
$$-\ \Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrow))$$

The result relation for this query appears in Figure 3.10.

Formal definition of the relational algebra

The five operators we have just seen allow us to give a complete definition of an expression in the relational algebra. A basic expression in the relational algebra consists of either one of the following:

- A relation in the database.
- A constant relation.

A general expression in the relational algebra is constructed out of smaller subexpressions. Let E_1 and E_2 be relational algebra expressions. Then,

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, where P is a predicate on attributes on E_1
- $\Pi_S(E_1)$, where S is a list consisting of some of the attributes appearing in E_1

are all relational algebra expressions.

customer-name
Williams

Figure 3.10 Customers with only an account at the Perryridge branch.

Additional operators

We have now seen the five fundamental operations of the relational algebra: σ, Π, $\times$, $\cup$, $-$. These five operators are sufficient to express any relational algebra query. However, if we restrict ourselves to just the five fundamental operators, some common queries are lengthy to express. Therefore, we define additional operators. These new operators do not add any power to the algebra, but they do simplify common queries.

For each new operator we define, we give an equivalent expression using only the five fundamental operators.

The first additional relational algebra operation we shall define is *set intersection* ($\cap$). Suppose we wish to find all customers that have *both* a loan and an account at the Perryridge branch. Using set intersection, we could write:

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrow)) \\ \cap\ \Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(deposit))$$

The result relation for this query appears in Figure 3.11.

Note, however, that we do not include set intersection as a fundamental operation. We do not do so because we can rewrite any relational algebra expression using set intersection by replacing the intersection operation with a pair of set difference operations as follows:

$$r \cap s = r - (r - s)$$

Thus, set intersection does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than $r - (r - s)$.

The next operations we add to the algebra are used to simplify many queries that require a cartesian product. Typically, a query that involves a cartesian product includes a selection operation on the result of the cartesian product. Consider the query "Find all customers who have a loan at the Perryridge branch and the cities in which they live." We first form the cartesian product of the *borrow* and *customer* relations, then we select those tuples that pertain to Perryridge and pertain to only one *customer-name*. Thus we write

$$\Pi_{borrow.customer\text{-}name,\ customer.customer\text{-}city}(\sigma_P(borrow \times customer))$$

Where:

$$P = borrow.branch\text{-}name = \text{"Perryridge"} \\ \wedge\ borrow.customer\text{-}name = customer.customer\text{-}name$$

customer-name
Hayes

Figure 3.11 Customers with an account and a loan at the Perryridge branch.

The *theta join* is a binary operation that allows us to combine the selection and cartesian product into one operation. The theta join is denoted by $\bowtie_\Theta$, where $\bowtie$ is the "join" symbol and the subscript Θ (the Greek letter theta) is replaced by the selection predicate. The theta join operator forms the cartesian product of its two arguments and then performs a selection using the predicate Θ.

We rewrite our relational algebra expression for "Find all customers having a loan at the Perryridge branch and the cities in which they live," using the theta join as follows:

$$\Pi_{borrow.customer\text{-}name,\ customer.customer\text{-}city}(borrow \bowtie_\Theta customer)$$

In this example, Θ is the predicate:

$$borrow.branch\text{-}name = \text{"Perryridge"} \\ \wedge\ borrow.customer\text{-}name = customer.customer\text{-}name$$

In general, Θ can be an arbitrary predicate. Given two relations, r and s, and a predicate Θ,

$$r \bowtie_\Theta s = \sigma_\Theta(r \times s)$$

The *natural-join* operation is a further notational simplification of the relational algebra. Let us consider a simpler version of the above example, "Find all customers having a loan *at some branch* and their cities." If we write this query as a theta join we obtain

$$\Pi_{borrow.customer\text{-}name,\ customer.customer\text{-}city} \\ (borrow \bowtie_{borrow.customer\text{-}name = customer.customer\text{-}name} customer)$$

Observe that this particular theta join forces equality on those attributes that appear in both relation schemes. This sort of predicate occurs frequently in practice. Indeed, if we are printing out pairs of (*customer-name, customer-city*), we would normally want the city to be the city in which customer lives, and not some arbitrary city. The *natural-join* operation is designed precisely for this sort of query.

Although the definition of natural-join is a bit complicated, it is applied easily. We can use the natural join to write the query "Find all customers having a loan at some branch and their cities" as follows:

$$\Pi_{\textit{customer-name, customer-city}}\ (\textit{borrow} \bowtie \textit{customer})$$

Since the schemes for *borrow* and *customer* (that is, *Borrow-scheme* and *Customer-scheme*) have the attribute *customer-name* in common, the natural-join operation considers only pairs of tuples that have the same value on *customer-name*. It combines each such pair of tuples into a single tuple on the union of the two schemes (that is, *branch-name, loan-number, customer-name, amount, street, customer-city*). After performing the projection, we obtain the relation shown in Figure 3.12. The earlier example, "Find all customers having a loan at the Perryridge branch and their cities," can be written as

$$\Pi_{\textit{customer-name, customer-city}}\ (\sigma_{\textit{branch-name} = \text{"Perryridge"}}\ (\textit{borrow} \bowtie \textit{customer}))$$

We are now ready for a formal definition of the natural join. Consider two relation schemes R and S which are, of course, lists of attribute names. Let us consider the schemes to be *sets* rather than lists. This allows us to denote those attributes in both R and S by $R \cap S$, and to denote those attributes that appear in R, in S, or in both by $R \cup S$. Note that we are talking, here, about union and intersection on sets of attributes, not relations.

Consider two relations $r(R)$ and $s(S)$. The natural join of r and s, denoted by $r \bowtie s$ is a relation on scheme $R \cup S$. It is the projection onto

customer-name	*customer-city*
Jones	Harrison
Smith	Rye
Hayes	Harrison
Curry	Rye
Turner	Stamford
Williams	Princeton
Adams	Pittsfield
Johnson	Palo Alto
Glenn	Woodside
Brooks	Brooklyn

Figure 3.12 Result of $\Pi_{\textit{customer-name, customer-city}}\ (\textit{borrow} \bowtie \textit{customer})$.

$R \cup S$ of a theta join where the predicate requires $r.A = s.A$ for each attribute A in $R \cap S$. Formally,

$$r \bowtie s = \Pi_{R \cup S}\ (r \bowtie_{r.A_1 = s.A_1 \wedge \ldots \wedge r.A_n = s.A_n} s)$$

where $R \cap S = \{A_1, \ldots, A_n\}$.

Now that we have introduced the natural join, we adopt the following convention for attribute names in cartesian products of relations: We shall use the notation *relation-name.attribute-name* only when necessary to avoid ambiguity. When no ambiguity results, we shall drop the *relation-name* prefix.

Because the natural join is central to much of relational database theory and practice, we give several examples of its use:

- Find the assets and name of all branches which have depositors (that is, customers with an account) living in Port Chester.

$$\Pi_{\textit{branch-name, assets}}\ (\sigma_{\textit{customer-city} = \text{"Port Chester"}}\ (\textit{customer} \bowtie \textit{deposit} \bowtie \textit{branch}))$$

Notice that we wrote customer ⋈ deposit ⋈ branch without inserting parentheses to specify

$$(\textit{customer} \bowtie \textit{deposit}) \bowtie \textit{branch}$$
$$\text{or}\quad \textit{customer} \bowtie (\textit{deposit} \bowtie \textit{branch})$$

We did not specify which expression we intended because they are equivalent. That is, the natural join is associative.

- Find all customers who have *both* an account and a loan at the Perryridge branch.

$$\Pi_{\textit{customer-name}}\ (\sigma_{\textit{branch-name} = \text{"Perryridge"}}\ (\textit{borrow} \bowtie \textit{deposit}))$$

Note that we could have written an expression for this query using set intersection:

$$\Pi_{\textit{customer-name}}\ (\sigma_{\textit{branch-name} = \text{"Perryridge"}}\ (\textit{deposit}))$$
$$\cap\ \Pi_{\textit{customer-name}}\ (\sigma_{\textit{branch-name} = \text{"Perryridge"}}\ (\textit{borrow}))$$

This example illustrates a general fact about the relational algebra: It is possible to write several equivalent relational algebra expressions that are quite different from each other.

- Let $r(R)$ and $s(S)$ be relations without any attributes in common, that is, $R \cap S = \emptyset$. ($\emptyset$ denotes the empty set.) Then $r \bowtie s = r \times s$.

We now introduce one final relational algebra operation, called *division* ($\div$). The division operation is suited to queries that include the phrase "for all." Suppose we wish to find all customers who have an account at all branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression:

$$r_1 = \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$$

We can find all *customer-name, branch-name* pairs for which the customer has an account at the branch by writing

$$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}(deposit)$$

Now we need to find customers who appear in r_2 with *every* branch name in r_1. The operation that provides exactly those customers is the divide operation. The query can be answered by writing

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(deposit)$$
$$\div\ \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$$

Formally, let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$. The relation $r \div s$ is a relation on scheme $R - S$. A tuple t is in $r \div s$ if for every tuple t_s in s there is a tuple t_r in r satisfying both of the following:

$$t_r[S] = t_s[S]$$
$$t_r[R - S] = t[R - S]$$

It may be surprising to discover that the division operation can, in fact, be defined in terms of the five fundamental operations. Let $r(R)$ and $s(S)$ be given, with $S \subseteq R$.

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - r)$$

To see that this is true, observe that $\Pi_{R-S}(r)$ gives us all tuples t that satisfy the second condition of the definition of division. The expression on the right side of the set difference operator,

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - r)$$

serves to eliminate those tuples that fail to satisfy the first condition of the definition of division. Let us see how it does this. Consider $\Pi_{R-S}(r) \times s$. This is a relation on scheme R which pairs every tuple in $\Pi_{R-S}(r)$ with every tuple in s. Thus $(\Pi_{R-S}(r) \times s) - r$ gives us those

pairs of tuples from $\Pi_{R-S}(r)$ and s which do not appear in r. If a tuple t is in

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - r)$$

then there is some tuple t_s in s that does not combine with tuple t to form a tuple in r. Thus t holds a value for attributes $R - S$ which does not appear in $r \div s$. It is these values that we eliminate from $\Pi_{R-S}(r)$.

At times it is necessary to express the cartesian product of a relation with itself. In order to distinguish between the attributes of the resulting relation, we must *rename* one of the operands of the cartesian product. For this purpose, we define a rename operator which allows us to refer to a relation by more than one name. Define rename($R1$, $R2$) to be a function which returns the relation specified by $R1$, but under the name $R2$. See exercises 3.5e, 3.5g and 3.11d for examples of the use of renaming.

3.2.2 The Relational Calculus

The relational algebra is a procedural language because, when we write a relational algebra expression, we provide a sequence of operations that generates the answer to our query. The relational calculus, on the other hand, is a nonprocedural language. In the relational calculus, we give a formal description of the information desired without specifying how to obtain that information.

There are two forms of the relational calculus, one in which the variables represent tuples, and one in which the variables represent values of domains. These variants are called the *tuple relational calculus* and the *domain relational calculus*. The two forms are very similar. As a result, we shall emphasize the tuple relational calculus.

A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

that is, the set of all tuples t such that predicate P is true for t. Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational algebra expressions in the last section.

Find the *branch-name, loan-number, customer-name,* and *amount* for loans of over $1200:

$$\{t \mid t \in borrow \wedge t[amount] > 1200\}$$

Suppose we want only the *customer-name* attribute, rather than all attributes of the *borrow* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on scheme (*customer-name*).

We need those tuples on (*customer-name*) such that there is a tuple in *borrow* pertaining to that *customer-name* with the *amount* attribute > 1200. In order to express this, we need the construct "there exists" from the predicate calculus in mathematical logic. The notation

$$\exists\, t\, (Q(t))$$

means "there exists a tuple t such that predicate $Q(t)$ is true."

Using this notation, we may write the query "Find all customers who have a loan for an amount greater than \$1200" as:

$$\{t \mid \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] = s[customer\text{-}name] \wedge s[amount] > 1200)\}$$

In English, we read the above expression as "the set of all tuples t such that there exists a tuple s in relation *borrow* for which the values of t and s for the *customer-name* attribute are equal, and the value of s for the *amount* attribute is greater than \$1200."

Consider the query "Find all customers having a loan from the Perryridge branch and the cities in which they live." This query is slightly more complex than we have seen so far since it involves two relations, namely, *customer* and *borrow*. But as we shall see, all this requires is that we have two "there exists" clauses in our tuple relational calculus expression. We write the query as follows:

$$\begin{aligned}\{t \mid \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] &= s[customer\text{-}name] \\ \wedge\, s[branch\text{-}name] &= \text{"Perryridge"} \\ \wedge\, \exists\, u\, (u \in customer \wedge u[customer\text{-}name] &= s[customer\text{-}name] \\ \wedge\, t[customer\text{-}city] &= u[customer\text{-}city]))\}\end{aligned}$$

In English, this is "the set of all (*customer-name*, *customer-city*) tuples for which *customer-name* is a borrower at Perryridge branch and *customer-city* is the city of *customer-name*." Tuple variable s ensures that the customer is a borrower at the Perryridge branch. Tuple variable u is restricted to pertain to the same customer as s, and u ensures that the *customer-city* is the city of the customer.

To find all customers having a loan, an account, or both at the Perryridge branch, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by "or" ($\vee$).

$$\begin{aligned}\{t \mid \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] &= s[customer\text{-}name] \\ \wedge\, s[branch\text{-}name] &= \text{"Perryridge"}) \\ \vee\, \exists\, u\, (u \in deposit \wedge t[customer\text{-}name] &= u[customer\text{-}name] \\ \wedge\, u[branch\text{-}name] &= \text{"Perryridge"})\}\end{aligned}$$

The above expression gives us the set of all *customer-name* tuples such that at least one of the following holds:

- The *customer-name* appears in some tuple of the *borrow* relation as a borrower from the Perryridge branch.
- The *customer-name* appears in some tuple of the *deposit* relation as a depositor of the Perryridge branch.

