# Chapter 3: Relational Model

## 3.1 Schema Diagram

A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by **schema diagram**s. Figure 3.9 shows the schema diagram for our banking enterprise. Each relation appears as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

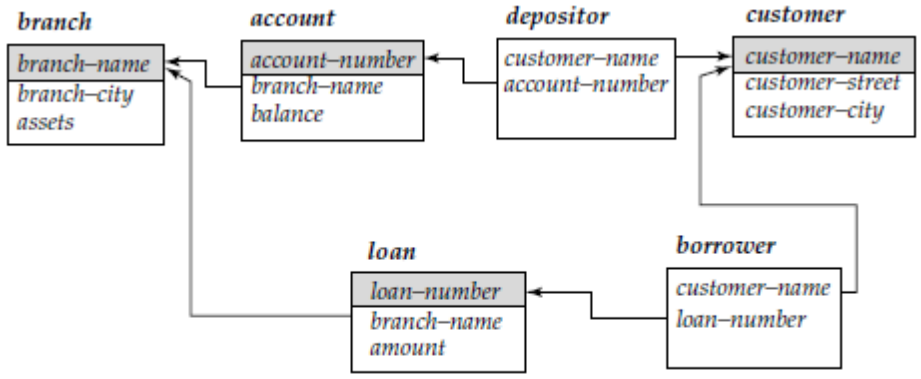


**Figure 3.9** Schema diagram for the banking enterprise.

## Keys in Relational Model:

Student:

| Stud_id | Name | Phone_no | Address |
|---|---|---|---|
| 1 | Ram | 9899967000 | Ktm |
| 2 | Ram | 9796582000 | Ktm |
| 3 | suresh | 9432516782 | pokhara |

Student_course:

| Stud_no | Course_no | Course_name |
|---|---|---|
| 1 | C1 | DBMS |
| 2 | C2 | Computer Network |
| 1 | C1 | Computer Network |

**Super Key:** The set of one or more attributes which can uniquely identify a tuple is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME) etc.

- Adding zero or more attributes to candidate key generates super key.
- A candidate key is a super key but vice versa is not true

**Candidate Key:** The minimal set of attribute which can uniquely identify a tuple is known as candidate key. OR A super key with no redundant attribute is known as candidate key. For Example, STUD_NO in STUDENT relation.

- The value of Candidate Key is unique and non-null for every tuple.
- There can be more than one candidate key in a relation. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT.
- The candidate key can be simple (having only one attribute) or composite as well. For Example, {STUD_NO, COURSE_NO} is a composite candidate key for relation STUDENT_COURSE.

**Primary Key:** There can be more than one candidate key in a relation out of which one can be chosen as primary key. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT but STUD_NO can be chosen as primary key (only one out of many candidate keys).
**Foreign Key:** If an attribute can only take the values which are present as values of some other attribute, it will be foreign key to the attribute to which it refers. The relation which is being referenced is called referenced relation and corresponding attribute is called referenced attribute and the relation which refers to referenced relation is called referencing relation and corresponding attribute is called referencing attribute. Referenced attribute of referencing attribute should be primary key. For Example, STUD_NO in STUDENT_COURSE is a foreign key to STUD_NO in STUDENT relation.

## 3.2 The Relational Algebra

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project*, *union*, *set difference*, *Cartesian product,* and *rename*. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment.

### 3.2.1 Fundamental Operations

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

#### 3.2.1.1 The Select Operation

The select operation selects tuples that satisfy a given predicate.We use the lowercase Greek letter sigma ($\sigma$) to denote selection. The predicate appears as a subscript to $\sigma$. The argument relation is in parentheses after the $\sigma$. Thus, to select those tuples of the *loan* relation where the branch is "Perryridge," we write

σbranch-name ="Perryridge" (loan)

If the *loan* relation is as shown in Figure 3.6, then the relation that results from the preceding query is as shown in Figure 3.10.

