Structure of this Chapter

In Section 4.1 we examine the relational algebra and in Section 4.2 we examine two forms of the relational calculus: tuple relational calculus and domain relational calculus. In Section 4.3 we briefly discuss some other relational languages. We use the *DreamHome* rental database instance shown in Figure 3.3 to illustrate the operations.

In Chapters 5 and 6 we examine SQL (Structured Query Language), the formal and *de facto* standard language for RDBMSs, which has constructs based on the tuple relational calculus. In Chapter 7 we examine QBE (Query-By-Example), another highly popular visual query language for RDBMSs, which is in part based on the domain relational calculus.

The Relational Algebra 4.1

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 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 4.1.

The Selection and Projection operations are **unary** operations, since 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.

Unary Operations 4.1.1

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

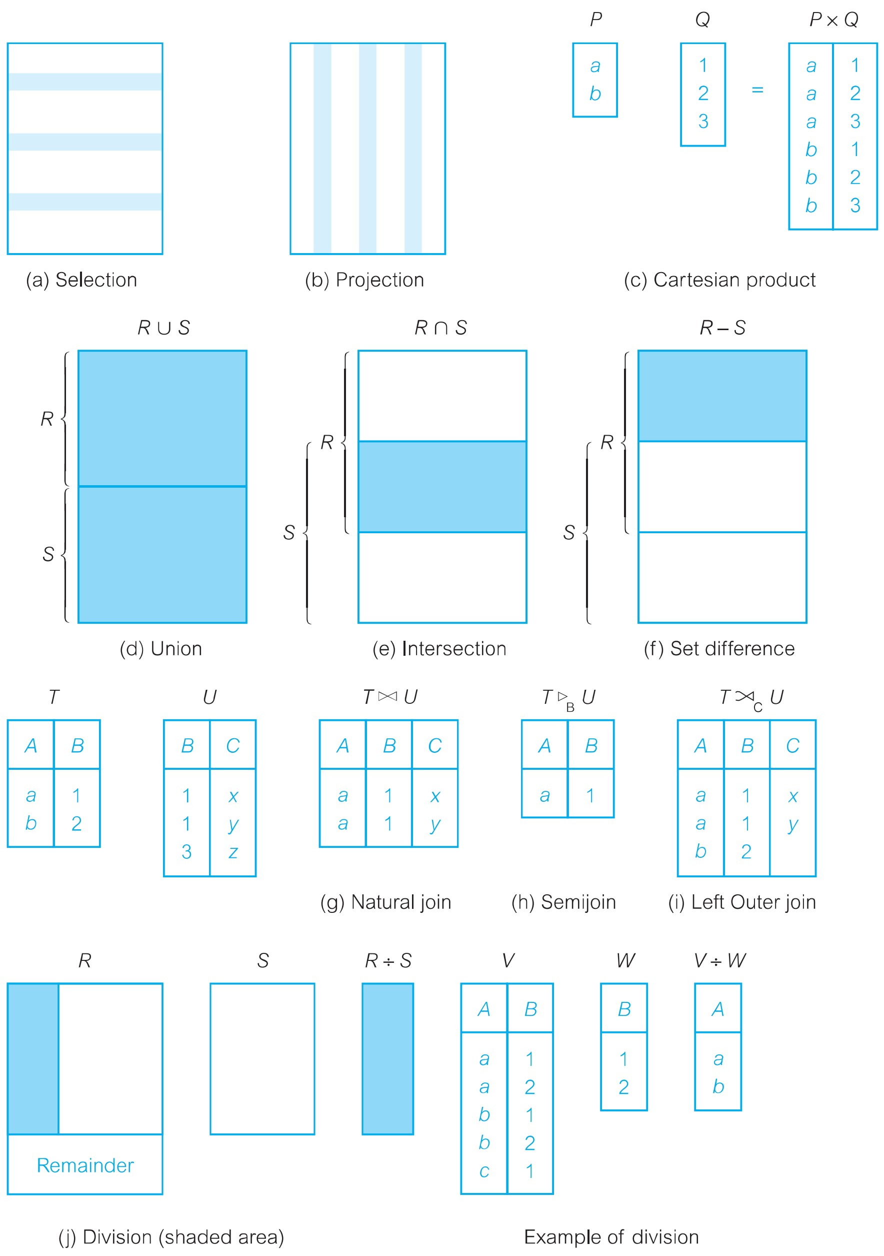

Figure 4.1 Illustration showing the function of the relational algebra operations.

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 4.1 Selection operation

List all staff with a salary greater than £10,000.

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

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 £10,000. The result of this operation is shown in Figure 4.2. More complex predicates can be generated using the logical operators $\wedge$ (AND), $\vee$ (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 4.2 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 4.2 Projection operation

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

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

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 4.3.