If some customer has both a loan and an account at the Perryridge branch, that customer appears only once in the result because the mathematical definition of a set does not allow duplicate members of a set.

If we now want *only* those customers that have *both* an account and a loan at the Perryridge branch, all we need to do is change the "or" ($\vee$) to "and" ($\wedge$) in the above expression.

$$\begin{aligned}\{t \mid \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] &= s[customer\text{-}name] \\ \wedge\, s[branch\text{-}name] &= \text{"Perryridge"}) \\ \wedge\, \exists\, u\, (u \in deposit \wedge t[customer\text{-}name] &= u[customer\text{-}name] \\ \wedge\, u[branch\text{-}name] &= \text{"Perryridge"})\}\end{aligned}$$

Now consider the query, "Find all customers who have an account at the Perryridge branch but do not have a loan from the Perryridge branch." The tuple relational calculus expression for this query is similar to those we have just seen, except for the use of the "not" ($\neg$) symbol.

$$\begin{aligned}\{t \mid \exists\, u\, (u \in deposit \wedge t[customer\text{-}name] &= u[customer\text{-}name] \\ \wedge\, u[branch\text{-}name] &= \text{"Perryridge"}) \\ \wedge\, \neg\, \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] &= s[customer\text{-}name] \\ \wedge\, s[branch\text{-}name] &= \text{"Perryridge"})\}\end{aligned}$$

The above tuple relational calculus expression uses the $\exists\, u\, (\ldots)$ clause to require that the customer have an account at the Perryridge branch, and it uses the $\neg\, \exists\, s\, (\ldots)$ clause to eliminate those customers who appear in some tuple of the *borrow* relation as having a loan from the Perryridge branch.

Finally, let us consider the query we used in Section 3.2.1 to illustrate the division operation, "Find all customers who have an account at all branches located in Brooklyn." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted $\forall$. The notation

$$\forall\, t(Q(t))$$

means "Q is true for all tuples t." We write the expression for our query as follows:

$$\{t \mid \forall\, u\ (u \notin branch \vee u[branch\text{-}city] \neq \text{"Brooklyn"} \vee \exists\, s\ (s \in deposit \wedge t[customer\text{-}name] = s[customer\text{-}name] \wedge u[branch\text{-}name] = s[branch\text{-}name]))\}$$

In English, we interpret the above expression as "the set of all (*customer-name*) tuples t such that for *all* (*branch-name, branch-city*) tuples at least one of the following is true:

- u is not a tuple of the *branch* relation (and therefore, does not pertain to a branch in Brooklyn).
- The value of u on attribute *branch-city* is not Brooklyn.
- The customer has an account at the branch whose name appears in the *branch-name* attribute of u.

We are now ready to give a formal definition of the tuple relational calculus. A tuple relational calculus expression is of the form:

$$\{t \mid P(t)\}$$

where P is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a "$\exists$" or "$\forall$". Thus in:

$$t \in borrow \wedge \exists\, s\ (t[customer\text{-}name] = s[customer\text{-}name])$$

t is a free variable. Tuple variable s is said to be a *bound* variable.

A tuple relational calculus formula is built up out of *atoms*. An atom is of one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation.
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operation ($<, \leq, =, \neq, >, \geq$). We require that attributes x and y have domains whose members can be compared by Θ.
- $s[x]\ \Theta\ c$, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x.

Formulae are built up from atoms using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$ and $P_1 \wedge P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, then:

$$\exists\, s\ (P_1(s)) \text{ and } \forall\, s\ (P_1(s))$$

are also formulae.

As was the case for the relational algebra, it is possible to write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include two rules:

1. $P_1 \wedge P_2$ is equivalent to $\neg(\neg P_1 \vee \neg P_2)$.
2. $\forall\, t\ (P_1(t))$ is equivalent to $\neg \exists\, t\ (\neg P_1(t))$.

There is one final issue we must address in the tuple relational calculus. A tuple relational calculus may generate an infinite relation. Suppose we wrote the expression:

$$\{t \mid t \notin borrow\}$$

There are infinitely many tuples that are not in *borrow*. Most of these tuples contain values that do not even appear in the database! Clearly, we do not wish to allow such expressions. Another type of expression we wish to disallow is:

$$\{t \mid \exists\, s\ (s[x] \neq c \wedge t[y] = s[y])\}$$

where x and y are attributes and c is a constant. It is possible that the only tuples that satisfy $s[x] \neq c$ are tuples whose values do not appear in the database. Finding such a tuple requires a search among the potentially infinite number of tuples that do not appear in the database.

To assist us in defining a restriction of the tuple relational calculus, we introduce the concept of the *domain* of a tuple relational calculus formula. Let P be a formula. Intuitively, the domain of P, denoted *dom(P)*, is the set of all values referenced in P. These include values mentioned in P itself as well as values that appear in a tuple of a relation mentioned in P. Thus, the domain of P is the set of all values that appear explicitly in P or that appear in one or more of the relations whose names appear in P.

These considerations motivate the concept of *safe* tuple relational calculus expressions. We say an expression $\{t \mid P(t)\}$ is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from $dom(P)$.
2. For every "there exists" subformula of the form $\exists\, s\, (P_1(s))$, the subformula is true if and only if there is a tuple s with values from $dom(P_1)$ such that $P_1(s)$ is true.
3. For every "for all" subformula of the form $\forall\, s\, (P_1(s))$, the subformula is true if and only if $P_1(s)$ is true for all tuples s with values from $dom(P_1)$.

The purpose of the notion of safety is to ensure that only values from $dom(P)$ appear in the result and to ensure that we can test "for all" and "there exists" subformulae without having to test infinitely many possibilities.

Consider the second rule in the definition of safety. For $\exists\, s\, (P_1(s))$ to be true, we need to find only one s for which $P_1(s)$ is true. In general, there would be infinitely many tuples to test. However, if the expression is safe, we know that we may restrict our attention to tuples with values from $dom(P_1)$. This reduces the number of tuples we must consider to a finite number. The situation for subformulae of the form $\forall\, s\, (P_1(s))$ is similar. To assert that $\forall\, s\, (P_1(s))$ is true, we must, in general, test all possible tuples. This requires us to examine infinitely many tuples. As above, if we know the expression is safe, it is sufficient for us to test $P_1(s)$ for those tuples s whose values are taken from $dom(P_1)$.

All the tuple relational calculus expressions we have written in the examples of this section are safe.

The tuple relational calculus, restricted to safe expressions, is equivalent in expressive power to the relational algebra. This means that for every relational algebra expression, there is an equivalent safe expression in the tuple relational calculus, and for every safe tuple relational calculus expression there is an equivalent relational algebra expression. We will not prove this fact here, but the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises.

There is a second form of the relational calculus called the *domain relational calculus*. In this form of the relational calculus, we use *domain* variables that take on values from an attribute's domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

An expression in the domain relational calculus is of the form $\{<x_1, x_2, \ldots, x_n> \mid P(x_1, x_2, \ldots, x_n)\}$ where the x_i, $1 \le i \le n$, represent domain variables. P represents a formula. As was the case for the tuple relational calculus, a formula is composed of atoms. An atom in the domain relational calculus is of the following forms:

- $<x_i, \ldots, x_n> \in r$, where r is a relation on n attributes and x_i, $1 \le i \le n$ are domain variables or domain constants.
- $x\ \Theta\ y$, where x and y are domain variables and Θ is a comparison operator ($<$, $\le$, $=$, $\ne$, $>$, $\ge$). We require that attributes x and y have domains that can be compared by Θ.
- $x\ \Theta\ c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

Formulae are built up from atoms using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$ and $P_1 \wedge P_2$.
- If $P_1(x)$ is a formula in x, where x is a domain variable, then

$$\exists\, x\, (P_1(x)) \text{ and } \forall\, x\, (P_1(x))$$

are also formulae.

As a notational shorthand, we write

$$\exists\, a,b,c\, (P(a,b,c))$$

for:

$$\exists\, a\, (\exists\, b\, (\exists\, c\, (P(a,b,c))))$$

The notion of safety applies to the domain calculus as well. The domain relational calculus, restricted to safe expressions, is equivalent to the tuple relational calculus, restricted to safe expressions. Since we noted earlier that the tuple relational calculus, restricted to safe expressions, is equivalent to the relational algebra, all three of the following are equivalent:

- The relational algebra.
- The tuple relational calculus restricted to safe expressions.
- The domain relational calculus restricted to safe expressions.

We now give domain relational calculus queries we considered earlier. Note the similarity of these expressions with the corresponding tuple relational calculus expressions:

Find the branch name, loan number, customer name, and amount for loans of over $1200:

$$\{<b,l,c,a> \mid <b,l,c,a> \in borrow \wedge a > 1200\}$$

Find all customers who have a loan for an amount greater than $1200:

$$\{<c> \mid \exists\, b,l,a\ (<b,l,c,a> \in borrow \wedge a > 1200)\}$$

Find all customers having a loan from the Perryridge branch and the city in which they live:

$$\{<c,x> \mid \exists\, b,l,a\ (<b,l,c,a> \in borrow \wedge b = \text{"Perryridge"} \wedge \exists\, y\ (<c,y,x> \in customer))\}$$

Find all customers having a loan, an account, or both at the Perryridge branch:

$$\{<c> \mid \exists\, b,l,a\ (<b,l,c,a> \in borrow \wedge b = \text{"Perryridge"}) \vee \exists\, b,a,n\ (<b,a,c,n> \in deposit \wedge b = \text{"Perryridge"})\}$$

Find all customers who have an account at all branches located in Brooklyn:

$$\{<c> \mid \forall\, x,y,z\ (<x,y,z> \notin branch \vee z \neq \text{"Brooklyn"} \vee (\exists\, a,n\ (<x,a,c,n> \in deposit)))\}$$

3.3 Commercial Query Languages

The formal languages we have just seen provide a concise language for representing queries. However, database system products require a more "user-friendly" query language. In this section, we study three of these product languages: SQL, Quel, and QBE. We have chosen these languages because they represent a variety of styles. QBE is based on the domain relational calculus; Quel is based on the tuple relational calculus; and SQL uses a combination of relational algebra and relational calculus constructs. All three of these languages have been influential not only in research database systems but also in commercially marketed systems.

Although we refer to these languages as "query languages," this is actually incorrect. SQL, Quel, and QBE contain many other capabilities besides querying a database. These include features for defining the structure of the data, features for modifying data in the database, and features for specifying security constraints. We shall defer discussion of these features to subsequent chapters and sections.

Our goal is not to provide a complete users' guide for these languages. Rather, we present the fundamental constructs and concepts of these languages. Individual implementations of these languages may differ in details, or support only a subset of the full language. We discuss some of these details in Chapter 15.

3.3.1 SQL

SQL was introduced as the query language for System R. SQL is an acronym for Structured Query Language. It is still referred to frequently by its former name, Sequel.

The basic structure of an SQL expression consists of three clauses: **select**, **from**, and **where**.

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- The **from** clause is a list of relations to be scanned in the execution of the expression.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

The different meaning of the term "select" in SQL and the relational algebra is an unfortunate historical fact. We emphasize the different interpretations here to minimize potential confusion.

A typical SQL query has the form:

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

The A_is represent attributes, the r_is represent relations, and P is a predicate. This query is equivalent to the relational algebra expression:

FST-IBM000857

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

If the **where** clause is omitted, the predicate P is **true**. The list $A_1, A_2, \ldots, A_n$ of attributes may be replaced with a star (*) to select all attributes of all relations appearing in the **from** clause.

SQL forms the cartesian product of the relations named in the **from** clause, performs a relational algebra selection using the **where** clause predicate, and projects the result onto the attributes of the **select** clause. In practice, SQL may convert the expression into an equivalent form that can be processed more efficiently. However, we shall defer concerns about efficiency to Chapter 9.

The result of an SQL query is, of course, a relation. Let us consider a very simple query using our banking example, "Find the names of all branches in the *deposit* relation":

select *branch-name*
from *deposit*

In the formal query languages, the mathematical notion of a relation being a set was used. Thus, duplicate tuples did not ever appear in relations. In practice, duplicate elimination is relatively time-consuming. Therefore, SQL (and most other commercial query languages) allow duplicates in relations. The above query will, thus, list each *branch-name* once for every tuple in which it appears in the *deposit* relation.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the above query as

select distinct *branch-name*
from *deposit*

if we want duplicates removed. We note for historical accuracy that early implementations of SQL used the keyword **unique** in place of **distinct**.

SQL includes the operations **union**, **intersect**, and **minus**, which operate on relations, and correspond directly to the relational algebra operations $\cup$, $\cap$, and $-$.

Let us see how the example queries that we considered earlier are written in SQL. First, we find all customers having an account at the Perryridge branch:

select *customer-name*
from *deposit*
where *branch-name* = "Perryridge"

Next, let us find all customers having a loan from the Perryridge branch:

select *customer-name*
from *borrow*
where *branch-name* = "Perryridge"

To find all customers having a loan, an account, or both at the Perryridge branch we write

(**select** *customer-name*
from *deposit*
where *branch-name* = "Perryridge")
union
(**select** *customer-name*
from *borrow*
where *branch-name* = "Perryridge")

Similarly, to find all customers who have both a loan and an account at the Perryridge branch, we write

(**select** *customer-name*
from *deposit*
where *branch-name* = "Perryridge")
intersect
(**select** *customer-name*
from *borrow*
where *branch-name* = "Perryridge")

To find all customers of the Perryridge branch who have an account there but no loan there, we write

(**select** *customer-name*
from *deposit*
where *branch-name* = "Perryridge")
minus
(**select** *customer-name*
from *borrow*
where *branch-name* = "Perryridge")

SQL does not have a direct representation of the natural-join operation. However, since the natural join is defined in terms of a cartesian product, a selection, and a projection, it is a relatively simple matter to write an SQL expression for the natural join.

FST-IBM000858

Recall that we wrote the relational algebra expression:

$$\Pi_{customer\text{-}name,\ customer\text{-}city}\ (borrow \bowtie customer)$$

for the query "Find all customers having a loan at some branch and their city." In SQL, we write

select *customer.customer-name, customer-city*
from *borrow, customer*
where *borrow.customer-name = customer.customer-name*

Notice that SQL uses the notation *relation-name.attribute-name*, as did the relational algebra, to avoid ambiguity in cases where an attribute appears in the scheme of more than one relation. We could have written *customer.customer-city* instead of *customer-city* in the **select** clause. However, since the attribute *customer-city* appears in only one of the relations named in the **from** clause, there is no ambiguity when we write *customer-city*.

Let us consider a somewhat more complicated query in which we require that the customers have a loan from the Perryridge branch: "Find the names of all customers having a loan at the Perryridge branch and their respective city." In order to state this query, we shall need to state two constraints in the **where** clause, connected by "and."

select *customer.customer-name, customer-city*
from *borrow, customer*
where *borrow.customer-name = customer.customer-name* **and**
    branch-name = "Perryridge"

SQL uses the logical connectives "**and**," "**or**," and "**not**" rather than the mathematical symbols "$\wedge$," "$\vee$" and "$\neg$."

SQL draws on the relational calculus for operations that allow testing tuples for membership in a relation. To illustrate this, reconsider the query "Find all customers who have both a loan and an account at the Perryridge branch." Earlier, we took the approach of intersecting two sets: the set of account holders at the Perryridge branch and the set of borrowers from the Perryridge branch. We can take the alternative approach of finding all account holders at the Perryridge branch who are members of the set of borrowers from the Perryridge branch. Clearly, this is an equivalent approach, but it leads us to write our query using the **in** connective of SQL.

The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

Let us use **in** to write the query "Find all customers who have both a loan and account at the Perryridge branch." We begin by finding all account holders, and write the subquery:

(**select** *customer-name*
from *deposit*
where *branch-name* = "Perryridge")

We then need to find those customers who are borrowers from the Perryridge branch and who appear in the list of account holders obtained in the above subquery. We do this by embedding the above subquery in an outer **select**. The resulting query is

select *customer-name*
from *borrow*
where *branch-name* = "Perryridge" **and**
    customer-name **in** (**select** *customer-name*
        from *deposit*
        where *branch-name* = "Perryridge")

These last two examples show that it is possible to write the same query several ways in SQL. This is beneficial since it allows a user to think about the query in the way that appears most natural. We shall see that there is a substantial amount of redundancy in SQL.

In the above example, we tested membership in a one-attribute relation. It is possible to test for membership in an arbitrary relation. SQL uses the notation $<v_1, v_2, ..., v_n>$ to denote a tuple of arity n containing values $v_1, v_2 ..., v_n$. Using this notation, we can write the query "Find all customers who have both an account and a loan at the Perryridge branch" in a third way:

select *customer-name*
from *borrow*
where *branch-name* = "Perryridge" **and**
    <*branch-name, customer-name*> **in**
    (**select** *branch-name, customer-name*
    from *deposit*)

We now illustrate the use of the "**not in**" construct. To find all customers who have an account at the Perryridge branch but do not have a loan at the Perryridge branch, we can write

```
select customer-name
from deposit
where branch-name = "Perryridge" and
      customer-name not in (select customer-name
                            from borrow
                            where branch-name = "Perryridge")
```

SQL borrows the notion of tuple variables from the tuple relational calculus. A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **from** clause. We illustrate the use of tuple variables by rewriting the query "Find all customers having a loan at some bank and their city":

```
select T.customer-name, customer-city
from borrow S, customer T
where S.customer-name = T.customer-name
```

Note that a tuple variable is defined in the **from** clause by placing it after the name of the relation it is associated with.

Tuple variables are most useful when we need to compare two tuples in the same relation. Suppose we want to find all customers who have an account at some bank at which Jones has an account. We write this query as follows:

```
select T.customer-name
from deposit S, deposit T
where S.customer-name = "Jones" and
      S.branch-name = T.branch-name
```

Observe that we could not use the notation *deposit.branch-name* since it would not be clear which reference to *deposit* is intended.

We note that an alternative way to express this query is

```
select customer-name
from deposit
where branch-name in
      (select branch-name
       from deposit
       where customer-name = "Jones")
```

We were able to use the **in** construct in the above query because we were testing for equality between two branch names. Consider the query "Find all branches that have greater assets than some branch located in Brooklyn." We can write the SQL expression:


```
select T.branch-name
from branch T, branch S
where T.assets > S.assets and
      S.branch-city = "Brooklyn"
```

Since the comparison is a "greater than" comparison, we cannot write this expression using the **in** construct.

SQL does, however, offer an alternative style for writing the above query. The phrase "greater than some" is represented in SQL by > **any**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select branch-name
from branch
where assets > any
     (select assets
      from branch
      where branch-city = "Brooklyn")
```

The subquery

```
(select assets
 from branch
 where branch-city = "Brooklyn")
```

generates the set of all asset values for branches in Brooklyn. The "> **any**" comparison in the where clause of the outer select is true if the *assets* value of the tuple is greater than at least one member of the set of all asset values for branches in Brooklyn.

SQL also allows "< **any**," "≤ **any**," "≥ **any**," "= **any**," and "≠ **any**" comparisons. As an exercise, verify that "= **any**" is identical to "**in**."

Now let us modify our query slightly. Let us find all branches that have greater assets than all branches in Brooklyn. We write this query using the "> **all**" construct:

```
select branch-name
from branch
where assets > all
     (select assets
      from branch
      where branch-city = "Brooklyn")
```

The constructs **in**, > **any**, > **all**, etc. allow us to test a single value against members of an entire set. Since a **select** generates a set of tuples, we may, at times, want to compare sets to determine if one set contains all the members of some other set. Such comparisons are made in SQL using the **contains** and **not contains** constructs.

Consider the query "Find all customers who have an account at all branches located in Brooklyn." For each customer, we need to see if the set of all branches at which that customer has an account contains the set of all branches in Brooklyn.

select *customer-name*
from *deposit S*
where (**select** *branch-name*
  from *deposit T*
  where *S.customer-name* = *T.customer-name*)
 contains
 (**select** *branch-name*
  from *branch*
  where *branch-city* = "Brooklyn")

The subquery

(**select** *branch-name*
 from *branch*
 where *branch-city* = "Brooklyn")

finds all the branches in Brooklyn. The subquery

(**select** *branch-name*
 from *deposit T*
 where *S.customer-name* = *T.customer-name*)

finds all the branches at which customer *S.customer-name* has an account. Thus, the outer **select** takes each customer and tests whether the set of all branches at which that customer has an account contains the set of all branches in Brooklyn.

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all customers having a loan at the Perryridge branch, we can write

select *customer-name*
from *borrow*
where *branch-name* = "Perryridge"
order by *customer-name*

In order to fulfill an **order by** request, SQL must perform a sort. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.

SQL offers the ability to compute functions of groups of tuples using the **group by** clause. The attribute given in the **group by** clause is used to form groups. Tuples with the same value on this attribute are placed in one group. SQL includes functions to compute:

- average: **avg**
- minimum: **min**
- maximum: **max**
- total: **sum**
- count: **count**

To find the average account balance at all branches, we write

select *branch-name*, **avg** (*balance*)
from *deposit*
group by *branch-name*

Operations like **avg** are called *aggregate operations* because they operate on aggregates of tuples.

At times it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested only in branches where the average account balance is more than $1,200. To express such a query, we use the **having** clause of SQL. Predicates in the **having** clause are applied after the formation of groups, so aggregate operators may be used in the **having** clause. We express this query in SQL by

select *branch-name*, **avg** (*balance*)
from *deposit*
group by *branch-name*
having avg (*balance*) > 1200.

The aggregate operator **count** is used frequently to count the number of tuples in a relation. The notation used for this in SQL is **count** (*). Thus, to find the number of tuples in the *customer* relation, we write

select count (*)
from *customer*.

Counting can be used to check for negative information. Suppose we wish to find all customers who have a deposit at the Perryridge branch,

FST-IBM000861

but for whom no address is on file. Our approach is to count the number of *customer* tuples pertaining to each depositor of the Perryridge branch. If the count is 0 for a depositor, we know we have no address for that person.

```
select customer-name
from deposit
where branch-name = "Perryridge"
    and 0 =
        select count (*)
        from customer
        where deposit.customer-name = customer.customer-name
```

As an alternative, SQL includes a special construct for the application of count used in the above example. The predicate **exists** takes a select statement as its argument and returns true unless the select results in an empty relation. We rewrite the example as follows:

```
select customer-name
from deposit
where branch-name = "Perryridge"
    and not exists
        select *
        from customer
        where deposit.customer-name = customer.customer-name
```

SQL is as powerful in expressiveness as the relational algebra (which, as we said earlier, is equivalent in power to the relational calculus). SQL includes the five basic relational algebra operators. Cartesian product is represented by the **from** clause of SQL. Projection is performed in the **select** clause. Algebra selection predicates are represented in SQL's **where** clause. Set union and difference appear in both the relational algebra and SQL. SQL allows intermediate results to be stored in temporary relations; thus we may encode any relational algebra expression in SQL.

SQL offers features that do not appear in the relational algebra. Most notable among these features are the aggregate operators. Thus, SQL is strictly more powerful than the algebra.

In this section, we have seen that SQL offers a rich collection of features, including capabilities not included in the formal query languages: aggregate operations, ordering of tuples, etc. Many SQL implementations allow SQL queries to be submitted from a program written in a general purpose language such as Pascal, PL/1, Fortran, C, or Cobol. This extends the programmer's ability to manipulate the database even further. We show how PL/1 and SQL are combined in an actual system in Chapter 15.

3.3.2 Quel

Quel was introduced as the query language for the Ingres database system. The basic structure of the language closely parallels that of the tuple relational calculus. Most Quel queries are expressed using three types of clauses: **range of**, **retrieve**, and **where**.

- Each tuple variable is declared in a **range of** clause. We say **range of** t **is** r to declare t to be a tuple variable restricted to take on values of tuples in r.
- The **retrieve** clause is similar in function to the **select** clause of SQL.
- The **where** clause contains the selection predicate.

A typical Quel query is of the form:

range of t_1 **is** r_1
range of t_2 **is** r_2
range of t_m **is** r_m
retrieve $(t_{i_1}.A_{j_1}, t_{i_2}.A_{j_2}, \ldots, t_{i_n}.A_{j_n})$
where P

The t_i, are the tuple variables. The r_i, are relations, and the A_{j_k}, are attributes. Quel uses the notation

$$t.A$$

to denote the value of tuple variable t on attribute A. This means the same as $t[A]$ in the tuple relational calculus.

Quel does not include relational algebra operations like **intersect**, **union**, or **minus**. Furthermore, Quel does not allow nested subqueries (unlike SQL). That is, we *cannot* have a nested **retrieve-where** clause inside a **where** clause.

Let us return to our bank example, and write some of our earlier queries using Quel. First, we find all customers having an account at the Perryridge branch:

```
range of t is deposit
retrieve (t.customer-name)
where t.branch-name = "Perryridge"
```

To show a Quel query involving more than one relation, let us consider the query "Find all customers having a loan at the Perryridge branch and their city."

range of *t* **is** *borrow*
range of *s* **is** *customer*
retrieve (*t.customer-name, s.customer-city*)
where *t.branch-name* = "Perryridge" **and**
t.customer-name = *s.customer-name*

Note that Quel, like SQL, uses the logical connectives **and**, **or**, and **not**, rather than the mathematical symbols "$\wedge$," "$\vee$," and "$\neg$" as used in the tuple relational calculus.

As another example of a query involving two relations, consider the query "Find all customers who have both a loan and an account at the Perryridge branch."

range of *s* **is** *borrow*
range of *t* **is** *deposit*
retrieve (*s.customer-name*)
where *t.branch-name* = "Perryridge" **and** *s.branch-name* = "Perryridge"
and *t.customer-name* = *s.customer-name*

In SQL, we had the option of writing a query such as the above one using the relational algebra operation **intersect**. As we noted above, Quel does not include this operation.

Let us consider a query for which we used the union operation in SQL: "Find all customers who have an account, a loan, or both at the Perryridge branch." Since we do not have a **union** operation in Quel, and we know that Quel is based on the tuple relational calculus, we might be guided by our tuple relational calculus expression for this query:

$$\{t \mid \exists\, s\, (s \in borrow \wedge t[customer\text{-}name] = s[customer\text{-}name] \wedge s[branch\text{-}name] = \text{"Perryridge"}) \vee \exists\, u\, (u \in deposit \wedge t[customer\text{-}name] = u[customer\text{-}name] \wedge u[branch\text{-}name] = \text{"Perryridge"})\}$$

Unfortunately, the above expression does not lead us to a Quel query. The problem is that in the tuple relational calculus query, we obtain customers from *both* tuple variable *s* (whose range is *borrow*) and tuple variable *u* (whose range is *deposit*). In Quel, our **retrieve** clause must be either

retrieve *s.customer-name*
or **retrieve** *u.customer-name*

If we choose the former, we exclude those depositors who are not borrowers. If we choose the latter, we exclude those borrowers who are not depositors.

In order to write this query in Quel, we must create a new relation and insert tuples into this new relation. Let us call this new relation *temp*. We obtain all depositors of the Perryridge branch by writing

range of *u* **is** *deposit*
retrieve into *temp* (*u.customer-name*)
where *u.branch-name* = "Perryridge"

The **into** *temp* clause causes a new relation, *temp*, to be created to hold the result of this query. Now we can find all borrowers of Perryridge branch and insert them in the newly created relation *temp*. We do this using the **append** command.

range of *s* **is** *borrow*
append to *temp* (*s.customer-name*)
where *s.branch-name* = "Perryridge"

The **append** command operates similarly to the **retrieve** command except that the tuples retrieved are added to the relation appearing after the keyword **to**.

We now have a relation *temp* containing all customers who have an account, a loan, or both, at the Perryridge branch. This relation may, of course, have the same customer appearing more than once. Quel, like SQL, eliminates duplicates only if specifically requested to do so. If we write

range of *t* **is** *temp*
retrieve unique (*t.customer-name*)

Quel sorts *temp* and eliminates duplicates.

The strategy of using **append** allows us to perform unions in Quel. To perform a set difference $r - s$ (**minus** in SQL), we create a temporary relation representing *r* and delete tuples of this temporary relation that are also in *s*. To find all customers who have an account at the Perryridge branch but do not have a loan from the Perryridge branch, we write the following

range of *u* **is** *deposit*
retrieve into *temp* (*u.customer-name*)
where *u.branch-name* = "Perryridge"

At this point *temp* has all customers who have an account at the Perryridge branch, including those with a loan from that branch. We now delete those customers who have a loan.

FST-IBM000863

range of *s* **is** *borrow*
range of *t* **is** *temp*
delete (*t*)
where *s.branch-name* = "Perryridge" **and**
t.customer-name = *s.customer-name*

The relation *temp* contains the desired list of customers. We write

range of *t* **is** *temp*
retrieve (*t.customer-name*)

to complete our query.

Fortunately, there is a more natural way to express this query in Quel. First, however, we must introduce the Quel aggregate expressions, which take the form

<aggregate-operation> (*t.A* **where** *P*)

where <aggregate-operation> is one of **count**, **sum**, **avg**, **max**, **min**, or **any**, *t* is a tuple variable, *A* is an attribute, and *P* is a predicate similar to the **where** clause in a **retrieve**. An aggregate expression may appear anywhere a constant may appear.

Thus, to find the average account balance for all accounts at the Perryridge branch, we write

range of *t* **is** *deposit*
retrieve avg (*balance* **where** *branch-name* = "Perryridge")

Aggregates may appear in the **where** clause. Suppose we wish to find all accounts whose balance is higher than the average balance at the branch where the account is held. We write:

range of *u* **is** *deposit*
range of *t* **is** *deposit*
retrieve *t.account-number*
where *t.balance* > **avg** (*u.balance* **where**
u.branch-name = *t.branch-name*)

The above **avg** (...) expression computes the average balance of all accounts at the branch represented by *t*. Because expressions of this sort are frequent, Quel allows the syntax:

range of *t* **is** *deposit*
retrieve *t.account-number*
where *t.balance* > **avg** (*t.balance* **by** *t.branch-name*)

The **avg** (...) expression performs the same computation as above. For a given *t*, the average balance is computed of the set of all tuples having the same value on the *branch-name* attribute as *t.branch-name*.

Let us return to the query "Find all customers who have an account at the Perryridge branch but do not have a loan from the Perryridge branch." We can write this query using the **count** aggregate operation if we think of the query as "Find all customers who have an account at the Perryridge branch and for whom the count of the number of loans from the Perryridge branch is zero."

range of *t* **is** *deposit*
range of *u* **is** *borrow*
retrieve *t.customer-name*
where *t.branch-name* = "Perryridge" and
count (*u.loan-number* **where** *u.branch-name* = "Perryridge"
and *u.customer-name* = *t.customer-name*) = 0

This is a more natural way to express this query than our earlier example.

Quel offers another aggregate operation that is applicable to this example, called **any**. If we replace **count** in the above query with **any**, we obtain 1 if the count is greater than 0; otherwise we obtain 0. The advantage in using **any** is that processing can stop as soon as one tuple is found. This allows faster execution of the query.

As a more complicated example, consider the query "Find all customers who have an account at all branches located in Brooklyn." Our strategy for expressing this query in Quel is as follows: First find out how many branches there are in Brooklyn. Then compare this number with the number of distinct branches in Brooklyn at which each customer has an account. The **count** aggregate operation we used earlier counts duplicates. Therefore, we use the **countu** operation, which counts unique values.

range of *t* **is** *deposit*
range of *u* **is** *branch*
range of *s* **is** *branch*
retrieve *t.customer-name*
where countu (*s.branch-name* **where** *s.branch-city* = "Brooklyn"
and *s.branch-name* = *t.branch-name*) =
countu (*u.branch-name* **where** *u.branch-city* = "Brooklyn")

We have observed that Quel is related closely to the tuple relational calculus. The **range of** clause corresponds to the "there exists." However, there is no analog in Quel to "for all." That is why we needed to use insertion and deletion to state in Quel some of the queries that we could write in the tuple relational calculus. To see more clearly the relationship between Quel and the tuple relational calculus, consider the following Quel query

range of t_1 is r_1
range of t_2 is r_2
range of t_m is r_m
retrieve $(t_{i_1}.A_{j_1}, t_{i_2}.A_{j_2}, \ldots, t_{i_n}.A_{j_n})$
where P

The above Quel query would be expressed in the tuple relational calculus as:

$$\{t \mid \exists\, t_1, t_2, \ldots, t_m (t_1 \in r_1 \wedge t_2 \in r_2 \wedge t_m \in r_m \wedge t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t[r_{i_2}.A_{j_2}] = t_{i_2}[A_{j_2}] \wedge \ldots \wedge t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}] \wedge P(t_1, t_2, \ldots, t_m))\}$$

This expression can be understood by looking at the formula within the "there exists" formula in three parts:

- $t_1 \in r_1 \wedge t_2 \in r_2 \wedge \ldots \wedge t_m \in r_m$. This part constrains each tuple in $t_1, t_2, \ldots, t_m$ to take on values of tuples in the relation it ranges over.
- $t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t_{i_2}[A_{j_2}] = t[r_{i_2}.A_{j_2}] \wedge \ldots \wedge t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}]$. This part corresponds to the **retrieve** clause of the Quel query. We need to ensure that the kth attribute in tuple t corresponds to the kth entry in the **retrieve** clause. Consider the first entry: $t_{i_1}.A_{j_1}$. This is the value of some tuple of r_{i_1} (since range of t_{i_1} is r_{i_1}) on attribute A_{j_1}. Thus, we need $t[A_{j_1}] = t_{i_1}[A_{j_1}]$. We used the more cumbersome notation $t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}]$ to be able to deal with the possibility that the same attribute name appears in more than one relation.
- $P(t_1, t_2, \ldots, t_m)$. This part is the constraint on acceptable values for $t_1, t_2, \ldots, t_m$ imposed by the **where** clause in the Quel query.

3.3.3 Query-by-Example

Query-by-Example (QBE) is the name of both a query language and the database system which includes this language. There are two distinctive features of QBE. Unlike most query languages and programming languages, QBE has a *two-dimensional* syntax. A query in a one-dimensional

branch	*branch-name*	*assets*	*branch-city*

customer	*customer-name*	*street*	*customer-city*

borrow	*branch-name*	*loan-number*	*customer-name*	*amount*

deposit	*branch-name*	*account-number*	*customer-name*	*balance*

Figure 3.13 QBE skeleton tables for the bank example.

language (for example, SQL or Quel) *can* be written in one (possibly very long) line. A two-dimensional language *requires* two dimensions for its expression. (There does exist a one-dimensional version of QBE. We shall not consider this version in our discussion of QBE.) The second distinctive feature of QBE is that queries are expressed "*by example*." Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired. The system generalizes this example to compute the answer to the query. Despite these unusual features, there is a close correspondence between QBE and the domain relational calculus.

Queries in QBE are expressed using *skeleton tables*. These tables show the relation scheme, and appear as in Figure 3.13. Rather than clutter the display with all skeletons, the user selects those skeletons needed for a given query. The user fills in these skeletons with "*example rows*." An example row consists of constants and "*example elements*." An example element is really a domain variable. To distinguish domain variables from constants, domain variables are preceded by an underscore character (" _ ") as in _ *x*. Constants appear without any qualification. This is in contrast to most other languages in which constants are quoted and variables appear without any qualification.

To find all customers having an account at the Perryridge branch, we bring up the skeleton for the *deposit* relation and fill it in as follows:

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P._*x*	

The above query causes the system to look for tuples in *deposit* that have "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the value of the *customer-name* attribute is assigned to the variable *x*. The value of the variable *x* is "printed" (actually displayed) because the command "P." appears in the *customer-name* column next to the variable *x*.

Unlike Quel and SQL, QBE performs duplicate elimination automatically. To suppress duplicate elimination, the command "ALL." is inserted after the "P." command.

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P.ALL._*x*	

The primary purpose of variables in QBE is to force values of certain tuples to have the same value on certain attributes. Suppose we wish to find all customers having a loan from the Perryridge branch, and their cities. We write:

borrow	*branch-name*	*loan-number*	*customer-name*	*amount*
	Perryridge		_*x*	

customer	*customer-name*	*street*	*customer-city*
	P._*x*		P._*y*

To execute the above query, the system finds tuples in *borrow* with "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the system finds tuples in *customer* with the same value for the *customer-name* attribute as the *borrow* tuple. The values for the *customer-name* and *customer-city* attributes are displayed. Observe that this is similar to what would be done to answer the domain relational calculus query:

$$\{<x,y> \mid \exists\, s\, (<x,s,y> \in customer)\}$$

A technique similar to the one above can be used to write the query "Find all customers who have both an account and a loan at the Perryridge branch":

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P._*x*	

borrow	*branch-name*	*loan-number*	*customer-name*	*amount*
	Perryridge		_*x*	

Suppose our query involves a less than or greater than comparison, rather than an equality comparison, as in "Find all account numbers with a balance of more than $1200":

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
		P._*x*		>1200

Until now, all the conditions we have imposed were connected by "*and*." To express an "*or*" in QBE, we give a separate example row for the two conditions being "*or*"-ed, using distinct domain variables. Consider the query "Find all customers having an account at the Perryridge branch, the Redwood branch, or both":

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P._*x*	
	Redwood		P._*y*	

Contrast the above query with "Find all customers having an account at *both* the Perryridge branch and the Redwood branch":

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P._*x*	
	Redwood		_*x*	

The critical distinction between these two queries is the use of the same domain variable (*x*) for both rows in the latter query, while, in the former query, we used distinct domain variables (*x* and *y*). To illustrate this, note that in the domain relational calculus, the former query would be written as

$$\{<x> \mid \exists\, b,a,n\ (<b,a,x,n> \in deposit \wedge b = \text{"Perryridge"}) \vee \exists\, b,a,n\ (<b,a,x,n> \in deposit \wedge b = \text{"Redwood"})\}$$

while the latter query would be written as

$$\{<x> \mid \exists\, b,a,n\ (<b,a,x,n> \in deposit \wedge b = \text{"Perryridge"}) \wedge \exists\, b,a,n\ (<b,a,x,n> \in deposit \wedge b = \text{"Redwood"})\}$$

Queries that involve negation are expressed in QBE by placing a not sign (¬) in a table skeleton under the relation name and next to an example row.

Let us now consider the query "Find all customers who have an account at the Perryridge branch but do not have a loan from that branch":

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	Perryridge		P. _x	

borrow	*branch-name*	*loan-number*	*customer-name*	*amount*
¬	Perryridge		_x	

Compare the above query with our earlier query "Find all customers who have both an account and a loan at the Perryridge branch." The only difference is the "¬" appearing next to the example row in the borrow skeleton. This difference, however, has a major effect on the processing of the query. QBE finds all x values for which

1. There is a tuple in the *deposit* relation in which *branch-name* is "Perryridge" and *customer-name* is the domain variable x.
2. There is no tuple in the *borrow* relation in which *branch-name* is "Perryridge" and *customer-name* is the same as in the domain variable x.

The "¬" can be read as "there does not exist."

The fact that we placed the "¬" under the relation name rather than under an attribute name is important. Use of a "¬" under an attribute name is a shorthand for "≠." To find all customers who have accounts at two different branches, we write

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	_y		P. _x	
	¬ _y		_x	

In English, the above query reads "display all *customer-name* values that appear in at least two tuples, with the second tuple having a *branch-name* different from the first."

If the result of a query is spread over several tables, we need a mechanism to display this result in a single table. To accomplish this, we can declare a temporary *result* table which includes all the attributes of the result relation. Printing of the desired result is done by including the command "P." only in the result table.

It is inconvenient at times to express all the constraints on the domain variables within the table skeletons. QBE includes a *condition box* feature that allows the expression of such constraints. Suppose we modify the above query to "Find all customers not named "Jones" who have accounts at two different branches." We want to include an "$x \neq$ Jones" constraint in the above query. We do that by bringing up the condition box and entering the constraint "$x \neq$ Jones":

conditions
$x \neq$ Jones

QBE includes aggregate operations similar to those of SQL and Quel. To find the average balance at all branches, we may write:

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	P.G			P.avg.ALL

Besides **avg**, the aggregate operators **max**, **min**, **count**, and **sum** are included in QBE. The "G" in the "P.G" entry in the *branch-name* column is analogous to SQL's "**group by** *branch-name*" construct. The average balance is computed on a branch-by-branch basis. The "ALL" in the "P.avg.ALL" entry in the balance column ensures that all balances are considered (recall that QBE eliminates duplicates by default).

3.4 Modifying the Database

We have restricted our attention until now to the extraction of information from the database. We have not, however, shown how to add new information, remove information, or change information. While we did do some **insert** and **delete** operations in our Quel examples, we never altered the database. Instead, we dealt with temporary relations constructed for the sole purpose of helping us to express the query.

The formal query languages (the relational algebra and the relational calculi) do not include any provision for modifying the database. All commercial languages do include such features, but we shall restrict our attention to examples in SQL.

3.4.1 Deletion

Deletion of tuples from a relation is simple. A delete request is expressed in much the same way as a query. However, instead of displaying tuples to the user, the selected tuples are removed from the database. We may delete only whole tuples; we cannot delete values on only particular attributes. In SQL, a deletion is expressed by

delete *r*
where *P*

P represents a predicate and *r* represents a relation. Those tuples *t* in *r* for which *P*(*t*) is true are deleted from *r*.

We note that a delete command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the where clause may be as complex as a select command's where clause. At the other extreme, we can have an empty where clause. The request:

delete *borrow*

deletes all tuples from the borrow relation. (Well-designed systems will seek confirmation from the user before executing such a devastating request.)

We give some examples of SQL delete requests:

- Delete all of Smith's account records.

delete *deposit*
where *customer-name* = "Smith"

- Delete all loans with loan numbers between 1300 and 1500.

delete *borrow*
where *loan-number* > 1300 **and** *loan-number* < 1500

- Delete all accounts at branches located in Needham

delete *deposit*
where *branch-name* **in** (**select** *branch-name*
from *branch*
where *branch-city* = "Needham")

The above **delete** request first finds all branches in Needham, and then deletes all *deposit* tuples pertaining to those branches.

Note that although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** embedded in the where clause of a delete.

If the delete request contains an embedded **select** that references the relation from which tuples are to be deleted, we face potential anomalies. Suppose we want to delete the records of all accounts with balances below the average. We might write

delete *deposit*
where *balance* < (**select avg** (*balance*)
from *deposit*)

However, as we delete tuples from deposit, the average balance changes! If we reevaluate the select for each tuple in deposit, the final result will depend upon the order in which we process tuples of *deposit*!

Such ambiguities are avoided by the following simple rule: During the execution of a delete request, we only mark tuples to be deleted; we do not actually delete them. Once we have finished processing the request, that is, once we are done marking tuples, then we delete all marked tuples. This rule guarantees a consistent interpretation of deletion. Thus, our delete request above does, in fact, work the way we would hope and expect. (Some implementations of SQL simply disallow delete requests like the above one.)

3.4.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity.

The simplest **insert** is a request to insert one tuple. Suppose we wish to insert the fact that Smith has $1200 in account 9732 at the Needham branch. We write

insert into *deposit*
values ("Needham", 9732, "Smith", 1200)

More generally, we might want to insert tuples based on the result of a query. Suppose that we want to provide all loan customers in the Needham branch with a $200 savings account. Let the loan number serve as the account number for the new savings account. We write

insert into *deposit*
select *branch-name*, *loan-number*, *customer-name*, 200
from *borrow*
where *branch-name* = "Needham"

Instead of specifying a tuple as we did earlier, we use a **select** to specify a set of tuples. Each tuple has the *branch-name* (Needham), a *loan-number* (which serves as the account number for the new account), the name of the loan customer who is being given the new account, and the initial balance of the new account, $200.

3.4.3 Updating

There are situations in which we wish to change a value in a tuple without changing *all* values in the tuple. If we make these changes using **delete** and **insert**, we may not be able to retain those values that we do not wish to change. Instead, we use the **update** statement. As was the case for **insert** and **delete**, we may choose the tuples to be updated using a query.

Suppose interest payments are being made, and all balances are to be increased by 5 percent. We write

update *deposit*
set *balance* = *balance* * 1.05

The above statement is applied once to each tuple in *deposit*.

Let us now suppose that accounts with balances over $10,000 receive 6 percent interest, while all others receive 5 percent. We write two **update** statements:

update *deposit*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *deposit*
set *balance* = *balance* * 1.05
where *balance* ≤ 10000

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **selects**). Note that in the above example the order in which we wrote the two **update** statements is important. If we changed the order of the two statements, an account whose balance is just under $10,000 would receive 11.3 percent interest!

3.5 Views

In our examples up to this point, we have operated at the conceptual model level. That is, we have assumed that the collection of relations we are given are the actual relations stored in the database.

It is not desirable for all users to see the entire conceptual model. Security considerations may require that we "hide" certain data from certain users. Consider, for example, a clerk who needs to know a customer's loan number but has no need to see the loan amount. This clerk should see a relation described, in the relational algebra, by

$$\Pi_{\textit{branch-name, loan-number, customer-name}}\ (\textit{borrow})$$

Aside from security concerns, we may wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the conceptual model. An employee in the advertising department, for example, might like to see a relation consisting of the customers of each branch; that is, for each branch we would like to list those people who have either an account or a loan at that branch. The relation we would like to create for the employee is

$$\Pi_{\textit{branch-name, customer-name}}\ (\textit{deposit})$$
$$\cup\ \Pi_{\textit{branch-name, customer-name}}\ (\textit{borrow})$$

We use the term *view* to refer to any relation not part of the conceptual model that is made visible to a user as a "virtual relation." It is possible to support a large number of views on top of any given set of actual relations.

Since the actual relations in the conceptual model may be modified by **insert**, **update**, or **delete** operations, it is not generally possible to store views. Instead, a view must be recomputed for each query that refers to it. In Chapter 9 we shall consider techniques for reducing the overhead of this recomputation. For now, we restrict our attention to the definition and use of views in SQL.

A view is defined in SQL using the **create view** command. To define a view, we must give the view a name and state the query that computes the view. The form of the **create view** command is

create view *v* **as** <query expression>

where <query expression> is any legal query expression. The view name is represented by *v*.

As an example, consider the view consisting of branches and their customers. Assume we wish this view to be called *all-customer*. We define this view as follows:

create view *all-customer* **as**
 (**select** *branch-name, customer-name*
 from *deposit*)
 union
 (**select** *branch-name, customer-name*
 from *borrow*)

Once we have defined a view, the view name can be used to refer to the virtual relation the view generates. View names may appear in any place that a relation name may appear. Using the view *all-customer*, we can find all customers of the Perryridge branch by writing

select *customer-name*
from *all-customer*
where *branch-name* = "Perryridge"

Recall that we wrote the same query in Section 3.3 without using views.

Although views are a useful tool for queries, they present significant problems if updates, insertions, or deletions are expressed using views. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the conceptual model of the database. We illustrate the problem of database modification through views with a simple example.

Consider the clerk we discussed earlier who needs to see all loan data in the *borrow* relation except *loan-amount*. Let *loan-info* be the view given to the clerk. We define this view as

create view *loan-info* **as**
  select *branch-name, loan-number, customer-name*
  from *borrow*

Since SQL allows a view name to appear wherever a relation name is allowed, the clerk may write

insert into *loan-info*
  values ("Perryridge", 3, "Ruth")

This insertion must be represented by an insertion into the relation *borrow*, since *borrow* is the actual relation from which the view *loan-info* is constructed. However, to insert a tuple into *borrow*, we must have some value for *amount*. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion and return an error message to the user.
- Insert a tuple ("Perryridge", 3, "Ruth", *null*) into the *borrow* relation.

The symbol *null* represents a *null-value*, or *place-holder value*. It signifies that the value is unknown or does not exist.

Most systems take the latter approach and create null values. However, the presence of null values adds complexity to database queries.

Assume we have inserted the above tuple, producing the relation shown in Figure 3.14. Consider the following query to total all loan balances:

select sum (*amount*)
from *borrow*

It is not possible to perform addition using *null*. Similar problems arise using other aggregate operators. As a result, all aggregate operations except **count** ignore tuples with null values on the argument attributes.

All comparisons involving *null* are false by definition. However, a special keyword, **null** may be used in a predicate to test for a null value. To find all customers who appear in the *borrow* relation with null values for *balance*, we write

select *customer-name*
from *borrow*
where *balance* **is null**

The predicate **is not null** tests for the absence of a null value.

We illustrate another problem resulting from modification of the database through views with an example involving the following view:

create view *branch-city* **as**
  select *branch-name, customer-city*
  from *borrow, customer*
  where *borrow.customer-name* = *customer.customer-name*

branch-name	*loan-number*	*customer-name*	*amount*
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200
Perryridge	3	Ruth	*null*

Figure 3.14 A *borrow* relation containing null values.

This view lists the cities in which borrowers of each branch live. Consider the following insertion through this view:

```
insert into branch-city
       values ("Brighton", "Woodside")
```

The only possible method of inserting tuples into the *borrow* and *customer* relations is to insert ("Brighton", *null, null, null*) into *borrow* and (*null, null,* "Woodside") into *customer*. Suppose the system did that. Then we obtain the relations shown in Figure 3.15. This turns out to be unsatisfactory since

```
select *
from branch-city
```

does *not* include the tuple ("Brighton", "Woodside"). To see why this is so, recall that all comparisons involving **null** are defined to be **false**. Thus, the **where** clause in the view definition (*borrow.customer-name* = *customer.customer-name*) is never satisfied for the tuples added to the *borrow* and *customer* relations.

As a result of the anomaly we have just discussed, many database systems impose the following constraint on modifications allowed through views:

- A modification is permitted through a view only if the view in question is defined in terms of one relation of the actual relational database.

Under this constraint, **update**, **insert**, and **delete** operations would be forbidden on the example views *branch-city* and *all-customer* that we defined above.

The general problem of database modification through views is a subject of current research. The bibliographic notes mention recent works on this subject.

Another view-related research area of interest is the *universal relation* model. In this model, the user is given a view consisting of one relation. This one relation is the natural join of all relations in the actual relational database. The major advantage of this model is that users need not be concerned with remembering what attributes are in which relation. Thus, most queries are easier to formulate in a universal-relation database system than in a standard relational database system. For example, a universal-relation version of SQL would not need a **from** clause.

There remain unresolved questions regarding modifications to universal relation databases. Furthermore, a consensus has not yet developed on the best definition of the meaning of certain complex types of universal-relation queries.

branch-name	*loan-number*	*customer-name*	*amount*
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200
Brighton	*null*	*null*	*null*

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford
null	*null*	Woodside

Figure 3.15 Tuples inserted into *borrow* and *customer*.

We can summarize our discussion of views briefly as follows. Views are a useful mechanism for simplifying database queries, but modification of the database through views has potentially disadvantageous consequences. A strong case can be made for requiring all database modifications to refer to actual relations in the database.

3.6 Summary

The relational data model is based on a collection of tables. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations. The tuple relational calculus and the domain relational calculus are nonprocedural languages that represent the basic power required in a relational query language. The relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus. The algebra defines the basic operations used within relational query languages.

The relational algebra and the relational calculi are terse, formal languages that are inappropriate for casual users of a database system. Commercial database system have, therefore, used languages with more "syntactic sugar." These languages include constructs for update, insertion, and deletion of information as well for querying the database. We have considered the three most influential of the commercial languages: SQL, Quel, and QBE.

Different users of a shared database may benefit from individualized views of the database. We used SQL as an example to show how such views can be defined and used.

Exercises

3.1 Design a relational database for a university registrar's office. The office maintains data about each class, including the instructor, the enrollment, and the time and place of the class meetings. For each student-class pair, a grade is recorded.

3.2 Describe the differences between the terms *relation* and *relation scheme*. Illustrate your answer be referring to your solution to Exercise 3.1.

3.3 Design a relational database corresponding to the E-R diagram of Figure 3.16.

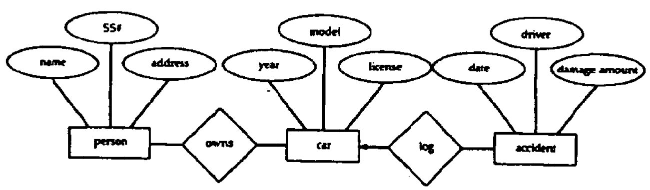

Figure 3.16 E-R diagram.

3.4 Construct the following SQL queries for the relational database of Exercise 3.3.

a. Find the total number of persons whose car was involved in an accident in 1983.

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

c. Add a new customer to the database.

d. Delete the car "Mazda" belonging to "John Smith."

e. Add a new accident record for the Toyota belonging to "Jones."

3.5 Consider the relational database of Figure 3.17. Give an expression in:

- The relational algebra
- The tuple relational calculus
- The domain relational calculus
- SQL
- Quel
- QBE

for each of the queries below:

a. Find the name of all people who work for First Bank Corporation.

b. Find the name and city of all people who work for First Bank Corporation.

c. Find the name, street, and city of all people who work for First Bank Corporation and earn more than $10,000.

lives (*person-name, street, city*)
works (*person-name, company-name, salary*)
located-in (*company-name, city*)
manages (*person-name, manager-name*)

Figure 3.17 Relational database.

d. Find all people who live in the same city as the company they work for.

e. Find all people who live in the same city and on the same street as their manager.

f. Find all people who do not work for First Bank Corporation.

g. Find all people who earn more than every employee of Small Bank Corporation.

h. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

3.6 Consider the relational database of Figure 3.17. Give an expression in:

- SQL
- Quel
- QBE

for each of the queries below:

a. Find all people who earn more than the average salary of people working in their company.

b. Find the company employing the most people.

c. Find the company with the smallest payroll.

d. Find those companies that pay more, on average, than the average salary at First Bank Corporation.

3.7 Consider the relational database of Figure 3.17. Give an expression in SQL for each query below:

a. Modify the database so that Jones now lives in Newtown.

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

c. Give all managers a 10 percent raise.

d. Give all managers a 10 percent raise unless the salary becomes greater than $100,000. In such cases, give only a 3 percent raise.

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

3.8 In Chapter 2, we showed how to represent many-to-many, many-to-one, one-to-many, and one-to-one relationship sets. Explain how primary keys help us to represent such relationship sets in the relational model.

3.9 Let the following relation schemes be given:

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

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

a. $\Pi_A(r)$

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

c. $r \times s$

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

3.10 Let $R = (A,B,C)$ and let r_1 and r_2 both be relations on scheme R. Give an expression in the domain relational calculus that is equivalent to:

a. $\Pi_A(r_1)$

b. $\sigma_{B\,=\,17}\,(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

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

3.11 Let $R = (A,B)$ and $S = (A,C)$, and let $r(R)$ and $s(S)$ be relations. Write relational algebra expressions equivalent to the following domain relational calculus expressions.

a. $\{\langle a \rangle \mid \exists\, b\,(\langle a,b \rangle \in r \wedge b = 17)\}$

b. $\{\langle a,b,c \rangle \mid \langle a,b \rangle \in r \wedge \langle a,c \rangle \in s)\}$

c. $\{\langle b \rangle \mid \forall\, a\,(\langle a,b \rangle \notin r \vee \exists\, c\,(\langle a,c \rangle \in s)\}$

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

3.12 Write expressions for the queries of Exercise 3.11 in

a. QBE

b. QUEL

c. SQL

3.13 Consider the relational database of Figure 3.17. Using SQL define a view consisting of *manager-name* and the average salary of employees working for that manager. Explain why the database system should not allow updates to be expressed in terms of this view.

3.14 List reasons why null values may be introduced into the database.

3.15 Some systems allow *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \neq j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *branch-city* (Section 3.5). Show how marked nulls can be used to allow the insertion of the tuple (Brighton, Woodside) through *branch-city*.

Bibliographic Notes

The relational model was proposed by E. F. Codd of the IBM San Jose Research Laboratory in the late 1960s [Codd 1970]. Following Codd's original paper, several research projects were formed with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory, Ingres at the University of California at Berkeley, Query-by-Example at the IBM T. J. Watson Research Center, and PRTV (Peterlee Relational Test Vehicle) at the IBM Scientific Center in Peterlee, United Kingdom. System R and Ingres are discussed in Chapter 15. The bibliographic notes of that chapter provide references to those systems. Query-by-Example is described in Zloof [1977] and IBM [1978b]. PRTV is described in Todd [1976]. The original definition of the relational algebra is in Codd [1970] and that of the relational calculi is in Codd [1972b]. A formal proof of the equivalence of the relational calculus and relational algebra can be found in Codd [1972b] and Ullman [1982a].

The query language SQL was first defined by Chamberlin et al. [1976]. Current versions of SQL available in commercial systems include those described by IBM [1982] and Oracle [1983]. There is a proposal in progress under the auspices of the American National Standards Institute (ANSI) for a standard SQL language. The SQL language served as the basis for a proposal for a more general relational database language being developed

by ANSI Committee X3H2 (Committee on Computer and Information Processing).

Quel is defined by Stonebraker et al. [1976], Wong and Youssefi [1976], and Zook et al. [1977]. A commercial version of Quel is described in RTI [1983].

The problem of updating relational databases through views is addressed by Cosmadakis and Papadimitriou [1984], Dayal and Bernstein [1978, 1982], and Keller [1982, 1985]. The universal relation view is discussed by Sciore [1980], Fagin et al. [1982] and Ullman [1982a, 1982b]. Several experimental database systems have been built to test the claim that a universal relation view is simpler to use. In such systems, the user views the entire database as one relation and the system translates operations on the universal relation view into operations on the set of relations forming the conceptual scheme. One such system is System/U, which was developed at Stanford University in 1980-1982. System/U is described by Ullman[1982a, 1982b] and Korth et al. [1984]. The System/U query language is similar to that of Quel. However, since the user sees only one relation, the from clause is eliminated. Another universal relation system, PITS, is discussed by Maier et al. [1981] and Maier [1983].

General discussion of the relation data model appears in most database texts, including Ullman [1982a] and Date [1986]. Maier [1983] is a text devoted exclusively to the relational data model.

9

Query Processing

In the preceding sections, we have considered how to structure the data in the database. These decisions are made at the time the database is designed. Although it is possible to change this structure, it is relatively costly to do so. Thus, when a query is presented to the system, it is necessary to find the best method of finding the answer using the existing database structure. There are a large number of possible strategies for processing a query, especially if the query is complex. Nevertheless, it is usually worthwhile for the system to spend a substantial amount of time on the selection of a strategy. Typically, strategy selection can be done using information available in main memory, with little or no disk accesses. The actual execution of the query will involve many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the central processor of the computer system, it is advantageous to spend a considerable amount of processing to save disk accesses.

9.1 Query Interpretation

Given a query, there are generally a variety of methods for computing the answer. For example, we saw that in SQL a query could be expressed in several different ways. Each way of expressing the query "suggests" a strategy for finding the answer. However, we do not expect users to write their queries in a way that suggests the most efficient strategy. Thus, it becomes the responsibility of the system to transform the query as entered by the user into an equivalent query which can be computed more efficiently. This "optimizing," or more accurately, improving of the strategy for processing a query, is called *query optimization*. There is a close analogy between code optimization by a compiler and query optimization by a database system. We shall study the issues involved in efficient query processing both in high-level languages and at the level of physical access to the data.

Query optimization is an important issue in any database system since the difference in execution time between a good strategy and a bad one may be huge. In the network model and the hierarchical model, query optimization is left, for the most part, to the application programmer. Since the data manipulation language statements are embedded in a host programming language, it is not easy to transform a network or hierarchical query to an equivalent one unless one has knowledge about the entire application program.

Since a relational query can be expressed entirely in a relational query language without the use of a host language, it is possible to optimize queries automatically. Since the most useful optimization techniques apply to the relational model, we shall emphasize the relational model in this chapter. The bibliographic notes reference techniques for optimization of network and hierarchical queries.

Before query processing can begin, the system must translate the query into a usable form. Languages such as SQL are suitable for human use, but ill-suited to be the system's internal representation of a query. A more useful internal representation of query is one based on the relational algebra. The only difference between the form of the relational algebra we shall use here and that of Chapter 3 is that we shall add redundant parentheses to indicate the order of operation evaluation.

Thus, the first action the system must take on a query is to translate the query into its internal form. This translation process is similar to that done by the parser of a compiler. In the process of generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relation in the database, etc. If the query was expressed in terms of a view, the parser replaces all references to the view name with the relational algebra expression to compute a view.

The details of the parser are beyond the scope of this text. Parsing is covered in most compiler texts (see the bibliographic notes).

Once the query has been translated to an internal relational algebra form, the optimization process begins. The first phase of optimization is done at the relational algebra level. An attempt is made to find an expression that is equivalent to the given expression but that is more efficient to execute. The next phase involves the selection of a detailed strategy for processing the query. A choice must be made as to exactly how the query will be executed. A choice of specific indices to use must be made. The order in which tuples are processed must be determined. The final choice of a strategy is based primarily on the number of disk accesses required.

9.2 Equivalence of Expressions

The relational algebra is a procedural language. Thus, each relational algebra expression represents a particular sequence of operations. We have already seen that there are several ways to express a given query in the relational algebra. The first step in selecting a query processing strategy is to find a relational algebra expression that is equivalent to the given query and is efficient to execute.

We use our bank example to illustrate optimization techniques. In particular, we shall use the relations *customer* (*Customer-scheme*), *deposit* (*Deposit-scheme*), and *branch* (*Branch-scheme*). As was the case earlier, we define our relation scheme as follows:

Customer-scheme = (*customer-name, street, customer-city*)
Deposit-scheme = (*branch-name, account-number, customer-name, balance*)
Branch-scheme = (*branch-name, assets, branch-city*)

9.2.1 Selection Operation

Let us consider the relational algebra expression we wrote in Chapter 3 for the query "Find the assets and name of all banks who have depositors living in Port Chester":

$$\Pi_{branch\text{-}name,\ assets}\ (\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (customer \bowtie deposit \bowtie branch))$$

This expression constructs a large relation, *customer* ⋈ *deposit* ⋈ *branch*. However, we are interested in only a few tuples of this relation (those pertaining to residents of Port Chester), and in only two of the eight attributes of this relation. The large intermediate result:

$$customer \bowtie deposit \bowtie branch$$

is probably too large to be kept in main memory and thus must be stored on disk. This means that in addition to the disk accesses required to read the relations *customer*, *deposit*, and *branch*, the system will need to access disk to read and write intermediate results. Clearly, we could process the query more efficiently if there were a way to reduce the size of the intermediate result.

Since we are concerned only about tuples for which *customer-city* = "Port Chester," we need not consider those tuples of the *customer* relation that do not have *customer-city* = "Port Chester." By reducing the number of tuples of the *customer* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational algebra expression:

$$\Pi_{branch\text{-}name,\ assets}\ ((\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (customer)) \bowtie deposit \bowtie branch)$$

The above example suggests the following rule for transforming relational algebra queries:

- Perform selection operations as early as possible.

In our example, we recognized that the selection operator pertained only to the *customer* relation, so we performed the selection on *customer* directly.

Suppose that we modify our original query to restrict attention to customers with a balance over $1000. The new relational algebra query is

$$\Pi_{branch\text{-}name,\ assets}\ (\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}\ \wedge\ balance\ >\ 1000}\ (customer \bowtie deposit \bowtie branch))$$

We cannot apply the selection:

$$customer\text{-}city = \text{"Port Chester"} \wedge balance > 1000$$

directly to the *customer* relation, since the predicate involves attributes of *customer* and *deposit*. However, the *branch* relation does not involve either *customer-city* or *balance*. If we decide to process the join as:

$$((customer \bowtie deposit) \bowtie branch)$$

then we can rewrite our query as:

$$\Pi_{branch\text{-}name,\ assets}\ ((\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}\ \wedge\ balance\ >\ 1000}\ (customer \bowtie deposit)) \bowtie branch)$$

Let us examine the subquery:

$$\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}\ \wedge\ balance\ >\ 1000}\ (customer \bowtie deposit)$$

We can split the selection predicate into two, forming the expression:

$$\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (\sigma_{balance\ >\ 1000}\ (customer \bowtie deposit))$$

Both of the above expressions select tuples with *customer-city* = "Port Chester" and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the "perform selections early" rule. We now rewrite our query as:

$$(\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (customer)) \bowtie (\sigma_{balance\ >\ 1000}\ (deposit))$$

We now add a second transformation rule:

- Replace expressions of the form:

$$\sigma_{P_1 \wedge P_2}\ (e)$$

by

$$\sigma_{P_1}\ (\sigma_{P_2}\ (e))$$

where P_1 and P_2 are predicates and e is a relational algebra expression.

An easy way to remember this transformation is by noting the following equivalences among relational algebra expressions:

$$\sigma_{P_1}\ (\sigma_{P_2}\ (e)) = \sigma_{P_2}\ (\sigma_{P_1}\ (e)) = \sigma_{P_1 \wedge P_2}\ (e)$$

9.2.2 Natural Join Operation

By modifying queries so that selections are done early, we reduce the size of temporary results. Another way to reduce the size of temporary results is to choose an optimal ordering of the join operations. We mentioned in Chapter 3 that natural join is associative. Thus, for all relations r_1, r_2, and r_3:

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

although these expressions are equivalent, the costs of computing them may differ. Consider again the expression:

$$\Pi_{branch\text{-}name,\ assets}\ ((\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (customer)) \bowtie deposit \bowtie branch)$$

We could choose to compute *deposit* ⋈ *branch* first and then join the result with:

$$\sigma_{customer\text{-}city\ =\ \text{"Port Chester"}}\ (customer)$$

However, *deposit* ⋈ *branch* is likely to be a large relation since it contains one tuple for every account. However,

$$\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)$$

is probably a small relation. To see this, note that since the bank has a large number of widely distributed branches, it is likely that only a small fraction of the bank's customers live in Port Chester. If we compute:

$$(\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)) \bowtie deposit$$

first, we obtain one tuple for each account held by a resident of Port Chester. Thus, the temporary relation we must store is smaller than if we compute *deposit* ⋈ *borrow* first.

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations r_1 and r_2:

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

That is, natural join is commutative.

Using this fact, we can consider rewriting our relational algebra expression as

$$\Pi_{branch\text{-}name,\ assets}\ (((\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)) \bowtie branch) \bowtie deposit)$$

That is, we could join $\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)$ with *branch* as the first join operation performed. Note, however, that there are no attributes in common between *Branch-scheme* and *Customer-scheme*, so the join is really just a cartesian product. If there are *c* customers in Port Chester and *b* branches, this cartesian product generates *bc* tuples, one for every possible pair of customers and branches (without regard for whether or not the customer has an account at the branch). Thus, it appears that this cartesian product will produce a large temporary relation. As a result, we would reject this strategy. However, if the user had entered the above expression, we could use the associativity and commutativity of natural join to transform this expression to the more efficient expression we used earlier.

9.2.3 Projection Operation

We now consider another technique for reducing the size of temporary results. The projection operation, like the selection operation, reduces the

size of relations. Thus, whenever we need to generate a temporary relation, it is advantageous to apply any projections that are possible. This suggests a companion to the "perform selections early" rule we stated earlier:

- Perform projections early.

Consider the following form of our example query:

$$\Pi_{branch\text{-}name,\ assets}\ (((\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)) \bowtie deposit) \bowtie branch)$$

When we compute the subexpression:

$$((\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)) \bowtie deposit)$$

we obtain a relation whose scheme is:

(*customer-name, customer-city, branch-name, account-number, balance*)

We can eliminate several attributes from the scheme. The only attributes we must retain are those that:

- Appear in the result of the query or
- Are needed to process subsequent operations.

By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, the size of the intermediate result is reduced. In our example, the only attribute we need is *branch-name*. Therefore, we modify the expression to:

$$\Pi_{branch\text{-}name,\ assets}\ ((\Pi_{branch\text{-}name}\ ((\sigma_{customer\text{-}city\ =\ \text{“Port Chester”}}\ (customer)) \bowtie deposit)) \bowtie branch)$$

9.2.4 Other Operations

The example we have used involves a sequence of natural joins. We chose this example because natural joins arise frequently in practice and because natural joins are one of the more costly operations in query processing. However, we note that equivalences similar to those presented above hold for the union and set difference operations. We list some of these equivalences below:

$$\sigma_P(r_1 \cup r_2) = \sigma_P(r_1) \cup \sigma_P(r_2)$$
$$\sigma_P(r_1 - r_2) = \sigma_P(r_1) - r_2 = \sigma_P(r_1) - \sigma_P(r_2)$$
$$(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$$
$$r_1 \cup r_2 = r_2 \cup r_1$$

We have seen several techniques for generating more efficient relational algebra expressions for a query. For queries whose structure is more complex than those of our example there may be a large number of possible strategies that appear to be efficient. Some query processors simply choose from such a set of strategies based on certain heuristics. Others retain all promising strategies and perform the latter phases of query optimization for each strategy. The final choice of strategy is made only after the details of each strategy have been worked out and an estimate is made of the processing cost of each strategy.

9.3 Estimation of Query-Processing Cost

The strategy we choose for a query depends upon the size of each relation and the distribution of values within columns. In the example we have used in this chapter, the fraction of customers who live in Port Chester has a major impact on the usefulness of our techniques. In order to be able to choose a strategy based on reliable information, database systems may store statistics for each relation r. These statistics include:

1. n_r, the number of tuples in the relation r.
2. s_r, the size of a record (tuple) of relation r in bytes (for fixed-length records).
3. $V(A,r)$, the number of distinct values that appear in the relation r for attribute A.

The first two statistics allow us to estimate accurately the size of a cartesian product. The cartesian product $r \times s$ contains $n_r n_s$ tuples. Each tuple of $r \times s$ occupies $s_r + s_s$ bytes.

The third statistic is used to estimate how many tuples satisfy a selection predicate of the form:

<attribute-name> = *<value>*

However, in order to perform such an estimation, we need to know how often each value appears in a column. If we assume that each value appears with equal probability, then $\sigma_{A=a}(r)$ is estimated to have

$n_r / V(A,r)$ tuples. However, it may not always be realistic to assume that each value appears with equal probability. The *branch-name* attribute in the *deposit* relation is an example of such a case. There is one tuple in the *deposit* relation for each amount. It is reasonable to expect that the large branches have more accounts than smaller branches. Therefore certain *branch-name* values appear with greater probability than others.

Despite the fact that our uniform distribution assumption is not always true, it is a good approximation of reality in many cases. Therefore, many query processors make such an assumption when choosing a strategy. For simplicity, we shall assume a uniform distribution for the remainder of this chapter.

Estimation of the size of a natural join is somewhat more complicated than estimation of the size of a selection or a cartesian product. Let $r_1(R_1)$ and $r_2(R_2)$ be relations. If $R_1 \cap R_2 = \emptyset$, then $r_1 \bowtie r_2$ is the same as $r_1 \times r_2$, and we can use our estimation technique for cartesian products. If $R_1 \cap R_2$ is a key for R_1, then we know that a tuple of r_2 will join with exactly one tuple from r_1. Therefore, the number of tuples in $r_1 \bowtie r_2$ is no greater than the number of tuples in r_2.

The most difficult case to consider is when $R_1 \cap R_2$ is a key for neither R_1 nor R_2. In this case, we use the third statistic and assume, as before, that each value appears with equal probability. Consider a tuple t of r_1, and assume $R_1 \cap R_2 = \{A\}$. We estimate that there are $n_{r_2} / V(A,r_2)$ tuples in r_2 with an A value of $t[A]$. So tuple t produces

$$\frac{n_{r_2}}{V(A,r_2)}$$

tuples in $r_1 \bowtie r_2$. Considering all of the tuples in r_1, we estimate that there are

$$\frac{n_{r_1} n_{r_2}}{V(A,r_2)}$$

tuples in $r_1 \bowtie r_2$. Observe that if we reverse the roles of r_1 and r_2 in the above estimate, we obtain an estimate of $n_{r_1} n_{r_2} / V(A,r_1)$ tuples in $r_1 \bowtie r_2$. These two estimates differ if $V(A,r_1) \neq V(A,r_2)$. If this situation occurs, there are likely to be some dangling tuples that do not participate in the join. Thus, the lower of the two estimates is probably the better one.

The above estimate of join size may be too high if the $V(A,r_1)$ A values in r_1 have few values in common with the $V(A,r_2)$ A values in r_2. However, it is unlikely that our estimate will be very far off in practice since dangling tuples are likely to be only a small fraction of the tuples in a real-world relation. If dangling tuples appear frequently, then a correction factor could be applied to our estimates.

If we wish to maintain accurate statistics, then every time a relation is modified, it is necessary also to update the statistics. This is a substantial amount of overhead. Therefore, most systems do not update the statistics on every modification. Instead, statistics are updated during periods of light load on the system. As a result, the statistics used for choosing a query processing strategy may not be accurate. However, if the interval between the update of the statistics is not too long, the statistics will be sufficiently accurate to provide a good estimation of the size of the results of expressions.

Statistical information about relations is particularly useful when several indices are available to assist in the processing of a query, as we shall see in Section 9.4.

9.4 Estimation of Costs of Access Using Indices

The cost estimates we have considered for relational algebra expressions did not consider the affects of indices and hash functions on the cost of evaluating an expression. The presence of these structures, however, has a significant influence on the choice of a query-processing strategy.

- Indices and hash functions allow fast access to records containing a specific value on the index key.
- Indices (though not most hash functions) allow the records of a file to be read in sorted order. In Chapter 8, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. If an index allows the records of a file to be read in an order that corresponds to the physical order of records, we call that index a *clustering index*. Clustering indices allow us to take advantage of the physical clustering of records into blocks.

The detailed strategy for processing a query is called an *access plan* for the query. A plan includes not only the relational operations to be performed but also the indices to be used and the order in which tuples are to be accessed and the order in which operations are to be performed.

Of course, the use of indices imposes the overhead of access to those blocks containing the index. We need to take these blocks accesses into account when we estimate the cost of a strategy that involves the use of indices.

In this section, we consider queries involving only one relation. We use the selection predicate to guide us in the choice of the best index to use in processing the query.

As an example of the estimation of the cost of a query using indices assume that we are processing the query:

select *account-number*
from *deposit*
where *branch-name* = "Perryridge" **and** *customer-name* = "Williams"
  and *balance* > 1000

Assume that we have the following statistical information about the *deposit* relation:

- 20 tuples of *deposit* fit in one block.
- V(*branch-name*, *deposit*) = 50.
- V(*customer-name*, *deposit*) = 200.
- V(*balance*, *deposit*) = 5000.
- The *deposit* relation has 10,000 tuples.

Let us assume that the following indices exist on *deposit*:

- A clustering, B^+-tree index for *branch-name*.
- A nonclustering, B^+-tree index for *customer-name*.

As before, we shall make the simplifying assumption that values are distributed uniformly.

Since V(*branch-name*, *deposit*) = 50, we expect that 10000/50 = 200 tuples of the *deposit* relation pertain to the Perryridge branch. If we use the index on *branch-name*, we will need to read these 200 tuples and check each one for satisfaction of the where clause. Since the index is a clustering index, 200/20 = 10 block reads are required to read the *deposit* tuples. In addition, several index blocks must be read. Assume the B^+-tree index stores 20 pointers per node. This means that the B^+-tree index must have between 3 and 5 leaf nodes. With this number of leaf nodes, the entire tree has a depth of 2, so at most 2 index blocks must be read. Thus the above strategy requires 12 total block reads.

If we use the index for *customer-name*, we estimate the number of block accesses as follows. Since V(*customer-name*, *deposit*) = 200, we expect that 10000/200 = 50 tuples of the *deposit* relation pertain to Williams. However, since the index for *customer-name* is nonclustering, we anticipate that one block read will be required for each tuple. Thus, 50 block reads are required, just to read the *deposit* tuples. Let us assume that 20 pointers fit into one node of the B^+-tree index for *customer-name*. Since there are 200 customer names, the tree has between 11 and 20 leaf nodes. So, as was the case for the other B^+-tree index, the index for *customer-name* has a depth of 2 and 2 block accesses are required to read the necessary index blocks. Therefore, this strategy requires a total of 52 block reads. We conclude that it is preferable to use the index for *branch-name*.

Observe that if both indices were nonclustering, we would prefer to use the index for *customer-name* since we expect only 50 tuples with *customer-name* = "Williams" versus 200 tuples with *branch-name* = "Perryridge." Without the clustering property, our first strategy would have required 200 block accesses to read the data plus 2 index block accesses for a total of 202 block reads. However, because of the clustering property of the *branch-name* index, it is actually less expensive in this example to use the *branch-name* index.

We did not consider using the *balance* attribute and the predicate *balance* $\geq$ 1000 as a starting point for a query processing strategy for two reasons:

- There is no index for *balance*.
- The selection predicate on *balance* involves a "greater than" comparison. In general, equality predicates are more selective than "greater than" predicates. Since we have an equality predicate available to us (indeed, we have two), we prefer to start by using such a predicate since it is likely to select fewer tuples.

Estimation of the cost of access using indices allows us to estimate the complete cost, in terms of block accesses, of a plan. For a given relational algebra expression, it may be possible to formulate several plans. The access plan selection phase of a query optimizer chooses the best plan for a given expression.

We have seen that different plans may have significant differences in cost. It is possible that a relational algebra expression for which a good plan exists may be preferable to an apparently more efficient algebra expression for which only inferior plans exist. Thus, it is often worthwhile for a large number of strategies to be evaluated down to the access plan level before a final choice of query-processing strategy is made.

9.5 Join Strategies

Earlier, we estimated the *size* of the result of a relational algebra expression involving a natural join. In this section, we apply our techniques for estimating the cost of processing a query to the problem of estimating the *cost* of processing a join. We shall see that several factors influence the selection of an optimal strategy:

- The physical order of tuples in a relation.
- The presence of indices and the type of index (clustering or nonclustering).
- The cost of computing a temporary index for the sole purpose of processing one query.

Let us begin by considering the expression

$$deposit \bowtie customer$$

and assume that we have no indices whatsoever. Let:

- $n_{deposit}$ = 10,000.
- $n_{customer}$ = 200.

9.5.1 Simple Iteration

If we are not willing to create an index, we must examine every possible pair of tuples t_1 in *deposit* and t_2 in *customer*. Thus, we examine $10000 * 200 = 2000000$ pairs of tuples.

If we execute this query cleverly, we can reduce the number of block accesses significantly. Suppose that we use the procedure of Figure 9.1 for computing the join. We read each tuple of *deposit* once. This may require as many as 10,000 block accesses. However, if the tuples of *deposit* are stored together physically, fewer accesses are required. If we assume that 20 tuples of *deposit* fit in one block, then reading *deposit* requires $10000/20$ = 500 block accesses.

```
for each tuple d in deposit do
  begin
    for each tuple c in customer do
      begin
        test pair (d,c) to see if a tuple should be added to the result
      end
  end
```

Figure 9.1 Procedure for computing join.

```
for each block Bd of deposit do
  begin
    for each block Bc of customer do
      begin
        for each tuple b in Bd do
          begin
            for each tuple c in Bc do
              begin
                test pair (b,c) to see if a tuple
                should be added to the result
              end
          end
      end
  end
```

Figure 9.2 Procedure to compute *deposit* ⋈ *customer* .

We read each tuple of *customer* once for each tuple of *deposit*. This suggests that we read each tuple of *customer* 10,000 times. Since $n_{customer} = 200$, we could make as many as 2,000,000 accesses to read *customer* tuples. As was the case for *deposit*, we can reduce the required number of accesses significantly if we store the *customer* tuples together physically. If we assume that 20 *customer* tuples fit in one block, then only 10 accesses are required to read the entire *customer* relation. Thus, only 10 accesses per tuple of *deposit* rather than 200 are required. This implies that only 100,000 block accesses are needed to process the query.

9.5.2 Block-Oriented Iteration

A major savings in block accesses results if we process the relations on a per-block basis rather than a per-tuple basis. Again, assuming that *deposit* tuples are stored together physically and that *customer* tuples are stored together physically, we can use the procedure of Figure 9.2 to compute *deposit* ⋈ *customer* . This procedure performs the join by considering an entire block of *deposit* tuples at once. We still must read the entire *deposit* relation at a cost of 500 accesses. However, instead of reading the *customer* relation once for each *tuple* of *deposit*, we read the *customer* relation once for each *block* of *deposit*. Since there are 500 blocks of *deposit* tuples and 10 blocks of *customer* tuples, reading *customer* once for every block of *deposit* tuples requires $10 \times 500 = 5000$ block accesses. Thus, the total cost in terms of block accesses is 5500 accesses (5000 accesses to *customer* blocks plus 500 accesses to *deposit* blocks). Clearly, this is a significant improvement over the number of accesses that were necessary for our initial strategy.

Our choice of *deposit* for the outer loop and *customer* for the inner loop was arbitrary. If we had used *customer* as the relation for the outer loop and *deposit* for the inner loop, the cost of our final strategy would have been slightly lower (5010 block accesses). See Exercise 9.10 for a derivation of these costs.

A major advantage to the use of the smaller relation (*customer*) in the inner loop is that it may be possible to store the entire relation in main memory temporarily. This speeds query processing significantly since it is necessary to read the inner loop relation only once. If *customer* is indeed small enough to fit in main memory, out strategy requires only 500 blocks to read *deposit* plus 10 blocks to read *customer* for a total of only 510 block accesses.

9.5.3 Merge-Join

In those cases in which neither relation fits in main memory, it is still possible to process the join efficiently if both relations happen to be stored in sorted order on the join attributes. Suppose that both *customer* and *deposit* are sorted by *customer-name*. We can then perform a *merge-join* operation. To compute a merge-join, we associate one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read. Then the corresponding tuples (if any) of the other relation are read. Since the relations are in sorted order, tuples with the same value on the join attributes are in consecutive order. This allows us to read each tuple only once. In the case in which the tuples of the relations are stored together physically, this algorithm allows us to compute the join by reading each block exactly once. For our example of *deposit* ⋈ *customer* there is a total of 510 block accesses. This is as good as the earlier join method we presented for the special case in which the entire *customer* relation fit in main memory. The algorithm of Figure 9.3 does not require the entire relation to fit in main memory. Rather, it suffices to keep all tuples with the same value for the join attributes in main memory. This is usually feasible even if both relations are large.

A disadvantage of the merge-join method is the requirement that both relations be sorted physically. However, it may be worthwhile to sort the relations in order to allow a merge-join to be performed.

9.5.4 Use of an Index

Frequently, the join attributes form a search key for an index as one of the relations being joined. In such a case, we may consider a join strategy that uses such an index. The simple strategy of Figure 9.1 is more efficient if an index exists on *customer* for *customer-name*. Given a tuple *d* in *deposit*, it is

```
pd := address of first tuple of deposit;
pc := address of first tuple of customer;
while (pc ≠ null) do
  begin
    t_c := tuple to which pc points;
    s_c := {t_c};
    set pc to point to next tuple of customer;
    done := false;
    while (not done) do
      begin
        t_c' := tuple to which pc points;
        if t_c'[customer-name] = t_c[customer-name]
          then begin
                 S_i := S_c ∪ {t_c'};
                 set pc to point to next tuple of customer;
               end
          else done := true;
      end
    t_d := tuple to which pd points;
    set pd to point to next tuple of deposit;
    while (t_d[customer-name] < t_c[customer-name]) do
      begin
        t_d := tuple to which pd points;
        set pd to point to next tuple of deposit;
      end
    while (t_d[customer-name] = t_c[customer-name]) do
      begin
        for each t in S_c do
          begin
            compute t ⋈ t_d and add this to result;
          end
        set pd to point to next tuple of deposit;
        td := tuple to which pd points;
      end
  end.
```

Figure 9.3 Merge-join.

no longer necessary to read the entire *customer* relation. Instead, the index is used to look up tuples in *customer* for which the *customer-name* value is *d*[*customer-name*].

Without use of an index, and without special assumptions about the physical storage of relations, it was shown that as many as 2 million accesses might be required. Using the index, but without making any assumptions about physical storage, the join can be computed with significantly fewer block accesses. We still need 10,000 accesses to read *deposit*. However, for each tuple of deposit only an index lookup is required. If we assume (as before) that $n_{customers} = 200$, and that 20 pointers fit in one block, then this lookup requires at most 2 index block accesses plus a block access to read the *customer* tuple itself. We access 3 blocks per tuple of *deposit* instead of 200. Adding this to the 10,000 accesses to read *deposit*, we find that the total cost of this strategy is 40,000 accesses.

Although a cost of 40,000 accesses appears high, we must remember that we achieved more efficient strategies only when we assumed that tuples were stored physically together. If this assumption does not hold for the relations being joined, then the strategy we just presented is highly desirable. Indeed the savings (160,000 accesses saved) is enough to justify creation of the index. Even if we create the index for the sole purpose of processing this one query and erase the index afterwards, we may perform fewer accesses than if we use the strategy of Figure 9.1.

9.5.5 Three-Way Join

Let us now consider a join involving three relations:

branch ⋈ *deposit* ⋈ *customer*

Assume that $n_{deposit}$ and $n_{customer}$ are as above and that $n_{branch} = 50$. Not only do we have a choice of strategy for join processing, but also we have a choice of which join to compute first. There are many possible strategies to consider. We shall analyze several of them below and leave others to the exercises.

- Strategy 1. Let us first compute the join (*deposit* ⋈ *customer*) using one of the strategies we presented above. Since *customer-name* is a key for *customer*, we know that the result of this join has at most 10,000 tuples (the number of tuples in *deposit*). If we build an index on *branch* for *branch-name*, we can compute:

  branch ⋈ (*deposit* ⋈ *customer*)

  by considering each tuple *t* of (*deposit* ⋈ *customer*) and looking up the tuple in *branch* with a *branch-name* value of *t*[*branch-name*]. Since *branch-name* is a key for *branch*, we know that we must examine only one *branch* tuple for each of the 10,000 tuples in (*deposit* ⋈ *customer*). The exact number of block accesses required by this strategy depends on the way we compute (*deposit* ⋈ *customer*) and on the way in which *branch* is stored physically. Several exercises examine the costs of various possibilities.

- **Strategy 2**. Compute the join without constructing any indices at all. This requires checking $50 \cdot 10000 \cdot 200$ possibilities, a total of 100,000,000.
- **Strategy 3**. Instead of performing two joins, we perform the pair of joins at once. The technique is first to build two indices:

  On *branch* for *branch-name*.

  On *customer* for *customer-name*.

  Next we consider each tuple t in *deposit*. For each t, we look up the corresponding tuples in *customer* and the corresponding tuples in *branch*. Thus, we examine each tuple of *deposit* exactly once.

Strategy 3 represents a form of strategy we have not considered before. It does not correspond directly to a relational-algebra operation. Instead, it combines two operations into one special-purpose operation. Using strategy 3, it is often possible to perform a join of three relations more efficiently than it is using two joins of two relations. The relative costs depend on the way in which the relations are stored, the distribution of values within columns, and the presence of indices. The exercises provide an opportunity to compute these costs in several examples.

9.6 Structure of the Query Optimizer

We have seen only some of the many query processing strategies used in database systems. Most systems implement only a few strategies and, as a result, the number of strategies to be considered by the query optimizer is limited. Other systems consider a large number of strategies. For each strategy a cost estimate is computed.

In order to simplify the strategy selection task, a query may be split into several subqueries. This not only simplifies strategy selection but also allows the query optimizer to recognize cases where a particular subquery appears several times in the same query. By performing such subqueries only once, time is saved both in the query optimizing phase and in the execution of the query itself. Recognition of common subqueries is analogous to the recognition of *common subexpressions* in many optimizing compilers for programming languages.

Clearly, examination of the query for common subqueries and the estimation of the cost of a large number of strategies impose a substantial overhead on query processing. However, the added cost of query optimization is usually more than offset by the savings at query execution

time. Therefore, most commercial systems include relatively sophisticated optimizers. The bibliographic notes give references to descriptions of query optimizers of actual database systems.

9.7 Summary

There are a large number of possible strategies for processing a query, especially if the query is complex. Strategy selection can be done using information available in main memory, with little or no disk accesses. The actual execution of the query will involve many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the central processor of the computer system, it is advantageous to spend a considerable amount of processing to save disk accesses.

Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the system to transform the query as entered by the user into an equivalent query which can be computed more efficiently. This "optimizing," or, more accurately, improving of the strategy for processing a query is called *query optimization*.

The first action the system must take on a query is to translate the query into its internal form which (for relational database systems) is usually based on the relational algebra. In the process of generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relation in the database, etc. If the query was expressed in terms of a view, the parser replaces all references to the view name with the relational algebra expression to compute the view.

Each relational algebra expression represents a particular sequence of operations. The first step in selecting a query-processing strategy is to find a relational algebra expression that is equivalent to the given expression and is efficient to execute. There are a number of different rules for transforming relational algebra queries, including:

- Perform selection operations as early as possible.
- Perform projections early.

The strategy we choose for a query depends upon the size of each relation and the distribution of values within columns. In order to be able to choose a strategy based on reliable information, database systems may store statistics for each relation r. These statistics include:

- The number of tuples in the relation r.
- The size of a record (tuple) of relation r in bytes (for fixed-length records).

- The number of distinct values that appear in the relation r for a particular attribute.

The first two statistics allow us to estimate accurately the size of a cartesian product. The third statistic allows us to estimate how many tuples satisfy a simple selection predicate.

Statistical information about relations is particularly useful when several indices are available to assist in the processing of a query. The presence of these structures has a significant influence on the choice of a query-processing strategy.

Queries involving a natural join may be processed in several ways, depending on the availability of indices and the form of physical storage used for the relations. If tuples of a relation are stored together physically, a *block-oriented* join strategy may be advantageous. If the relations are sorted, a *merge-join* may be desirable. It may be more efficient to sort a relation prior to join computation (so as to allow use of the merge-join strategy). It may also be advantageous to compute a temporary index for the sole purpose of allowing a more efficient join strategy to be used.

Exercises

9.1 At what point during query processing does optimization occur?

9.2 Why is it not desirable to force users to make an explicit choice of a query processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query processing strategies?

9.3 Consider the following SQL query for our bank database:

```
select customer-name
from deposit S
where (select branch-name
        from deposit T
        where S.customer-name = T.customer-name)
       contains
       (select branch-name
        from branch
        where branch-city = "Brooklyn")
```

Write an efficient relational algebra expression that is equivalent to this query. Justify your choice.

9.4 Consider the following SQL query for our bank database:

```
select T.branch-name
from branch T, branch S
where T.assets > S.assets and
        S.branch-city = "Brooklyn"
```

Write an efficient relational algebra expression that is equivalent to this query. Justify your choice.

9.5 Show that the following equivalences hold, and explain how they can be applied to improve the efficiency of certain queries:

a. $\sigma_P(r_1 \cup r_2) = \sigma_P(r_1) \cup \sigma_P(r_2)$

b. $\sigma_P(r_1 - r_2) = \sigma_P(r_1) - r_2 = \sigma_P(r_1) - \sigma_P(r_2)$

c. $(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$

d. $r_1 \cup r_2 = r_2 \cup r_1$

9.6 Consider the relations $r_1(A,B,C)$, $r_2(C,D,E)$, and $r_3(E,F)$, with primary keys A, C, and E respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

9.7 Consider the relations $r_1(A,B,C)$, $r_2(C,D,E)$, and $r_3(E,F)$ of Exercise 9.6 again, but now assume there are no primary keys except the entire scheme. Let $V(C,r_1)$ be 900, $V(C,r_2)$ be 1100, $V(E,r_2)$ be 50, and $V(E,r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

9.8 Clustering indices may allow faster access to data than a nonclustering index. When must we create a nonclustering index despite the advantages of a clustering index?

9.9 What are the advantages and disadvantages of hash functions relative to B^+-tree indices? How might the type of index available influence the choice of a query processing strategy?

9.10 Recompute the cost of the strategy of Section 9.5.2 using *deposit* as the relation of the inner loop and *customer* as the relation of the output loop (thereby reversing the roles they played in the example of Section 9.5.2).

9.11 Explain the difference between a clustering index and a nonclustering index.

9.12 Let relations $r_1(A,B,C)$ and $r_2(C,D,E)$ have the following properties:

- r_1 has 20,000 tuples.
- r_2 has 45,000 tuples.
- 25 tuples of r_1 fit on one block.
- 30 tuples of r_2 fit on one block.

Estimate the number of block accesses required using each of the following join strategies for $r_1 \bowtie r_2$:

- Simple iteration.
- Block-oriented iteration.
- Merge-join

9.13 Consider relations r_1 and r_2 of Exercise 9.12 along with a relation $r_3(E,F)$. Assume that r_3 has 30,000 tuples and that 40 tuples of r_3 fit on one block. Estimate the costs of the 3 strategies of Section 9.5.5 for computing $r_1 \bowtie r_2 \bowtie r_3$.

Bibliographic Notes

Some the ideas used in query optimization are derived from solutions to similar problems in code optimization as performed by compilers of standard programming languages. There are several texts that present optimization from a programming languages point of view, including [Aho et al. 1986], and [Tremblay and Sorenson 1985]. Selinger et al. [1979] describes access path selection in System R. Kim [1981, 1982] describe join strategies and the optimal use of available main memory. These papers discuss many of the strategies that we presented in this chapter. Wong and Youssefi [1976] introduce a technique called *decomposition*, which is used in the Ingres database system. The Ingres decomposition strategy motivated the third strategy we presented for three-way joins. In Ingres, an extension of this technique is used to choose a strategy for general queries. Ingres and System R are discussed in more detail in Chapter 15.

If an entire group of queries is considered, it is possible to discover *common subexpressions* that can be evaluated once for the entire group. Finkelstein [1982], and Hall [1976] consider optimization of a group of

queries and the use of common subexpressions. When queries are generated through views, it is often the case that more relations are joined than is necessary to compute the query. A collection of techniques for join minimization have been grouped under the name *tableau optimization*. The notion of a tableau was introduced by Aho et al. [1979a, 1979c]. Ullman [1982a] and Maier [1983] provide a textbook coverage of tableaux.

Theoretical results on the complexity of the computation of relational algebra operations appear in [Gotlieb 1975], [Pecherer 1975], and [Blasgen and Eswaren 1976]. A survey of query processing techniques appears in [Jarke and Koch 1984].

An actual query processor must translate statements in the query language into an internal form suitable for the analysis we have discussed in this chapter. Parsing query languages differs little from parsing of traditional programming languages. Most compiler texts (including [Aho et al. 1986], and [Tremblay and Sorenson 1985]) cover the main parsing techniques. A more theoretical presentation of parsing and language translation is given by Aho and Ullman [1972, 1973].

Query processing for distributed database systems use some concepts from this chapter. Techniques specific to distributed systems appear in Chapter 12 and the bibliographic notes to that chapter.