We can find all tuples in which the amount lent is more than $1200 by writing

σamount>1200 (loan)

In general, we allow comparisons using =, _=, <, ≤, >, ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ( $\neg$). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write

*σbranch-name ="Perryridge"∧ amount>1200 (loan)*

| loan-number | branch-name | amount |
|---|---|---|
| L-15 | Perryridge | 1500 |
| L-16 | Perryridge | 1300 |

**Figure 3.10** Result of $\sigma_{branch\text{-}name}$ ="Perryridge" (*loan*).

### 3.2.1.2 The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi ($\Pi$).We list those attributes that we wish to appear in the result as a subscript to $\Pi$. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as

$\Pi_{loan\text{-}number,\ amount}(loan)$

### 3.2.1.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find those customers who live in Harrison." We write:

$\Pi_{customer\text{-}name}(\sigma_{customer\text{-}city = \text{"Harrison"}}(customer))$

### 3.2.1.4 The Union Operation

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the *customer* relation does not contain the information, since a customer does not need to have either an account or a loan at the bank. To answer this query, we need the information in the *depositor* relation (Figure 3.5) and in the *borrower* relation (Figure 3.7).We know how to find the names of all customers with a loan in the bank:

$\Pi_{customer\text{-}name}(borrower)$

We also know how to find the names of all customers with an account in the bank:

$\Pi_{customer\text{-}name}(depositor)$

To answer the query, we need the **union** of these two sets; that is, we need all customer names that appear in either or both of the two relations.We find these data by the binary operation union, denoted, as in set theory, by $\cup$. So the expression needed is

$$\Pi_{customer\text{-}name}(borrower) \cup \Pi_{customer\text{-}name}(depositor)$$

The result relation for this query appears in Figure 3.12. Notice that there are 10 tuples in the result, even though there are seven distinct borrowers and six depositors. This apparent discrepancy occurs because Smith, Jones, and Hayes are borrowers as well as depositors. Since relations are sets, duplicate values are eliminated.

| *customer-name* |
|---|
| Adams |
| Curry |
| Hayes |
| Jackson |
| Jones |
| Smith |
| Williams |
| Lindsay |
| Johnson |
| Turner |

**Figure 3.12** Names of all customers who have either a loan or an account.

Therefore, for a union operation $r \cup s$ to be valid, 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.
**2.** The domains of the *i*th attribute of *r* and the *i*th attribute of *s* must be the same, for all $i$.
Note that *r* and *s* can be, in general, temporary relations that are the result of relational-algebra expressions.

### 3.2.1.5 The Set Difference Operation

The set-difference operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another. The expression r $-$ s produces a relation containing those tuples in r but not in s.
We can find all customers of the bank who have an account but not a loan by writing

$$\Pi_{customer\text{-}name}(depositor) - \Pi_{customer\text{-}name}(borrower)$$

The result relation for this query appears in Figure 3.13.

As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set difference operation $r - s$ to be valid, we require that the relations *r* and *s* be of the same arity, and that the domains of the *i*th attribute of *r* and the *i*th attribute of *s* be the same.

## 3.2.1.6 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations.We write the Cartesian product of relations $r_1$ and $r_2$ as $r_1 \times r_2$.

For example, the relation schema for $r = \textbf{borrower} \times \textbf{loan}$ is
(*borrower.customer-name*, *borrower.loan-number*, *loan.loan-number*, *loan.branch-name*, *loan.amount*)
With this schema, we can distinguish *borrower.loan-number* from *loan.loan-number*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for *r* as
(*customer-name*, *borrower.loan-number*, *loan.loan-number*, *branch-name*, *amount*)

Now that we know the relation schema for $r = borrower \times loan$, what tuples appear in *r*? As you may suspect, we construct a tuple of *r* out of each possible pair of tuples: one from the *borrower* relation and one from the *loan* relation. Thus, *r* is a large relation, as you can see from Figure 3.14, which includes only a portion of the tuples that make up *r*.

Assume that we have $n_1$ tuples in *borrower* and $n_2$ tuples in *loan*. 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 for some tuples *t* in *r*, it may be that ***t*[*borrower.loan-number*] _= *t*[*loan.loan-number*].**

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch.We need the information in both the *loan* relation and the *borrower* relation to do so. If we write

$$\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)$$

then the result is the relation in Figure 3.15. We have a relation that pertains to only the Perryridge branch. However, the *customer-name* column may contain customers who do not have a loan at the Perryridge branch. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *borrower* with one tuple of *loan*.)