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

4.1.2 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 4.3 Union operation

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

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

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 4.4.

city
London
Aberdeen
Glasgow
Bristol

Figure 4.4 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 4.4 Set difference operation

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

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

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 4.5.

city
Bristol

Figure 4.5 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.

Example 4.5 Intersection operation

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

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

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 4.6.

city
Aberdeen
London
Glasgow

Figure 4.6 Intersection based on city attribute from the Branch and PropertyForRent relations.

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 4.6 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:

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

This result of this operation is shown in Figure 4.7. 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:

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

The result of this operation is shown in Figure 4.8.

Figure 4.7
Cartesian product of reduced Client and Viewing relations.

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 4.8
Restricted 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

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:

$\text{TempViewing(clientNo, propertyNo, comment)} \leftarrow \Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing})$
$\text{TempClient(clientNo, fName, lName)} \leftarrow \Pi_{\text{clientNo, fName, lName}}(\text{Client})$
$\text{Comment(clientNo, fName, lName, vclientNo, propertyNo, comment)} \leftarrow$
$\quad \text{TempClient} \times \text{TempViewing}$
$\text{Result} \leftarrow \sigma_{\text{clientNo = vclientNo}}(\text{Comment})$

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.

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

Join Operations 4.1.3

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 21.4.3.

There are various forms of 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 $R.a_i \; \theta \; S.b_i$ where θ may be one of the comparison operators (<, ≤, >, ≥, =, ≠).

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 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 4.6.

Example 4.7 Equijoin operation

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

In Example 4.6 we used the Cartesian product and Selection operations to obtain this list. However, the same result is obtained using the Equijoin operation:

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

or

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

The result of these operations was shown in Figure 4.8.

Natural join

$R \bowtie S$ 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 4.8 Natural join operation

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

In Example 4.7 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:

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

or

$$\text{Result} \leftarrow \text{TempClient} \bowtie \text{TempViewing}$$

The result of this operation is shown in Figure 4.9.

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 4.9 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 5.3.7). 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 4.9 Left Outer join operation

Produce a status report on property viewings.

In this case, 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:

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

The resulting relation is shown in Figure 4.10. 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.

Figure 4.10
Left (natural) Outer join of PropertyForRent and Viewing relations.

propertyNo	street	city	clientNo	viewDate	comment
PA14	16 Holhead	Aberdeen	CR56	24-May-04	too small
PA14	16 Holhead	Aberdeen	CR62	14-May-04	no dining room
PL94	6 Argyll St	London	**null**	**null**	**null**
PG4	6 Lawrence St	Glasgow	CR76	20-Apr-04	too remote
PG4	6 Lawrence St	Glasgow	CR56	26-May-04	
PG36	2 Manor Rd	Glasgow	CR56	28-Apr-04	
PG21	18 Dale Rd	Glasgow	**null**	**null**	**null**
PG16	5 Novar Dr	Glasgow	**null**	**null**	**null**

Strictly speaking, Example 4.9 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

$R \triangleright_F S$ The Semijoin operation defines a relation that contains the tuples of R that participate in the join of R with S.

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 22.4.2 and 23.6.2). We can rewrite the Semijoin using the Projection and Join operations:

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

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

Example 4.10 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 4.11.

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

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 4.11 Semijoin of Staff and Branch relations.

4.1.4 Division Operation

The Division operation is useful for a particular type of query that occurs quite frequently in database applications. Assume 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:

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

Example 4.11 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 4.12.

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

21.3.1 Transformation Rules for the Relational Algebra Operations

By applying transformation rules, the optimizer can transform one relational algebra expression into an equivalent expression that is known to be more efficient. We will use these rules to restructure the (canonical) relational algebra tree generated during query decomposition. Proofs of the rules can be found in Aho *et al*. (1979). In listing these rules, we use three relations R, S, and T, with R defined over the attributes $A = \{A_1, A_2, \ldots, A_n\}$, and S defined over $B = \{B_1, B_2, \ldots, B_n\}$; p, q, and r denote predicates, and L, L_1, L_2, M, M_1, M_2, and N denote sets of attributes.

(1) Conjunctive Selection operations can cascade into individual Selection operations (and vice versa).

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

This transformation is sometimes referred to as *cascade of selection*. For example:

$$\sigma_{branchNo='B003' \wedge salary>15000}(Staff) = \sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff))$$

(2) Commutativity of Selection operations.

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

For example:

$$\sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff)) = \sigma_{salary>15000}(\sigma_{branchNo='B003'}(Staff))$$

(3) In a sequence of Projection operations, only the last in the sequence is required.

$$\Pi_L \Pi_M \ldots \Pi_N(R) = \Pi_L(R)$$

For example:

$$\Pi_{lName}\Pi_{branchNo,\ lName}(Staff) = \Pi_{lName}(Staff)$$

(4) Commutativity of Selection and Projection.
If the predicate p involves only the attributes in the projection list, then the Selection and Projection operations commute:

$$\Pi_{A_1, \ldots, A_m}(\sigma_p(R)) = \sigma_p(\Pi_{A_1, \ldots, A_m}(R)) \qquad \text{where } p \in \{A_1, A_2, \ldots, A_m\}$$

For example:

$$\Pi_{fName,\ lName}(\sigma_{lName='Beech'}(Staff)) = \sigma_{lName='Beech'}(\Pi_{fName,\ lName}(Staff))$$

(5) Commutativity of Theta join (and Cartesian product).

$$R \bowtie_p S = S \bowtie_p R$$

$$R \times S = S \times R$$

As the Equijoin and Natural join are special cases of the Theta join, then this rule also applies to these Join operations. For example, using the Equijoin of Staff and Branch:

$$\text{Staff} \bowtie_{\text{Staff.branchNo=Branch.branchNo}} \text{Branch} = \text{Branch} \bowtie_{\text{Staff.branchNo=Branch.branchNo}} \text{Staff}$$

(6) Commutativity of Selection and Theta join (or Cartesian product).
If the selection predicate involves only attributes of one of the relations being joined, then the Selection and Join (or Cartesian product) operations commute:

$$\sigma_p(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r S$$

$$\sigma_p(R \times S) = (\sigma_p(R)) \times S \qquad \text{where } p \in \{A_1, A_2, \ldots, A_n\}$$

Alternatively, if the selection predicate is a conjunctive predicate of the form ($p \wedge q$), where p involves only attributes of R, and q involves only attributes of S, then the Selection and Theta join operations commute as:

$$\sigma_{p \wedge q}(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r (\sigma_q(S))$$

$$\sigma_{p \wedge q}(R \times S) = (\sigma_p(R)) \times (\sigma_q(S))$$

For example:

$$\sigma_{\text{position='Manager'} \wedge \text{city='London'}}(\text{Staff} \bowtie_{\text{Staff.branchNo=Branch.branchNo}} \text{Branch}) =$$

$$(\sigma_{\text{position='Manager'}}(\text{Staff})) \bowtie_{\text{Staff.branchNo=Branch.branchNo}} (\sigma_{\text{city='London'}}(\text{Branch}))$$

(7) Commutativity of Projection and Theta join (or Cartesian product).
If the projection list is of the form $L = L_1 \cup L_2$, where L_1 involves only attributes of R, and L_2 involves only attributes of S, then provided the join condition only contains attributes of L, the Projection and Theta join operations commute as:

$$\Pi_{L_1 \cup L_2}(R \bowtie_r S) = (\Pi_{L_1}(R)) \bowtie_r (\Pi_{L_2}(S))$$

For example:

$$\Pi_{\text{position, city, branchNo}}(\text{Staff} \bowtie_{\text{Staff.branchNo=Branch.branchNo}} \text{Branch}) =$$

$$(\Pi_{\text{position, branchNo}}(\text{Staff})) \bowtie_{\text{Staff.branchNo=Branch.branchNo}} (\Pi_{\text{city, branchNo}}(\text{Branch}))$$

If the join condition contains additional attributes not in L, say attributes $M = M_1 \cup M_2$ where M_1 involves only attributes of R, and M_2 involves only attributes of S, then a final Projection operation is required:

$$\Pi_{L_1 \cup L_2}(R \bowtie_r S) = \Pi_{L_1 \cup L_2}(\Pi_{L_1 \cup M_1}(R)) \bowtie_r (\Pi_{L_2 \cup M_2}(S))$$

For example:

$$\Pi_{\text{position, city}}(\text{Staff} \bowtie_{\text{Staff.branchNo=Branch.branchNo}} \text{Branch}) =$$

$$\Pi_{\text{position, city}}((\Pi_{\text{position, branchNo}}(\text{Staff})) \bowtie_{\text{Staff.branchNo=Branch.branchNo}} (\Pi_{\text{city, branchNo}}(\text{Branch})))$$

(8) Commutativity of Union and Intersection (but not Set difference).

$$R \cup S = S \cup R$$

$$R \cap S = S \cap R$$

(9) Commutativity of Selection and set operations (Union, Intersection, and Set difference).

$$\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$$

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

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

(10) Commutativity of Projection and Union.

$$\Pi_L(R \cup S) = \Pi_L(S) \cup \Pi_L(R)$$

(11) Associativity of Theta join (and Cartesian product).
Cartesian product and Natural join are always associative:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$(R \times S) \times T = R \times (S \times T)$$

If the join condition q involves only attributes from the relations S and T, then Theta join is associative in the following manner:

$$(R \bowtie_p S) \bowtie_{q \wedge r} T = R \bowtie_{p \wedge r} (S \bowtie_q T)$$

For example:

$$(\text{Staff} \bowtie_{\text{Staff.staffNo=PropertyForRent.staffNo}} \text{PropertyForRent}) \bowtie_{\text{ownerNo=Owner.ownerNo} \wedge \text{Staff.lName=Owner.lName}} \text{Owner}$$

$$= \text{Staff} \bowtie_{\text{Staff.staffNo=PropertyForRent.staffNo} \wedge \text{Staff.lName=lName}} (\text{PropertyForRent} \bowtie_{\text{ownerNo}} \text{Owner})$$

Note that in this example it would be incorrect simply to 'move the brackets' as this would result in an undefined reference (Staff.lName) in the join condition between PropertyForRent and Owner:

$$\text{PropertyForRent} \bowtie_{\text{PropertyForRent.ownerNo=Owner.ownerNo} \wedge \text{Staff.lName=Owner.lName}} \text{Owner}$$

(12) Associativity of Union and Intersection (but not Set difference).

$$(R \cup S) \cup T = S \cup (R \cup T)$$

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

Example 21.3 Use of transformation rules

For prospective renters who are looking for flats, find the properties that match their requirements and are owned by owner CO93.

We can write this query in SQL as:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.prefType = 'Flat' AND c.clientNo = v.clientNo AND
          v.propertyNo = p.propertyNo AND c.maxRent >= p.rent AND
          c.prefType = p.type AND p.ownerNo = 'CO93';
```

For the purposes of this example we will assume that there are fewer properties owned by owner CO93 than prospective renters who have specified a preferred property type of Flat. Converting the SQL to relational algebra, we have:

$$\Pi_{\text{p.propertyNo, p.street}}(\sigma_{\text{c.prefType='Flat'} \wedge \text{c.clientNo=v.clientNo} \wedge \text{v.propertyNo=p.propertyNo} \wedge \text{c.maxRent>=p.rent} \wedge \text{c.prefType=p.type} \wedge \text{p.ownerNo='CO93'}}((\text{c} \times \text{v}) \times \text{p}))$$

We can represent this query as the canonical relational algebra tree shown in Figure 21.4(a). We now use the following transformation rules to improve the efficiency of the execution strategy:

(1) (a) Rule 1, to split the conjunction of Selection operations into individual Selection operations.
    (b) Rule 2 and Rule 6, to reorder the Selection operations and then commute the Selections and Cartesian products.
    The result of these first two steps is shown in Figure 21.4(b).

(2) From Section 4.1.3, we can rewrite a Selection with an Equijoin predicate and a Cartesian product operation, as an Equijoin operation; that is:

$$\sigma_{\text{R.a=S.b}}(\text{R} \times \text{S}) = \text{R} \bowtie_{\text{R.a=S.b}} \text{S}$$

Apply this transformation where appropriate. The result of this step is shown in Figure 21.4(c).

(3) Rule 11, to reorder the Equijoins, so that the more restrictive selection on (p.ownerNo = 'CO93') is performed first, as shown in Figure 21.4(d).

(4) Rules 4 and 7, to move the Projections down past the Equijoins, and create new Projection operations as required. The result of applying these rules is shown in Figure 21.4(e).

An additional optimization in this particular example is to note that the Selection operation (c.prefType=p.type) can be reduced to (p.type = 'Flat'), as we know that (c.prefType='Flat') from the first clause in the predicate. Using this substitution, we push this Selection down the tree, resulting in the final reduced relational algebra tree shown in Figure 21.4(f).

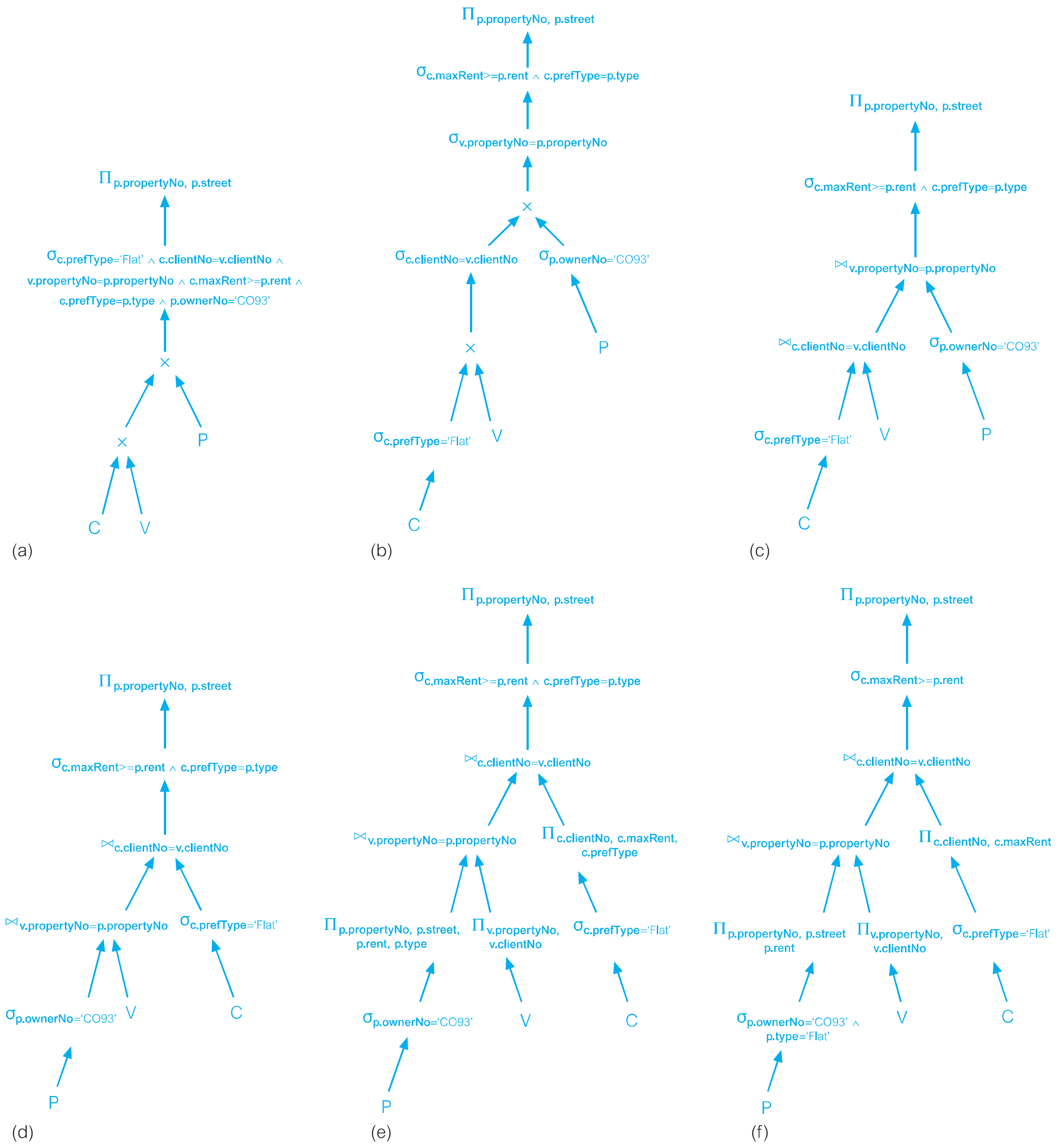
Π p.propertyNo, p.street
σ c.prefType='Flat' ∧ c.clientNo=v.clientNo ∧ v.propertyNo=p.propertyNo ∧ c.maxRent>=p.rent ∧ c.prefType=p.type ∧ p.ownerNo='CO93'
×
×
C
V
P
(a)
Π p.propertyNo, p.street
σ c.maxRent>=p.rent ∧ c.prefType=p.type
σ v.propertyNo=p.propertyNo
×
σ c.clientNo=v.clientNo
σ p.ownerNo='CO93'
×
σ c.prefType='Flat'
V
P
C
(b)
Π p.propertyNo, p.street
σ c.maxRent>=p.rent ∧ c.prefType=p.type
⋈ v.propertyNo=p.propertyNo
⋈ c.clientNo=v.clientNo
σ p.ownerNo='CO93'
σ c.prefType='Flat'
V
P
C
(c)
Π p.propertyNo, p.street
σ c.maxRent>=p.rent ∧ c.prefType=p.type
⋈ c.clientNo=v.clientNo
⋈ v.propertyNo=p.propertyNo
σ c.prefType='Flat'
σ p.ownerNo='CO93'
V
C
P
(d)
Π p.propertyNo, p.street
σ c.maxRent>=p.rent ∧ c.prefType=p.type
⋈ c.clientNo=v.clientNo
⋈ v.propertyNo=p.propertyNo
Π c.clientNo, c.maxRent, c.prefType
Π p.propertyNo, p.street, p.rent, p.type
Π v.propertyNo, v.clientNo
σ c.prefType='Flat'
σ p.ownerNo='CO93'
V
C
P
(e)
Π p.propertyNo, p.street
σ c.maxRent>=p.rent
⋈ c.clientNo=v.clientNo
⋈ v.propertyNo=p.propertyNo
Π c.clientNo, c.maxRent
Π p.propertyNo, p.street p.rent
Π v.propertyNo, v.clientNo
σ c.prefType='Flat'
σ p.ownerNo='CO93' ∧ p.type='Flat'
V
C
P
(f)

Figure 21.4 Relational algebra tree for Example 21.3: (a) canonical relational algebra tree; (b) relational algebra tree formed by pushing Selections down; (c) relational algebra tree formed by changing Selection/Cartesian products to Equijoins; (d) relational algebra tree formed using associativity of Equijoins; (e) relational algebra tree formed by pushing Projections down; (f) final reduced relational algebra tree formed by substituting c.prefType = 'Flat' in Selection on p.type and pushing resulting Selection down tree.

overhead is removed, and there may be more time available to evaluate a larger number of execution strategies, thereby increasing the chances of finding a more optimum strategy. For queries that are executed many times, taking some additional time to find a more optimum plan may prove to be highly beneficial. The disadvantages arise from the fact that the execution strategy that is chosen as being optimal when the query is compiled may no longer be optimal when the query is run. However, a hybrid approach could be used to overcome this disadvantage, where the query is re-optimized if the system detects that the database statistics have changed significantly since the query was last compiled. Alternatively, the system could compile the query for the first execution in each session, and then cache the optimum plan for the remainder of the session, so the cost is spread across the entire DBMS session.

21.2 Query Decomposition

Query decomposition is the first phase of query processing. The aims of query decomposition are to transform a high-level query into a relational algebra query, and to check that the query is syntactically and semantically correct. The typical stages of query decomposition are analysis, normalization, semantic analysis, simplification, and query restructuring.

(1) Analysis

In this stage, the query is lexically and syntactically analyzed using the techniques of programming language compilers (see, for example, Aho and Ullman, 1977). In addition, this stage verifies that the relations and attributes specified in the query are defined in the system catalog. It also verifies that any operations applied to database objects are appropriate for the object type. For example, consider the following query:

```
SELECT staffNumber
FROM Staff
WHERE position > 10;
```

This query would be rejected on two grounds:

(1) In the select list, the attribute staffNumber is not defined for the Staff relation (should be staffNo).

(2) In the WHERE clause, the comparison '>10' is incompatible with the data type position, which is a variable character string.

On completion of this stage, the high-level query has been transformed into some internal representation that is more suitable for processing. The internal form that is typically chosen is some kind of query tree, which is constructed as follows:

- A leaf node is created for each base relation in the query.
- A non-leaf node is created for each intermediate relation produced by a relational algebra operation.
- The root of the tree represents the result of the query.
- The sequence of operations is directed from the leaves to the root.

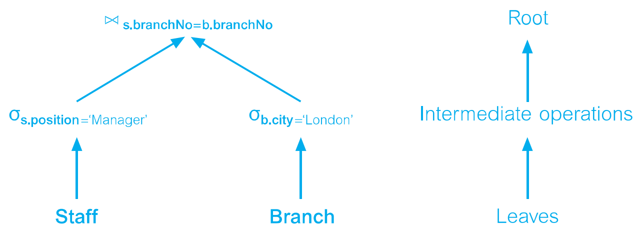

Figure 21.2 Example relational algebra tree.

Figure 21.2 shows an example of a query tree for the SQL statement of Example 21.1 that uses the relational algebra in its internal representation. We refer to this type of query tree as a **relational algebra tree**.

(2) Normalization

The normalization stage of query processing converts the query into a normalized form that can be more easily manipulated. The predicate (in SQL, the WHERE condition), which may be arbitrarily complex, can be converted into one of two forms by applying a few transformation rules (Jarke and Koch, 1984):

- *Conjunctive normal form* A sequence of conjuncts that are connected with the ∧ (AND) operator. Each conjunct contains one or more terms connected by the ∨ (OR) operator. For example:

  $(\text{position} = \text{'Manager'} \vee \text{salary} > 20000) \wedge \text{branchNo} = \text{'B003'}$

  A conjunctive selection contains only those tuples that satisfy all conjuncts.
- *Disjunctive normal form* A sequence of disjuncts that are connected with the ∨ (OR) operator. Each disjunct contains one or more terms connected by the ∧ (AND) operator. For example, we could rewrite the above conjunctive normal form as:

  $(\text{position} = \text{'Manager'} \wedge \text{branchNo} = \text{'B003'}) \vee (\text{salary} > 20000 \wedge \text{branchNo} = \text{'B003'})$

  A disjunctive selection contains those tuples formed by the union of all tuples that satisfy the disjuncts.

(3) Semantic analysis

The objective of semantic analysis is to reject normalized queries that are incorrectly formulated or contradictory. A query is incorrectly formulated if components do not contribute to the generation of the result, which may happen if some join specifications are missing. A query is contradictory if its predicate cannot be satisfied by any tuple. For example, the predicate $(\text{position} = \text{'Manager'} \wedge \text{position} = \text{'Assistant'})$ on the Staff relation is contradictory, as a member of staff cannot be both a Manager and an Assistant simultaneously. However, the predicate $((\text{position} = \text{'Manager'} \wedge \text{position} = \text{'Assistant'}) \vee \text{salary} > 20000)$ could be simplified to $(\text{salary} > 20000)$ by interpreting the contradictory clause

S1: This Selection operation contains an equality condition on the primary key. Therefore, as the attribute staffNo is hashed we can use strategy 3 defined above to estimate the cost as 1 block. The estimated cardinality of the result relation is $SC_{staffNo}(Staff) = 1$.

S2: The attribute in the predicate is a non-key, non-indexed attribute, so we cannot improve on the linear search method, giving an estimated cost of 100 blocks. The estimated cardinality of the result relation is $SC_{position}(Staff) = 300$.

S3: The attribute in the predicate is a foreign key with a clustering index, so we can use Strategy 6 to estimate the cost as $2 + \lceil 6/30 \rceil = 3$ blocks. The estimated cardinality of the result relation is $SC_{branchNo}(Staff) = 6$.

S4: The predicate here involves a range search on the salary attribute, which has a B^+-tree index, so we can use strategy 7 to estimate the cost as: $2 + \lceil 50/2 \rceil + \lceil 3000/2 \rceil = 1527$ blocks. However, this is significantly worse than the linear search strategy, so in this case we would use the linear search method. The estimated cardinality of the result relation is $SC_{salary}(Staff) = \lceil 3000*(50000–20000)/(50000–10000) \rceil = 2250$.

S5: In the last example, we have a composite predicate but the second condition can be implemented using the clustering index on branchNo (S3 above), which we know has an estimated cost of 3 blocks. While we are retrieving each tuple using the clustering index, we can check whether it satisfies the first condition (position = 'Manager'). We know that the estimated cardinality of the second condition is $SC_{branchNo}(Staff) = 6$. If we call this intermediate relation T, then we can estimate the number of distinct values of position in T, $nDistinct_{position}(T)$, as: $\lceil (6 + 10)/3 \rceil = 6$. Applying the second condition now, the estimated cardinality of the result relation is $SC_{position}(T) = 6/6 = 1$, which would be correct if there is one manager for each branch.

21.4.3 Join Operation ($T = (R \bowtie_F S)$)

We mentioned at the start of this chapter that one of the main concerns when the relational model was first launched commercially was the performance of queries. In particular, the operation that gave most concern was the Join operation which, apart from Cartesian product, is the most time-consuming operation to process, and one we have to ensure is performed as efficiently as possible. Recall from Section 4.1.3 that the Theta join operation defines a relation containing tuples that satisfy a specified predicate *F* from the Cartesian product of two relations R and S, say. The predicate *F* is of the form R.a θ S.b, where θ may be one of the logical comparison operators. If the predicate contains only equality (=), the join is an Equijoin. If the join involves all common attributes of R and S, the join is called a Natural join. In this section, we look at the main strategies for implementing the Join operation:

- block nested loop join;
- indexed nested loop join;
- sort–merge join;
- hash join.

Table 21.2 Summary of estimated I/O cost of strategies for Join operation.

Strategies	Cost
Block nested loop join	nBlocks(R) + (nBlocks(R) * nBlocks(S)), if buffer has only one block for R and S
	nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer − 2))], if (nBuffer − 2) blocks for R
	nBlocks(R) + nBlocks(S), if all blocks of R can be read into database buffer
Indexed nested loop join	Depends on indexing method; for example:
	nBlocks(R) + nTuples(R)*(nLevels_A(I) + 1), if join attribute A in S is the primary key
	nBlocks(R) + nTuples(R)*(nLevels_A(I) + [SC_A(R)/bFactor(R)]), for clustering index I on attribute A
Sort–merge join	nBlocks(R)*[$\log_2$(nBlocks(R)] + nBlocks(S)*[$\log_2$(nBlocks(S)], for sorts
	nBlocks(R) + nBlocks(S), for merge
Hash join	3(nBlocks(R) + nBlocks(S)), if hash index is held in memory
	2(nBlocks(R) + nBlocks(S))*[$\log_{\text{nBuffer}-1}$(nBlocks(S)) − 1] + nBlocks(R) + nBlocks(S), otherwise

For the interested reader, a more complete survey of join strategies can be found in Mishra and Eich (1992). The cost estimates for the different Join operation strategies are summarized in Table 21.2. We start by estimating the cardinality of the Join operation.

Estimating the cardinality of the Join operation

The cardinality of the Cartesian product of R and S, $R \times S$, is simply:

nTuples(R) * nTuples(S)

Unfortunately, it is much more difficult to estimate the cardinality of any join as it depends on the distribution of values in the joining attributes. In the worst case, we know that the cardinality of the join cannot be any greater than the cardinality of the Cartesian product, so:

nTuples(T) $\leq$ nTuples(R) * nTuples(S)

Some systems use this upper bound, but this estimate is generally too pessimistic. If we again assume a uniform distribution of values in both relations, we can improve on this estimate for Equijoins with a predicate (R.A = S.B) as follows:

(1) If A is a key attribute of R, then a tuple of S can only join with one tuple of R. Therefore, the cardinality of the Equijoin cannot be any greater than the cardinality of S:

nTuples(T) $\leq$ nTuples(S)

(2) Similarly, if B is a key of S, then:

nTuples(T) $\leq$ nTuples(R)

(3) If neither A nor B are keys, then we could estimate the cardinality of the join as:

$$\text{nTuples}(T) = SC_A(R)*\text{nTuples}(S)$$

or

$$\text{nTuples}(T) = SC_B(S)*\text{nTuples}(R)$$

To obtain the first estimate, we use the fact that for any tuple *s* in S, we would expect on average $SC_A(R)$ tuples with a given value for attribute A, and this number to appear in the join. Multiplying this by the number of tuples in S, we get the first estimate above. Similarly, for the second estimate.

(1) Block nested loop join

The simplest join algorithm is a nested loop that joins the two relations together a tuple at a time. The outer loop iterates over each tuple in one relation R, and the inner loop iterates over each tuple in the second relation S. However, as we know that the basic unit of reading/writing is a disk block, we can improve on the basic algorithm by having two additional loops that process blocks, as indicated in the outline algorithm of Figure 21.8.

Since each block of R has to be read, and each block of S has to be read for each block of R, the estimated cost of this approach is:

$$\text{nBlocks}(R) + (\text{nBlocks}(R) * \text{nBlocks}(S))$$

With this estimate the second term is fixed, but the first term could vary depending on the relation chosen for the outer loop. Clearly, we should choose the relation that occupies the smaller number of blocks for the outer loop.

Figure 21.8
Algorithm for block nested loop join.

```
//
// Block nested loop join
// Blocks in both files are numbered sequentially from 1.
// Returns a result table containing the join of R and S.
//
for iblock = 1 to nBlocks(R) {                          // outer loop
    Rblock = read_block(R, iblock);
    for jblock = 1 to nBlocks(S) {                      // inner loop
        Sblock = read_block(S, jblock);
        for i = 1 to nTuples(Rblock) {
            for j = 1 to nTuples(Sblock) {
                if (Rblock.tuple[i]/Sblock.tuple[j] match join condition)
                then    add them to result;
            }
        }
    }
}
```

Another improvement to this strategy is to read as many blocks as possible of the smaller relation, R say, into the database buffer, saving one block for the inner relation, and one for the result relation. If the buffer can hold nBuffer blocks, then we should read (nBuffer − 2) blocks from R into the buffer at a time, and one block from S. The total number of R blocks accessed is still nBlocks(R), but the total number of S blocks read is reduced to approximately [nBlocks(S)*(nBlocks(R)/(nBuffer − 2))]. With this approach, the new cost estimate becomes:

nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer − 2))]

If we can read all blocks of R into the buffer, this reduces to:

nBlocks(R) + nBlocks(S)

If the join attributes in an Equijoin (or Natural join) form a key on the inner relation, then the inner loop can terminate as soon as the first match is found.

(2) Indexed nested loop join

If there is an index (or hash function) on the join attributes of the inner relation, then we can replace the inefficient file scan with an index lookup. For each tuple in R, we use the index to retrieve the matching tuples of S. The indexed nested loop join algorithm is outlined in Figure 21.9. For clarity, we use a simplified algorithm that processes the outer loop a block at a time. As noted above, however, we should read as many blocks of R into the database buffer as possible. We leave this modification of the algorithm as an exercise for the reader (see Exercise 21.19).

This is a much more efficient algorithm for a join, avoiding the enumeration of the Cartesian product of R and S. The cost of scanning R is nBlocks(R), as before. However,

```
//
// Indexed block loop join of R and S on join attribute A
// Assume that there is an index I on attribute A of relation S, and
// that there are m index entries I[1], I[2], ... , I[m] with indexed value of tuple R[i].A
// Blocks in R are numbered sequentially from 1.
// Returns a result table containing the join of R and S.
//
for iblock = 1 to nBlocks(R) {
    Rblock = read_block(R, iblock);
    for i = 1 to nTuples(Rblock) {
        for j = 1 to m {
            if (Rblock.tuple[i].A = I[j])
            then   add corresponding tuples to result;
        }
    }
}
```

Figure 21.9 Algorithm for indexed nested loop join.

the cost of retrieving the matching tuples in S depends on the type of index and the number of matching tuples. For example, if the join attribute A in S is the primary key, the cost estimate is:

$$\text{nBlocks}(R) + \text{nTuples}(R)*(\text{nLevels}_A(I) + 1)$$

If the join attribute A in S is a clustering index, the cost estimate is:

$$\text{nBlocks}(R) + \text{nTuples}(R)*(\text{nLevels}_A(I) + \lceil SC_A(R)/\text{bFactor}(R)\rceil)$$

(3) Sort–merge join

For Equijoins, the most efficient join is achieved when both relations are sorted on the join attributes. In this case, we can look for qualifying tuples of R and S by merging the two relations. If they are not sorted, a preprocessing step can be carried out to sort them. