CHAPTER

5 Relational Algebra and Relational Calculus

Chapter Objectives

In this chapter you will learn:

- The meaning of the term "relational completeness."
- How to form queries in the relational algebra.
- How to form queries in the tuple relational calculus.
- How to form queries in the domain relational calculus.
- The categories of relational Data Manipulation Languages (DMLs).

In the previous chapter we introduced the main structural components of the relational model. As we discussed in Section 2.3, another important part of a data model is a manipulation mechanism, or *query language*, to allow the underlying data to be retrieved and updated. In this chapter we examine the query languages associated with the relational model. In particular, we concentrate on the relational algebra and the relational calculus as defined by Codd (1971) as the basis for relational languages. Informally, we may describe the relational algebra as a (high-level) procedural language: it can be used to tell the DBMS how to build a new relation from one or more relations in the database. Again, informally, we may describe the relational calculus as a nonprocedural language: it can be used to formulate the definition of a relation in terms of one or more database relations. However, the relational algebra and relational calculus are formally equivalent to one another: for every expression in the algebra, there is an equivalent expression in the calculus (and vice versa).

Both the algebra and the calculus are formal, non-user-friendly languages. They have been used as the basis for other, higher-level Data Manipulation Languages (DMLs) for relational databases. They are of interest because they illustrate the basic operations required of any DML and because they serve as the standard of comparison for other relational languages.

The relational calculus is used to measure the selective power of relational languages. A language that can be used to produce any relation that can be derived using the relational calculus is said to be **relationally complete**. Most relational query languages are relationally complete but have more expressive

power than the relational algebra or relational calculus, because of additional operations such as calculated, summary, and ordering functions.

Structure of this Chapter In Section 5.1 we examine the relational algebra and in Section 5.2 we examine two forms of the relational calculus: tuple relational calculus and domain relational calculus. In Section 5.3 we briefly discuss some other relational languages. We use the *DreamHome* rental database instance shown in Figure 4.3 to illustrate the operations.

In Chapters 6–9 we examine SQL, the formal and *de facto* standard language for RDBMSs, which has constructs based on the tuple relational calculus. In Appendix M we examine QBE (Query-By-Example), another highly popular visual query language for RDBMSs, which is based in part on the domain relational calculus.

5.1 The Relational Algebra

The relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation(s). Thus both the operands and the results are relations, and so the output from one operation can become the input to another operation. This ability allows expressions to be nested in the relational algebra, just as we can nest arithmetic operations. This property is called **closure**: relations are closed under the algebra, just as numbers are closed under arithmetic operations.

The relational algebra is a relation-at-a-time (or set) language in which all tuples, possibly from several relations, are manipulated in one statement without looping. There are several variations of syntax for relational algebra commands and we use a common symbolic notation for the commands and present it informally. The interested reader is referred to Ullman (1988) for a more formal treatment.

There are many variations of the operations that are included in relational algebra. Codd (1972a) originally proposed eight operations, but several others have been developed. The five fundamental operations in relational algebra—Selection, Projection, Cartesian product, Union, and Set difference—perform most of the data retrieval operations that we are interested in. In addition, there are also the Join, Intersection, and Division operations, which can be expressed in terms of the five basic operations. The function of each operation is illustrated in Figure 5.1.

The Selection and Projection operations are **unary** operations, as they operate on one relation. The other operations work on pairs of relations and are therefore called **binary** operations. In the following definitions, let R and S be two relations defined over the attributes $A = (a_1, a_2, \ldots, a_N)$ and $B = (b_1, b_2, \ldots, b_M)$, respectively.

5.1.1 Unary Operations

We start the discussion of the relational algebra by examining the two unary operations: Selection and Projection.

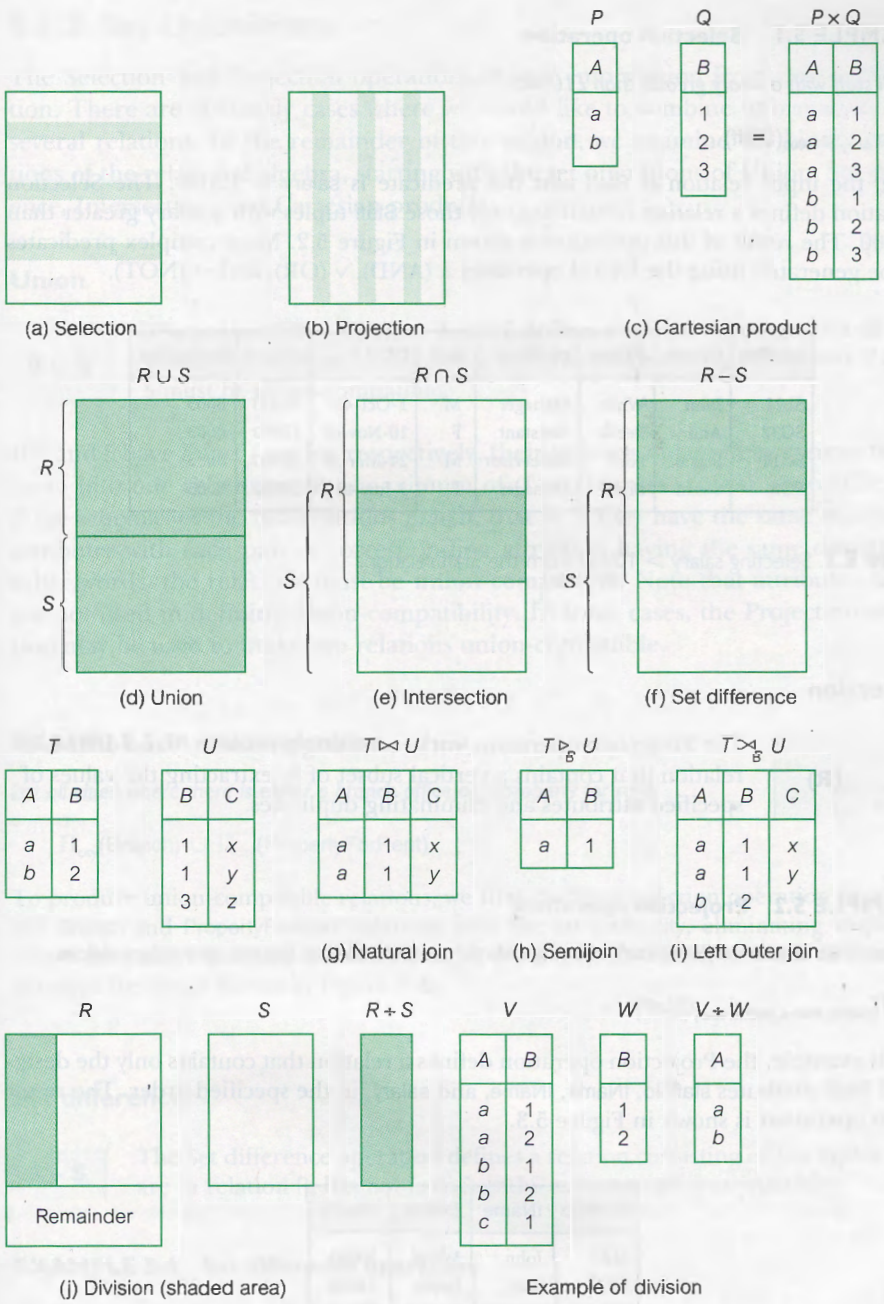

Figure 5.1 Illustration showing the function of the relational algebra operations.

Selection (or Restriction)

$\sigma_{predicate}(R)$

The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition *(predicate)*.

EXAMPLE 5.1 Selection operation

List all staff with a salary greater than £10000.

$\sigma_{salary > 10000}(Staff)$

Here, the input relation is Staff and the predicate is salary > 10000. The Selection operation defines a relation containing only those Staff tuples with a salary greater than £10000. The result of this operation is shown in Figure 5.2. More complex predicates can be generated using the logical operators ∧ (AND), ∨ (OR), and ~ (NOT).

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Figure 5.2 Selecting salary > 10000 from the Staff relation.

Projection

$\Pi_{a_1, \ldots, a_n}(R)$

The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

EXAMPLE 5.2 Projection operation

Produce a list of salaries for all staff, showing only the staffNo, fName, lName, *and* salary *details.*

$\Pi_{staffNo, fName, lName, salary}(Staff)$

In this example, the Projection operation defines a relation that contains only the designated Staff attributes staffNo, fName, lName, and salary, in the specified order. The result of this operation is shown in Figure 5.3.

staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

Figure 5.3 Projecting the Staff relation over the staffNo, fName, lName, and salary attributes.

5.1.2 Set Operations

The Selection and Projection operations extract information from only one relation. There are obviously cases where we would like to combine information from several relations. In the remainder of this section, we examine the binary operations of the relational algebra, starting with the set operations of Union, Set difference, Intersection, and Cartesian product.

Union

R ∪ S The union of two relations R and S defines a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.

If R and S have I and J tuples, respectively, their union is obtained by concatenating them into one relation with a maximum of $(I + J)$ tuples. Union is possible only if the schemas of the two relations match, that is, if they have the same number of attributes with each pair of corresponding attributes having the same domain. In other words, the relations must be **union-compatible**. Note that attributes names are not used in defining union-compatibility. In some cases, the Projection operation may be used to make two relations union-compatible.

EXAMPLE 5.3 Union operation

List all cities where there is either a branch office or a property for rent.

$$\Pi_{city}(\text{Branch}) \cup \Pi_{city}(\text{PropertyForRent})$$

To produce union-compatible relations, we first use the Projection operation to project the Branch and PropertyForRent relations over the attribute city, eliminating duplicates where necessary. We then use the Union operation to combine these new relations to produce the result shown in Figure 5.4.

city
London
Aberdeen
Glasgow
Bristol

Figure 5.4 Union based on the city attribute from the Branch and PropertyForRent relations

Set difference

R – S The Set difference operation defines a relation consisting of the tuples that are in relation R, but not in S. R and S must be union-compatible.

EXAMPLE 5.4 Set difference operation

List all cities where there is a branch office but no properties for rent.

$$\Pi_{city}(\text{Branch}) - \Pi_{city}(\text{PropertyForRent})$$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city. We then use the Set difference operation to combine these new relations to produce the result shown in Figure 5.5.

city
Bristol

Figure 5.5 Set difference based on the city attribute from the Branch and PropertyForRent relations

Intersection

R ∩ S The Intersection operation defines a relation consisting of the set of all tuples that are in both R and S. R and S must be union-compatible.

city
Aberdeen
London
Glasgow

Figure 5.6 Intersection based on city attribute from the Branch and PropertyForRent relations.

EXAMPLE 5.5 Intersection operation

List all cities where there is both a branch office and at least one property for rent.

$$\Pi_{city}(\text{Branch}) \cap \Pi_{city}(\text{PropertyForRent})$$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city. We then use the Intersection operation to combine these new relations to produce the result shown in Figure 5.6.

Note that we can express the Intersection operation in terms of the Set difference operation:

$$R \cap S = R - (R - S)$$

Cartesian product

R × S The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

The Cartesian product operation multiplies two relations to define another relation consisting of all possible pairs of tuples from the two relations. Therefore, if one relation has I tuples and N attributes and the other has J tuples and M attributes, the Cartesian product relation will contain $(I * J)$ tuples with $(N + M)$ attributes. It is possible that the two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names within a relation.

EXAMPLE 5.6 Cartesian product operation

List the names and comments of all clients who have viewed a property for rent.

The names of clients are held in the Client relation and the details of viewings are held in the Viewing relation. To obtain the list of clients and the comments on properties they have viewed, we need to combine these two relations:

$$\Pi_{clientNo,\ fName,\ lName}(\text{Client})) \times (\Pi_{clientNo,\ propertyNo,\ comment}(\text{Viewing}))$$

The result of this operation is shown in Figure 5.7. In its present form, this relation contains more information than we require. For example, the first tuple of this relation contains different clientNo values. To obtain the required list, we need to carry out a Selection operation on this relation to extract those tuples where Client.clientNo = Viewing.clientNo. The complete operation is thus:

$$\sigma_{Client.clientNo = Viewing.clientNo}((\Pi_{clientNo,\ fName,\ lName}(\text{Client})) \times (\Pi_{clientNo,\ propertyNo,\ comment}(\text{Viewing}))$$

The result of this operation is shown in Figure 5.8.

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR56	PG4	
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	
CR74	Mike	Ritchie	CR56	PA14	too small
CR74	Mike	Ritchie	CR76	PG4	too remote
CR74	Mike	Ritchie	CR56	PG4	
CR74	Mike	Ritchie	CR62	PA14	no dining room
CR74	Mike	Ritchie	CR56	PG36	
CR62	Mary	Tregear	CR56	PA14	too small
CR62	Mary	Tregear	CR76	PG4	too remote
CR62	Mary	Tregear	CR56	PG4	
CR62	Mary	Tregear	CR62	PA14	no dining room
CR62	Mary	Tregear	CR56	PG36	

Figure 5.7 Cartesian product of reduced Client and Viewing relations.

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR56	PG36	
CR62	Mary	Tregear	CR62	PA14	no dining room

Figure 5.8 Restricted Cartesian product of reduced Client and Viewing relations.

Decomposing complex operations

The relational algebra operations can be of arbitrary complexity. We can decompose such operations into a series of smaller relational algebra operations and give a name to the results of intermediate expressions. We use the assignment operation, denoted by ←, to name the results of a relational algebra operation. This works in a similar manner to the assignment operation in a programming language: in this case, the right-hand side of the operation is assigned to the left-hand side. For instance, in the previous example we could rewrite the operation as follows:

$$\text{TempViewing(clientNo, propertyNo, comment)} \leftarrow \Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing})$$

$$\text{TempClient(clientNo, fName, lName)} \leftarrow P_{\text{clientNo, fName, lName}}(\text{Client})$$

$$\text{Comment(clientNo, fName, lName, vclientNo, propertyNo, comment)} \leftarrow \text{TempClient} \times \text{TempViewing}$$

$$\text{Result} \leftarrow \sigma_{\text{clientNo} = \text{vclientNo}}(\text{Comment})$$

Alternatively, we can use the Rename operation ρ (rho), which gives a name to the result of a relational algebra operation. Rename allows an optional name for each of the attributes of the new relation to be specified.

$\rho_S(E)$ or $\rho_{S(a_1, a_2, \ldots, a_n)}(E)$	The Rename operation provides a new name S for the expression E, and optionally names the attributes as $a_1, a_2, \ldots, a_n$.

5.1.3 Join Operations

Typically, we want only combinations of the Cartesian product that satisfy certain conditions and so we would normally use a **Join operation** instead of the Cartesian product operation. The Join operation, which combines two relations to form a new relation, is one of the essential operations in the relational algebra. Join is a derivative of Cartesian product, equivalent to performing a Selection operation, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Join is one of the most difficult operations to implement efficiently in an RDBMS and is one of the reasons why relational systems have intrinsic performance problems. We examine strategies for implementing the Join operation in Section 23.4.3.

There are various forms of the Join operation, each with subtle differences, some more useful than others:

- Theta join
- Equijoin (a particular type of Theta join)
- Natural join
- Outer join
- Semijoin

Theta join (θ-join)

$R \bowtie_F S$	The Theta join operation defines a relation that contains tuples satisfying the predicate F from the Cartesian product of R and S. The predicate F is of the form $R.a_i\ \theta\ S.b_j$, where θ may be one of the comparison operators ($<, \leq, >, \geq, =, \neq$).

We can rewrite the Theta join in terms of the basic Selection and Cartesian product operations:

$$R \bowtie_F S = \sigma_F(R \times S)$$

As with a Cartesian product, the degree of a Theta join is the sum of the degrees of the operand relations R and S. In the case where the predicate F contains only equality (=), the term **Equijoin** is used instead. Consider again the query of Example 5.6.

EXAMPLE 5.7

List the names and

In Example 5.6
list. However, th

$(\Pi_{clientNo, fName,}$

or

Result ← Ter

The result of the

Natural join

$R \bowtie S$	Th co eli

The Natural joi
relations that ha
degrees of the r

EXAMPLE 5.8

List the names and

In Example 5.7 v
two occurrences
occurrence of the

$(\Pi_{clientNo, fName,}$

or

Result ← Ter

The result of this

Figure 5.9 Natur

EXAMPLE 5.7 Equijoin operation

List the names and comments of all clients who have viewed a property for rent.

In Example 5.6 we used the Cartesian product and Selection operations to obtain this list. However, the same result is obtained using the Equijoin operation:

$$(\Pi_{clientNo,\ fName,\ lName}(Client)) \bowtie_{Client.clientNo\ =\ Viewing.clientNo} (\Pi_{clientNo,\ propertyNo,\ comment}(Viewing))$$

or

$$Result \leftarrow TempClient \bowtie_{TempClient.clientNo\ =\ TempViewing.clientNo} TempViewing$$

The result of these operations was shown in Figure 5.8.

Natural join

R ⋈ S

The Natural join is an Equijoin of the two relations R and S over all common attributes *x*. One occurrence of each common attribute is eliminated from the result.

The Natural join operation performs an Equijoin over all the attributes in the two relations that have the same name. The degree of a Natural join is the sum of the degrees of the relations R and S less the number of attributes in *x*.

EXAMPLE 5.8 Natural join operation

List the names and comments of all clients who have viewed a property for rent.

In Example 5.7 we used the Equijoin to produce this list, but the resulting relation had two occurrences of the join attribute clientNo. We can use the Natural join to remove one occurrence of the clientNo attribute:

$$(\Pi_{clientNo,\ fName,\ lName}(Client)) \bowtie (\Pi_{clientNo,\ propertyNo,\ comment}(Viewing))$$

or

$$Result \leftarrow TempClient \bowtie TempViewing$$

The result of this operation is shown in Figure 5.9.

clientNo	fName	lName	propertyNo	comment
CR76	John	Kay	PG4	too remote
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR56	Aline	Stewart	PG36	
CR62	Mary	Tregear	PA14	no dining room

Figure 5.9 Natural join of restricted Client and Viewing relations.

Outer join

Often in joining two relations, a tuple in one relation does not have a matching tuple in the other relation; in other words, there is no matching value in the join attributes. We may want tuples from one of the relations to appear in the result even when there are no matching values in the other relation. This may be accomplished using the Outer join.

R ⟕ S The (left) Outer join is a join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation. Missing values in the second relation are set to null.

The Outer join is becoming more widely available in relational systems and is a specified operator in the SQL standard (see Section 6.3.7). The advantage of an Outer join is that information is preserved; that is, the Outer join preserves tuples that would have been lost by other types of join.

EXAMPLE 5.9 Left Outer join operation

Produce a status report on property viewings.

In this example, we want to produce a relation consisting of the properties that have been viewed with comments and those that have not been viewed. This can be achieved using the following Outer join:

$$(\Pi_{propertyNo,\ street,\ city}(PropertyForRent)) \text{ ⟕ } Viewing$$

The resulting relation is shown in Figure 5.10. Note that properties PL94, PG21, and PG16 have no viewings, but these tuples are still contained in the result with nulls for the attributes from the Viewing relation.

propertyNo	street	city	clientNo	viewDate	comment
PA14	16 Holhead	Aberdeen	CR56	24-May-13	too small
PA14	16 Holhead	Aberdeen	CR62	14-May-13	no dining room
PL94	6 Argyll St	London	null	null	null
PG4	6 Lawrence St	Glasgow	CR76	20-Apr- 13	too remote
PG4	6 Lawrence St	Glasgow	CR56	26-May-13	
PG36	2 Manor Rd	Glasgow	CR56	28-Apr- 13	
PG21	18 Dale Rd	Glasgow	null	null	null
PG16	5 Novar Dr	Glasgow	null	null	null

Figure 5.10 Left (natural) Outer join of PropertyForRent and Viewing relations.

Strictly speaking, Example 5.9 is a **Left (natural) Outer join**, as it keeps every tuple in the left-hand relation in the result. Similarly, there is a **Right Outer join** that keeps every tuple in the right-hand relation in the result. There is also a **Full Outer join** that keeps all tuples in both relations, padding tuples with nulls when no matching tuples are found.

Semijoin

R ▷$_F$ S The Semijoin operation defines a relation that contains the tuples of R that participate in the join of R with S satisfying the predicate *F*.

The Semijoin operation performs a join of the two relations and then projects over the attributes of the first operand. One advantage of a Semijoin is that it decreases the number of tuples that need to be handled to form the join. It is particularly useful for computing joins in distributed systems (see Sections 24.4.2 and 25.6.2). We can rewrite the Semijoin using the Projection and Join operations:

$$R \triangleright_F S = \Pi_A(R \bowtie_F S) \qquad A \text{ is the set of all attributes for } R$$

This is actually a Semi-Theta join. There are variants for the Semi-Equijoin and the Semi-Natural join.

EXAMPLE 5.10 Semijoin operation

List complete details of all staff who work at the branch in Glasgow.

If we are interested in seeing only the attributes of the Staff relation, we can use the following Semijoin operation, producing the relation shown in Figure 5.11.

$$\text{Staff} \triangleright_{\text{Staff.branchNo} = \text{Branch.branchNo}} (\sigma_{\text{city} = \text{'Glasgow'}} (\text{Branch}))$$

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Figure 5.11 Semijoin of Staff and Branch relations.

5.1.4 Division Operation

The Division operation is useful for a particular type of query that occurs quite frequently in database applications. Assume that relation R is defined over the attribute set A and relation S is defined over the attribute set B such that $B \subseteq A$ (B is a subset of A). Let $C = A - B$, that is, C is the set of attributes of R that are not attributes of S. We have the following definition of the Division operation.

R ÷ S The Division operation defines a relation over the attributes C that consists of the set of tuples from R that match the combination of **every** tuple in S.

We can express the Division operation in terms of the basic operations:

$$T \leftarrow \Pi_C(R)$$
$$T_2 \leftarrow \Pi_C((T_1 \times S) - R)$$
$$T \leftarrow T_1 - T_2$$

EXAMPLE 5.11 Division operation

Identify all clients who have viewed all properties with three rooms.

We can use the Selection operation to find all properties with three rooms followed by the Projection operation to produce a relation containing only these property numbers. We can then use the following Division operation to obtain the new relation shown in Figure 5.12.

$$(\Pi_{clientNo,\ propertyNo}(Viewing)) \div (\Pi_{propertyNo}(\sigma_{rooms\ =\ 3}(PropertyForRent)))$$

$\Pi_{clientNo,propertyNo}$(Viewing)

clientNo	propertyNo
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

$\Pi_{propertyNo}(\sigma_{rooms=3}$(PropertyForRent))

propertyNo
PG4
PG36

RESULT

clientNo
CR56

Figure 5.12 Result of the Division operation on the Viewing and PropertyForRent relations.

5.1.5 Aggregation and Grouping Operations

As well as simply retrieving certain tuples and attributes of one or more relations, we often want to perform some form of summation or **aggregation** of data, similar to the totals at the bottom of a report, or some form of **grouping** of data, similar to subtotals in a report. These operations cannot be performed using the basic relational algebra operations considered earlier. However, additional operations have been proposed, as we now discuss.

Aggregate operations

$\Im_{AL}(R)$	Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.

The main aggregate functions are:

- COUNT – returns the number of values in the associated attribute.
- SUM – returns the sum of the values in the associated attribute.
- AVG – returns the average of the values in the associated attribute.
- MIN – returns the smallest value in the associated attribute.
- MAX – returns the largest value in the associated attribute.

EXAMPLE 5.12 Aggregate operations

(a) How many properties cost more than £350 per month to rent?

We can use the aggregate function COUNT to produce the relation R shown in Figure 5.13(a):

myCount
5

(a)

myMin	myMax	myAverage
9000	30000	17000

(b)

Figure 5.13 Result of the Aggregate operations: (a) finding the number of properties whose rent is greater than £350; (b) finding the minimum, maximum, and average staff salary.

$$\rho_R(\text{myCount})\ \Im_{\text{COUNT propertyNo}}\ (\sigma_{\text{rent} > 350}\ (\text{PropertyForRent}))$$

(b) Find the minimum, maximum, and average staff salary.

We can use the aggregate functions—MIN, MAX, and AVERAGE—to produce the relation R shown in Figure 5.13(b) as follows:

$$\rho_R(\text{myMin, myMax, myAverage})\ \Im_{\text{MIN salary, MAX salary, AVERAGE salary}}\ (\text{Staff})$$

Grouping operation

$_{GA}\Im_{AL}(R)$

Groups the tuples of relation R by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. AL contains one or more (<aggregate_function>, <attribute>) pairs. The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

The general form of the grouping operation is as follows:

$$a_1, a_2, \ldots, a_n\ \Im\ _{<A_pa_p>, <A_qa_q>, \ldots, <A_za_z>}\ (R)$$

where R is any relation, $a_1, a_2, \ldots, a_n$ are attributes of R on which to group, $a_p, a_q, \ldots, a_z$ are other attributes of R, and $A_p, A_q, \ldots, A_2$ are aggregate functions. The tuples of R are partitioned into groups such that:

- all tuples in a group have the same value for $a_1, a_2, \ldots, a_n$;
- tuples in different groups have different values for $a_1, a_2, \ldots, a_n$.

We illustrate the use of the grouping operation with the following example.

EXAMPLE 5.13 Grouping operation

Find the number of staff working in each branch and the sum of their salaries.

We first need to group tuples according to the branch number, branchNo, and then use the aggregate functions COUNT and SUM to produce the required relation. The relational algebra expression is as follows:

$$\rho_R(\text{branchNo, myCount, mySum})\ _{\text{branchNo}}\ \Im\ _{\text{COUNT staffNo, SUM salary}}\ (\text{Staff})$$

The resulting relation is shown in Figure 5.14.

branchNo	myCount	mySum
B003	3	54000
B005	2	39000
B007	1	9000

Figure 5.14 Result of the grouping operation to find the number of staff working in each branch and the sum of their salaries.

5.1.6 Summary of the Relational Algebra Operations

The relational algebra operations are summarized in Table 5.1.

TABLE 5.1 Operations in the relational algebra.

OPERATION	NOTATION	FUNCTION
Selection	$\sigma_{predicate}(R)$	Produces a relation that contains only those tuples of R that satisfy the specified *predicate*.
Projection	$\Pi_{a_1,\ldots,a_n}(R)$	Produces a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.
Union	$R \cup S$	Produces a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.
Set difference	$R - S$	Produces a relation that contains all the tuples in R that are not in S. R and S must be union-compatible.
Intersection	$R \cap S$	Produces a relation that contains all the tuples in both R and S. R and S must be union-compatible.
Cartesian product	$R \times S$	Produces a relation that is the concatenation of every tuple of relation R with every tuple of relation S.
Theta join	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate *F* from the Cartesian product of R and S.
Equijoin	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate *F* (which contains only equality comparisons) from the Cartesian product of R and S.
Natural join	$R \bowtie S$	An Equijoin of the two relations R and S over all common attributes *x*. One occurrence of each common attribute is eliminated.
(Left) Outer join	$R \rtimes S$	A join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation.
Semijoin	$R \triangleright_F S$	Produces a relation that contains the tuples of R that participate in the join of R with S satisfying the predicate *F*.
Division	$R \div S$	Produces a relation that consists of the set of tuples from R defined over the attributes C that match the combination of **every** tuple in S, where C is the set of attributes that are in R but not in S.
Aggregate	$\Im_{AL}(R)$	Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.
Grouping	${}_{GA}\Im_{AL}(R)$	Groups the tuples of relation R by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. AL contains one or more (<aggregate_function>, <attribute>) pairs. The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

5.2 The Relational Calculus

A certain order is always explicitly specified in a relational algebra expression and a strategy for evaluating the query is implied. In the relational calculus, there is no description of how to evaluate a query; a relational calculus query specifies *what* is to be retrieved rather than *how* to retrieve it.

The relational calculus is not related to differential and integral calculus in mathematics, but takes its name from a branch of symbolic logic called **predicate calculus**. When applied to databases, it is found in two forms: **tuple** relational calculus, as originally proposed by Codd (1972a), and **domain** relational calculus, as proposed by Lacroix and Pirotte (1977).

In first-order logic or predicate calculus, a **predicate** is a truth-valued function with arguments. When we substitute values for the arguments, the function yields an expression, called a **proposition**, which can be either true or false. For example, the sentences, "John White is a member of staff" and "John White earns more than Ann Beech" are both propositions, because we can determine whether they are true or false. In the first case, we have a function, "is a member of staff," with one argument (John White); in the second case, we have a function, "earns more than," with two arguments (John White and Ann Beech).

If a predicate contains a variable, as in "x is a member of staff," there must be an associated **range** for x. When we substitute some values of this range for x, the proposition may be true; for other values, it may be false. For example, if the range is the set of all people and we replace x by John White, the proposition "John White is a member of staff" is true. If we replace x by the name of a person who is not a member of staff, the proposition is false.

If P is a predicate, then we can write the set of all x such that P is true for x, as:

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

We may connect predicates by the logical connectives $\wedge$ (AND), $\vee$ (OR), and $\sim$ (NOT) to form compound predicates.

5.2.1 Tuple Relational Calculus

In the tuple relational calculus, we are interested in finding tuples for which a predicate is true. The calculus is based on the use of **tuple variables**. A tuple variable is a variable that "ranges over" a named relation: that is, a variable whose only permitted values are tuples of the relation. (The word "range" here does not correspond to the mathematical use of range, but corresponds to a mathematical domain.) For example, to specify the range of a tuple variable S as the Staff relation, we write:

Staff(S)

To express the query "Find the set of all tuples S such that $F(S)$ is true," we can write:

$$\{S \mid F(S)\}$$

F is called a **formula** (**well-formed formula**, or **wff** in mathematical logic). For example, to express the query "Find the staffNo, fName, lName, position,

sex, DOB, salary, and branchNo of all staff earning more than £10,000," we can write:

$$\{S \mid Staff(S) \wedge S.salary > 10000\}$$

S.salary means the value of the salary attribute for the tuple variable S. To retrieve a particular attribute, such as salary, we would write:

$$\{S.salary \mid Staff(S) \wedge S.salary > 10000\}$$

The existential and universal quantifiers

There are two **quantifiers** we can use with formulae to tell how many instances the predicate applies to. The **existential quantifier** $\exists$ ("there exists") is used in formulae that must be true for at least one instance, such as:

$$Staff(S) \wedge (\exists B)\,(Branch(B) \wedge (B.branchNo = S.branchNo) \wedge B.city = \text{'London'})$$

This means, "There exists a Branch tuple that has the same branchNo as the branchNo of the current Staff tuple, S, and is located in London." The **universal quantifier** $\forall$ ("for all") is used in statements about every instance, such as:

$$(\forall B)\,(B.city \neq \text{'Paris'})$$

This means, "For all Branch tuples, the address is not in Paris." We can apply a generalization of De Morgan's laws to the existential and universal quantifiers. For example:

$$(\exists X)(F(X)) \equiv \sim (\forall X)(\sim(F(X)))$$

$$(\forall X)(F(X)) \equiv \sim(\exists X)(\sim(F(X)))$$

$$(\exists X)(F_1(X) \wedge F_2(X)) \equiv \sim(\forall X)(\sim(F_1(X)) \vee \sim(F_2(X)))$$

$$(\forall X)(F_1(X) \wedge F_2(X)) \equiv \sim(\exists X)(\sim(F_1(X)) \vee \sim(F_2(X)))$$

Using these equivalence rules, we can rewrite the previous formula as:

$$\sim(\exists B)\,(B.city = \text{'Paris'})$$

which means, "There are no branches with an address in Paris."

Tuple variables that are qualified by $\forall$ or $\exists$ are called **bound variables**; the other tuple variables are called **free variables**. The only free variables in a relational calculus expression should be those on the left side of the bar (|). For example, in the following query:

$$\{S.fName, S.lName \mid Staff(S) \wedge (\exists B)\,(Branch(B) \wedge (B.branchNo = S.branchNo) \wedge B.city = \text{'London'})\}$$

S is the only free variable and S is then bound successively to each tuple of Staff.

Expressions and formulae

As with the English alphabet, in which some sequences of characters do not form a correctly structured sentence, in calculus not every sequence of formulae is acceptable. The formulae should be those sequences that are unambiguous and make sense. An expression in the tuple relational calculus has the following general form:

$$\{S_1.a_1, S_2.a_2, \ldots, S_n.a_n \mid F(S_1, S_2, \ldots, S_m)\} \quad m \geq n$$

where $S_1, S_2, \ldots, S_n \ldots, S_m$ are tuple variables; each a_i is an attribute of the relation over which S_i ranges; and F is a formula. A (well-formed) formula is made out of one or more *atoms*, where an atom has one of the following forms:

- $R(S_i)$, where S_i is a tuple variable and R is a relation.
- $S_1.a_1 \ \theta \ S_j.a_2$, where S_i and S_j are tuple variables, a_1, is an attribute of the relation over which S_i ranges, a_2 is an attribute of the relation over which S_j ranges, and θ is one of the comparison operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$); the attributes a_1 and a_2 must have domains whose members can be compared by θ.
- $S_i.a_1 \ \theta \ c$, where S_i is a tuple variable, a_1 is an attribute of the relation over which S_i ranges, c is a constant from the domain of attribute a_1, and θ is one of the comparison operators.

We recursively build up formulae from atoms using the following rules:

- An atom is a formula.
- If F_1 and F_2 are formulae, so are their conjunction $F_1 \wedge F_2$, their disjunction $F_1 \vee F_2$, and the negation $\sim F_1$.
- If F is a formula with free variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

EXAMPLE 5.14 Tuple relational calculus

(a) List the names of all managers who earn more than £25,000.

$$\{S.fName, S.lName \mid Staff(S) \wedge S.position = \text{'Manager'} \wedge S.salary > 25000\}$$

(b) List the staff who manage properties for rent in Glasgow.

$$\{S \mid Staff(S) \wedge (\exists P)\ (PropertyForRent(P) \wedge (P.staffNo = S.staffNo) \wedge P.city = \text{'Glasgow'})\}$$

The staffNo attribute in the PropertyForRent relation holds the staff number of the member of staff who manages the property. We could reformulate the query as: "For each member of staff whose details we want to list, there exists a tuple in the relation PropertyForRent for that member of staff with the value of the attribute city in that tuple being Glasgow."

Note that in this formulation of the query, there is no indication of a strategy for executing the query—the DBMS is free to decide the operations required to fulfil the request and the execution order of these operations. On the other hand, the equivalent relational algebra formulation would be: "Select tuples from PropertyForRent such that the city is Glasgow and perform their join with the Staff relation," which has an implied order of execution.

(c) List the names of staff who currently do not manage any properties.

$$\{S.fName, S.lName \mid Staff(S) \wedge (\sim(\exists P)\ (PropertyForRent(P) \wedge (S.staffNo = P.staffNo)))\}$$

Using the general transformation rules for quantifiers given previously, we can rewrite this as:

$$\{S.fName, S.lName \mid Staff(S) \wedge ((\forall P)\ (\sim PropertyForRent(P) \vee \sim(S.staffNo = P.staffNo)))\}$$

(d) List the names of clients who have viewed a property for rent in Glasgow.

$$\{C.fName, C.lName \mid Client(C) \wedge ((\exists V)\ (\exists P)\ (Viewing(V) \wedge PropertyForRent(P) \wedge (C.clientNo = V.clientNo) \wedge (V.propertyNo = P.propertyNo) \wedge P.city = \text{'Glasgow'}))\}$$

To answer this query, note that we can rephrase "clients who have viewed a property in Glasgow" as "clients for whom there exists some viewing of some property in Glasgow."

(e) List all cities where there is either a branch office or a property for rent.

$$\{T.city \mid (\exists B)\ (Branch(B) \wedge (B.city = T.city)) \vee (\exists P)\ (PropertyForRent(P) \wedge (P.city = T.city))\}$$

Compare this with the equivalent relational algebra expression given in Example 5.3.

(f) List all the cities where there is a branch office but no properties for rent.

$$\{B.city \mid Branch(B) \wedge (\sim(\exists P)\ (PropertyForRent(P) \wedge (B.city = P.city)))\}$$

Compare this with the equivalent relational algebra expression given in Example 5.4.

(g) List all the cities where there is both a branch office and at least one property for rent.

$$\{B.city \mid Branch(B) \wedge ((\exists P)\ (PropertyForRent(P) \wedge (B.city = P.city)))\}$$

Compare this with the equivalent relational algebra expression given in Example 5.5.

Safety of expressions

Before we complete this section, we should mention that it is possible for a calculus expression to generate an infinite set. For example:

$$\{S \mid \sim Staff(S)\}$$

would mean the set of all tuples that are not in the Staff relation. Such an expression is said to be **unsafe**. To avoid this, we have to add a restriction that all values that appear in the result must be values in the *domain* of the expression E, denoted $dom(E)$. In other words, the domain of E is the set of all values that appear explicitly in E or that appear in one or more relations whose names appear in E. In this example, the domain of the expression is the set of all values appearing in the Staff relation.

An expression is *safe* if all values that appear in the result are values from the domain of the expression. The previous expression is not safe, as it will typically include tuples from outside the Staff relation (and so outside the domain of the expression). All other examples of tuple relational calculus expressions in this section are safe. Some authors have avoided this problem by using range variables that are defined by a separate RANGE statement. The interested reader is referred to Date (2000).

5.2.2 Domain Relational Calculus

In the tuple relational calculus, we use variables that range over tuples in a relation. In the domain relational calculus, we also use variables, but in this case the variables take their values from *domains* of attributes rather than tuples of relations. An expression in the domain relational calculus has the following general form:

$$\{d_1, d_2, \ldots, d_n \mid F(d_1, d_2, \ldots, d_m)\} \qquad m \geq n$$

where $d_1, d_2, \ldots, d_n, \ldots, d_m$ represent domain variables and $F(d_1, d_2, \ldots, d_m)$ represents a formula composed of atoms, where each atom has one of the following forms:

- $R(d_1, d_2, \ldots, d_n)$, where R is a relation of degree n and each d_i is a domain variable.

- $d_i \, \theta \, dj$, where d_i and d_j are domain variables and θ is one of the comparison operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$); the domains d_i and d_j must have members that can be compared by θ.
- $d_i \, \theta \, c$, where d_i is a domain variable, c is a constant from the domain of d_i, and θ is one of the comparison operators.

We recursively build up formulae from atoms using the following rules:

- An atom is a formula.
- If F_1 and F_2 are formulae, so are their conjunction $F_1 \wedge F_2$, their disjunction $F_1 \vee F_2$, and the negation $\sim F_1$.
- If F is a formula with domain variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

EXAMPLE 5.15 Domain relational calculus

In the following examples, we use the following shorthand notation:

$(\exists d_1, d_2, \ldots, d_n)$ in place of $(\exists d_1), (\exists d_2), \ldots, (\exists d_n)$

(a) Find the names of all managers who earn more than £25,000.

{fN, lN | (∃sN, posn, sex, DOB, sal, bN) (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) ∧
posn = 'Manager' ∧ sal > 25000)}

If we compare this query with the equivalent tuple relational calculus query in Example 5.14(a), we see that each attribute is given a (variable) name. The condition Staff(sN, fN, . . . , bN) ensures that the domain variables are restricted to be attributes of the same tuple. Thus, we can use the formula posn = 'Manager', rather than Staff.position = 'Manager'. Also note the difference in the use of the existential quantifier. In the tuple relational calculus, when we write ∃posn for some tuple variable posn, we bind the variable to the relation Staff by writing Staff(posn). On the other hand, in the domain relational calculus posn refers to a domain value and remains unconstrained until it appears in the subformula Staff(sN, fN, lN, posn, sex, DOB, sal, bN), when it becomes constrained to the position values that appear in the Staff relation.

For conciseness, in the remaining examples in this section we quantify only those domain variables that actually appear in a condition (in this example, posn and sal).

(b) List the staff who manage properties for rent in Glasgow.

{sN, fN, lN, posn, sex, DOB, sal, bN | (∃sN1, cty) (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) ∧
PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1) ∧ (sN = sN1) ∧
cty = 'Glasgow')}

This query can also be written as:

{sN, fN, lN, posn, sex, DOB, sal, bN | (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) ∧
PropertyForRent(pN, st, 'Glasgow', pc, typ, rms, rnt, oN, sN, bN1))}

In this version, the domain variable cty in PropertyForRent has been replaced with the constant "Glasgow" and the same domain variable sN, which represents the staff number, has been repeated for Staff and PropertyForRent.

(c) List the names of staff who currently do not manage any properties for rent.

{fN, lN | (∃sN) (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) ∧
(~(∃sN1) (PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1) ∧ (sN = sN1))))}

(d) List the names of clients who have viewed a property for rent in Glasgow.

$$\{fN, lN \mid (\exists cN, cN1, pN, pN1, cty)(Client(cN, fN, lN, tel, pT, mR, eM) \wedge Viewing(cN1, pN1, dt, cmt) \wedge PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN, bN) \wedge (cN = cN1) \wedge (pN = pN1) \wedge cty = \text{'Glasgow'})\}$$

(e) List all cities where there is either a branch office or a property for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \vee PropertyForRent(pN, st1, cty, pc1, typ, rms, rnt, oN, sN, bN1))\}$$

(f) List all the cities where there is a branch office but no properties for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \wedge (\neg(\exists cty1)(PropertyForRent(pN, st1, cty1, pc1, typ, rms, rnt, oN, sN, bN1) \wedge (cty = cty1))))\}$$

(g) List all the cities where there is both a branch office and at least one property for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \wedge (\exists cty1)(PropertyForRent(pN, st1, cty1, pc1, typ, rms, rnt, oN, sN, bN1) \wedge (cty = cty1)))\}$$

These queries are **safe**. When the domain relational calculus is restricted to safe expressions, it is equivalent to the tuple relational calculus restricted to safe expressions, which in turn is equivalent to the relational algebra. This means that for every relational algebra expression there is an equivalent expression in the relational calculus, and for every tuple or domain relational calculus expression there is an equivalent relational algebra expression.

5.3 Other Languages

Although the relational calculus is hard to understand and use, it was recognized that its nonprocedural property is exceedingly desirable, and this resulted in a search for other easy-to-use nonprocedural techniques. This led to another two categories of relational languages: transform-oriented and graphical.

Transform-oriented languages are a class of nonprocedural languages that use relations to transform input data into required outputs. These languages provide easy-to-use structures for expressing what is desired in terms of what is known. SQUARE (Boyce *et al.*, 1975), SEQUEL (Chamberlin *et al.*, 1976), and SEQUEL's offspring, SQL, are all transform-oriented languages. We discuss SQL in Chapters 6–9.

Graphical languages provide the user with a picture or illustration of the structure of the relation. The user fills in an example of what is wanted and the system returns the required data in that format. QBE (Query-By-Example) is an example of a graphical language (Zloof, 1977). We demonstrate the capabilities of QBE in Appendix M.

Another category is **fourth-generation languages** (**4GLs**), which allow a complete customized application to be created using a limited set of commands in a user-friendly, often menu-driven environment (see Section 2.2). Some systems accept a form of *natural language*, a restricted version of natural English, sometimes called a **fifth-generation language** (**5GL**), although this development is still at an early stage.

Chapter Summary

- The **relational algebra** is a (high-level) procedural language: it can be used to tell the DBMS how to build a new relation from one or more relations in the database. The **relational calculus** is a nonprocedural language: it can be used to formulate the definition of a relation in terms of one or more database relations. However, formally the relational algebra and relational calculus are equivalent to one another: for every expression in the algebra, there is an equivalent expression in the calculus (and vice versa).
- The relational calculus is used to measure the selective power of relational languages. A language that can be used to produce any relation that can be derived using the relational calculus is said to be **relationally complete**. Most relational query languages are relationally complete but have more expressive power than the relational algebra or relational calculus because of additional operations such as calculated, summary, and ordering functions.
- The five fundamental operations in relational algebra—Selection, Projection, Cartesian product, Union, and Set difference—perform most of the data retrieval operations that we are interested in. In addition, there are also the Join, Intersection, and Division operations, which can be expressed in terms of the five basic operations.
- The **relational calculus** is a formal nonprocedural language that uses predicates. There are two forms of the relational calculus: tuple relational calculus and domain relational calculus.
- In the **tuple relational calculus**, we are interested in finding tuples for which a predicate is true. A tuple variable is a variable that "ranges over" a named relation: that is, a variable whose only permitted values are tuples of the relation.
- In the **domain relational calculus**, domain variables take their values from domains of attributes rather than tuples of relations.
- The relational algebra is logically equivalent to a safe subset of the relational calculus (and vice versa).
- Relational data manipulation languages are sometimes classified as **procedural** or **nonprocedural**, **transform-oriented**, **graphical**, **fourth-generation**, or **fifth-generation**.

Review Questions

5.1 What is the difference between a procedural and a nonprocedural language? How would you classify the relational algebra and relational calculus?

5.2 Explain the following terms:
(a) relationally complete
(b) closure of relational operations.

5.3 Define the five basic relational algebra operations. Define the Join, Intersection, and Division operations in terms of these five basic operations.

5.4 Discuss the differences between the five Join operations: Theta join, Equijoin, Natural join, Outer join, and Semi-join. Give examples to illustrate your answer.

5.5 Compare and contrast the tuple relational calculus with domain relational calculus. In particular, discuss the distinction between tuple and domain variables.

5.6 Define the structure of a (well-formed) formula in both the tuple relational calculus and domain relational calculus.

5.7 Explain how a relational calculus expression can be unsafe. Illustrate your answer with an example. Discuss how to ensure that a relational calculus expression is safe.

Exercises

For the following exercises, use the Hotel schema defined at the start of the Exercises at the end of Chapter 4.

5.8 Describe the relations that would be produced by the following relational algebra operations:

(a) $\Pi_{hotelNo}(\sigma_{price > 50}(Room))$

(b) $\sigma_{Hotel.hotelNo = Room.hotelNo}(Hotel \times Room)$

(c) $\Pi_{hotelName}(Hotel \bowtie_{Hotel.hotelNo = Room.hotelNo}(\sigma_{price > 50}(Room)))$

(d) $Guest \ltimes (\sigma_{dateTo \geq '1\text{-}Jan\text{-}2007'}(Booking))$

(e) $Hotel \triangleright_{Hotel.hotelNo = Room.hotelNo}(\sigma_{price > 50}(Room))$

(f) $\Pi_{guestName, hotelNo}(Booking \bowtie_{Booking.guestNo = Guest.guestNo} Guest) \div \Pi_{hotelNo}(\sigma_{city = 'London'}(Hotel))$

5.9 Provide the equivalent tuple relational calculus and domain relational calculus expressions for each of the relational algebra queries given in Exercise 5.8.

5.10 Describe the relations that would be produced by the following tuple relational calculus expressions:

(a) $\{H.hotelName \mid Hotel(H) \wedge H.city = \text{'London'}\}$

(b) $\{H.hotelName \mid Hotel(H) \wedge (\exists R)\,(Room(R) \wedge H.hotelNo = R.hotelNo \wedge R.price > 50)\}$

(c) $\{H.hotelName \mid Hotel(H) \wedge (\exists B)\,(\exists G)\,(Booking(B) \wedge Guest(G) \wedge H.hotelNo = B.hotelNo \wedge B.guestNo = G.guestNo \wedge G.guestName = \text{'John Smith'})\}$

(d) $\{H.hotelName, G.guestName, B1.dateFrom, B2.dateFrom \mid Hotel(H) \wedge Guest(G) \wedge Booking(B1) \wedge Booking(B2) \wedge H.hotelNo = B1.hotelNo \wedge G.guestNo = B1.guestNo \wedge B2.hotelNo = B1.hotelNo \wedge B2.guestNo = B1.guestNo \wedge B2.dateFrom \neq B1.dateFrom\}$

Hotel (hotelNo, hotelName, city)
Room (roomNo, hotelNo, type, price)
Booking (hotelNo, guestNo, dateFrom, dateTo, roomNo)
Guest (guestNo, guestName, guestAddress)

where Hotel *contains hotel details and* hotelNo *is the primary key;*
Room contains room details for each hotel and (roomNo, hotelNo) forms the primary key;
Booking contains details of bookings and (hotelNo, guestNo, dateFrom) forms the primary key;
Guest contains guest details and guestNo is the primary key.

5.13 Using relational algebra, create a view of all rooms in the Grosvenor Hotel, excluding price details. What are the advantages of this view?

The following tables form part of a database held in an RDBMS:

Employee	(empNo, fName, lName, address, DOB, sex, position, deptNo)
Department	(deptNo, deptName, mgrEmpNo)
Project	(projNo, projName, deptNo)
WorksOn	(empNo, projNo, dateWorked, hoursWorked)

where	Employee	contains employee details and **empNo** is the key.
	Department	contains department details and **deptNo** is the key. **mgrEmpNo** identifies the employee who is the manager of the department. There is only one manager for each department.
	Project	contains details of the projects in each department and the key is **projNo** (no two departments can run the same project).
and	WorksOn	contains details of the hours worked by employees on each project, and **empNo/projNo/dateWorked** form the key.

Formulate the following queries in relational algebra, tuple relational calculus, and domain relational calculus.

5.14 List all employees.

5.15 List all the details of employees who are female.

5.16 List the names and addresses of all employees who are managers.

5.17 Produce a list of the names and addresses of all employees who work for the IT department.

5.18 Produce a list of the names of all employees who work on the SCCS project.

5.19 Produce a complete list of all managers who are due to retire this year, in alphabetical order of surname.

Formulate the following queries in relational algebra.

5.20 Find out how many employees are managed by "James Adam."

5.21 Produce a report of the total hours worked by each employee.

5.22 For each project on which more than two employees worked, list the project number, project name, and the number of employees who work on that project.

5.23 List the total number of employees in each department for those departments with more than 10 employees. Create an appropriate heading for the columns of the results table.

The following tables form part of a Library database held in an RDBMS:

Book	**(ISBN, title, edition, year)**
BookCopy	**(copyNo, ISBN, available)**
Borrower	**(borrowerNo, borrowerName, borrowerAddress)**
BookLoan	**(copyNo, dateOut, dateDue, borrowerNo)**

where

Book	contains details of book titles in the library and the ISBN is the key.
BookCopy	contains details of the individual copies of books in the library and **copyNo** is the key. ISBN is a foreign key identifying the book title.
Borrower	contains details of library members who can borrow books and **borrowerNo** is the key.
BookLoan	contains details of the book copies that are borrowed by library members and **copyNo/dateOut** forms the key. **borrowerNo** is a foreign key identifying the borrower.

Formulate the following queries in relational algebra, tuple relational calculus, and domain relational calculus.

5.24 List all book titles.

5.25 List all borrower details.

5.26 List all book titles published in the year 2012.

5.27 List all copies of book titles that are available for borrowing.

5.28 List all copies of the book title *Lord of the Rings* that are available for borrowing.

5.29 List the names of borrowers who currently have the book title *Lord of the Rings* on loan.

5.30 List the names of borrowers with overdue books.

Formulate the following queries in relational algebra.

5.31 How many copies of ISBN "0-321-52306-7" are there?

5.32 How many copies of ISBN "0-321-52306-7" are currently available?

5.33 How many times has the book title with ISBN "0-321-52306-7" been borrowed?

5.34 Produce a report of book titles that have been borrowed by "Peter Bloomfield."

5.35 For each book title with more than three copies, list the names of library members who have borrowed them.

5.36 Produce a report with the details of borrowers who currently have books overdue.

5.37 Produce a report detailing how many times each book title has been borrowed.

5.38 Analyze the RDBMSs that you are currently using. What types of relational language does the system provide? For each of the languages provided, what are the equivalent operations for the eight relational algebra operations defined in Section 5.1?