Since the Cartesian-product operation associates *every* tuple of *loan* with every tuple of *borrower*, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in *borrower*×*loan* that contains his name, and *borrower.loan-number*= *loan.loan-number*. So, if we write

$$\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan))$$

we get only those tuples of *borrower* × *loan* that pertain to customers who have a loan at the Perryridge branch.

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

$$\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)))$$

| *customer-name* | *loan-number* |
|---|---|
| Adams | L-16 |
| Curry | L-93 |
| Hayes | L-15 |
| Jackson | L-14 |
| Jones | L-17 |
| Smith | L-11 |
| Smith | L-23 |
| Williams | L-17 |

**Figure 3.7** The *borrower* relation.

| *loan-number* | *branch-name* | *amount* |
|---|---|---|
| L-11 | Round Hill | 900 |
| L-14 | Downtown | 1500 |
| L-15 | Perryridge | 1500 |
| L-16 | Perryridge | 1300 |
| L-17 | Downtown | 1000 |
| L-23 | Redwood | 2000 |
| L-93 | Mianus | 500 |

**Figure 3.6** The *loan* relation.

| *customer-name* | *account-number* |
|---|---|
| Hayes | A-102 |
| Johnson | A-101 |
| Johnson | A-201 |
| Jones | A-217 |
| Lindsay | A-222 |
| Smith | A-215 |
| Turner | A-305 |

**Figure 3.5** The *depositor* relation.

| *customer-name* |
|---|
| Johnson |
| Lindsay |
| Turner |

**Figure 3.13** Customers with an account but no loan.

The result of this expression, shown in Figure 3.16, is the correct answer to our query

| customer-name | borrower. loan-number | loan. loan-number | branch-name | amount |
|---|---|---|---|---|
| Adams | L-16 | L-11 | Round Hill | 900 |
| Adams | L-16 | L-14 | Downtown | 1500 |
| Adams | L-16 | L-15 | Perryridge | 1500 |
| Adams | L-16 | L-16 | Perryridge | 1300 |
| Adams | L-16 | L-17 | Downtown | 1000 |
| Adams | L-16 | L-23 | Redwood | 2000 |
| Adams | L-16 | L-93 | Mianus | 500 |
| Curry | L-93 | L-11 | Round Hill | 900 |
| Curry | L-93 | L-14 | Downtown | 1500 |
| Curry | L-93 | L-15 | Perryridge | 1500 |
| Curry | L-93 | L-16 | Perryridge | 1300 |
| Curry | L-93 | L-17 | Downtown | 1000 |
| Curry | L-93 | L-23 | Redwood | 2000 |
| Curry | L-93 | L-93 | Mianus | 500 |
| Hayes | L-15 | L-11 | | 900 |
| Hayes | L-15 | L-14 | | 1500 |
| Hayes | L-15 | L-15 | | 1500 |
| Hayes | L-15 | L-16 | | 1300 |
| Hayes | L-15 | L-17 | | 1000 |
| Hayes | L-15 | L-23 | | 2000 |
| Hayes | L-15 | L-93 | | 500 |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |
| Smith | L-23 | L-11 | Round Hill | 900 |
| Smith | L-23 | L-14 | Downtown | 1500 |
| Smith | L-23 | L-15 | Perryridge | 1500 |
| Smith | L-23 | L-16 | Perryridge | 1300 |
| Smith | L-23 | L-17 | Downtown | 1000 |
| Smith | L-23 | L-23 | Redwood | 2000 |
| Smith | L-23 | L-93 | Mianus | 500 |
| Williams | L-17 | L-11 | Round Hill | 900 |
| Williams | L-17 | L-14 | Downtown | 1500 |
| Williams | L-17 | L-15 | Perryridge | 1500 |
| Williams | L-17 | L-16 | Perryridge | 1300 |
| Williams | L-17 | L-17 | Downtown | 1000 |
| Williams | L-17 | L-23 | Redwood | 2000 |
| Williams | L-17 | L-93 | Mianus | 500 |

**Figure 3.14** Result of *borrower* × *loan*.

| customer-name | borrower. loan-number | loan. loan-number | branch-name | amount |
|---|---|---|---|---|
| Adams | L-16 | L-15 | Perryridge | 1500 |
| Adams | L-16 | L-16 | Perryridge | 1300 |
| Curry | L-93 | L-15 | Perryridge | 1500 |
| Curry | L-93 | L-16 | Perryridge | 1300 |
| Hayes | L-15 | L-15 | Perryridge | 1500 |
| Hayes | L-15 | L-16 | Perryridge | 1300 |
| Jackson | L-14 | L-15 | Perryridge | 1500 |
| Jackson | L-14 | L-16 | Perryridge | 1300 |
| Jones | L-17 | L-15 | Perryridge | 1500 |
| Jones | L-17 | L-16 | Perryridge | 1300 |
| Smith | L-11 | L-15 | Perryridge | 1500 |
| Smith | L-11 | L-16 | Perryridge | 1300 |
| Smith | L-23 | L-15 | Perryridge | 1500 |
| Smith | L-23 | L-16 | Perryridge | 1300 |
| Williams | L-17 | L-15 | Perryridge | 1500 |
| Williams | L-17 | L-16 | Perryridge | 1300 |

**Figure 3.15** Result of $\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)$.

| customer-name |
|---|
| Adams |
| Hayes |

**Figure 3.16** Result of $\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)))$

## 3.2.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho ($\rho$), lets us do this. Given a relational-algebra expression $E$, the expression

$$\rho x\,(E)$$

returns the result of expression *E* under the name *x*.

## 3.2.3 Additional Operations

## 3.2.3.1 The Set-Intersection Operation

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

$\Pi$*customer-name* (*borrower* ) $\cap$ $\Pi$*customer-name* (*depositor*)

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

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

### 3.2.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount." We first form the Cartesian product of the *borrower* and *loan* relations. Then, we select those tuples that pertain to only the same *loan-number*, followed by the projection of the resulting *customer-name*, *loan-number*, and *amount*:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan))$$

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol . The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

As an illustration, consider again the example "Find the names of all customers who have a loan at the bank, and find the amount of the loan."We express this query by using the natural join as follows:

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

Since the schemas for *borrower* and *loan* (that is, *Borrower-schema* and *Loan-schema*) have the attribute *loan-number* in common, the natural-join operation considers only pairs of tuples that have the same value on *loan-number*. It combines each such pair of tuples into a single tuple on the union of the two schemas (that is, *customer-name,branch-name, loan-number, amount*). After performing the projection, we obtain the relation in Figure 3.21.

| *customer-name* | *loan-number* | *amount* |
|---|---|---|
| Adams | L-16 | 1300 |
| Curry | L-93 | 500 |
| Hayes | L-15 | 1500 |
| Jackson | L-14 | 1500 |
| Jones | L-17 | 1000 |
| Smith | L-23 | 2000 |
| Smith | L-11 | 900 |
| Williams | L-17 | 1000 |

**Figure 3.21** Result of $\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The **natural join** of *r* and *s*, denoted by $r\ s$, is a relation on schema $R \cup S$ formally defined as follows:

$$r \bowtie\ s\ =\ \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ r\ \times\ s)$$

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

• Find the names of all branches with customers who have an account in the bank and who live in Harrison.

$$\Pi_{branch\text{-}name}\ (\sigma_{customer\text{-}city = \text{"Harrison"}}\ (customer \bowtie\ account \bowtie\ depositor))$$

Find all customers who have *both* a loan and an account at the bank.

$$\Pi_{customer\text{-}name}\ (borrower \bowtie depositor)$$

### 3.2.3.3 The Division Operation

The **division** operation, denoted by ÷, is suited to queries that include the phrase "for all." Suppose that we wish to find all customers who have an account at *all* the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression

$r1 = \Pi branch\text{-}name\ (\sigma branch\text{-}city$ ="Brooklyn" (*branch*))

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

$$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$

Figure 3.24 shows the result relation for this expression.Now, we need to find customers who appear in $r2$ with *every* branch name in $r1$. The operation that provides exactly those customers is the divide operation. We formulate the query by writing

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

A tuple *t* is in $r \div s$ if and only if both of two conditions hold:

**1.** *t* is in $\Pi_{R-S}(r)$

**2.** For every tuple $t_s$ in *s*, there is a tuple $t_r$ in *r* satisfying both of the following:

**a.** $t_r[S] = t_s[S]$

**b.** $t_r[R - S] = t$

## 3.3.2 Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result. To illustrate the concept of aggregation, we shall use the *pt-works* relation in Figure 3.27, for part-time employees. Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query is:

***G*sum(*salary*)(*pt-works*)**

The symbol G is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation G signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "distinct" appended to the end of the function name (for example, count-distinct). An example arises in the query "Find the number of branches appearing in the pt-works relation."

***G*count-distinct(*branch-name*)(*pt-works*)**

Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation *pt-works* into groups based on the branch, and to apply the aggregate function on each group.

***branch-name G*sum(*salary*)(*pt-works*)**

Going back to our earlier example, if we want to find the maximum salary for part-time employees at each branch, in addition to the sum of the salaries, we write the expression

***branch-name G*sum(*salary*),max(*salary*)(*pt-works*)**

# 3.4 Modification of the Database

## 3.4.1 Deletion

We express a delete request in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database. We can delete only whole tuples; we cannot delete values on only particular attributes.
In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

where $r$ is a relation and $E$ is a relational-algebra query. Here are several examples of relational-algebra delete requests:

- Delete all of Smith's account records.

$$depositor \leftarrow depositor - \sigma customer\text{-}name = \text{“Smith”}\ (depositor)$$

- Delete all loans with amount in the range 0 to 50.

$$loan \leftarrow loan - \sigma amount \geq 0 \text{ and } amount \leq 50\ (loan)$$

- Delete all accounts at branches located in Needham.

$$r1 \leftarrow \sigma branch\text{-}city = \text{“Needham”}\ (account \square branch)$$

$$r2 \leftarrow \Pi branch\text{-}name, account\text{-}number, balance\ (r1)$$

$$account \leftarrow account - r2$$

Note that, in the final example, we simplified our expression by using assignment to temporary relations ($r_1$ and $r_2$).

## 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 relational algebra expresses an insertion by

$$r \leftarrow r \cup E$$

where $r$ is a relation and $E$ is a relational-algebra expression.
Suppose that we wish to insert the fact that Smith has $1200 in account A-973 at the Perryridge branch.We write

$$account \leftarrow account \cup \{(\text{A-973, “Perryridge”, 1200})\}$$

$$depositor \leftarrow depositor \cup \{(\text{“Smith”, A-973})\}$$

## 3.4.3 Updating

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple.We can use the generalized-projection operator to do this task:

$$r \leftarrow \Pi F1, F2, \ldots, Fn(r)$$

where each $F_i$ is either the $i$th attribute of $r$, if the $i$th attribute is not updated, or, if the attribute is to be updated, $F_i$ is an expression, involving only constants and the attributes of $r$, that gives the new value for the attribute.If we want to select some tuples from $r$ and to update only them, we can use the following expression; here, $P$ denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi F1, F2, \ldots, Fn(\sigma P\ (r)) \cup (r - \sigma P\ (r))$$

To illustrate the use of the update operation, suppose that interest payments are being made, and that all balances are to be increased by 5 percent.We write

$$account \leftarrow \Pi account\text{-}number, branch\text{-}name, balance * 1.05\ (account)$$

Now suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent.We write

$$account \leftarrow \Pi_{AN, BN, balance * 1.06}\ (\sigma balance > 10000\ (account)) \cup \Pi_{AN, BN\ balance * 1.05}\ (\sigma balance \leq 10000\ (account))$$

where the abbreviations $AN$ and $BN$ stand for *account-number* and *branch-name*, respectively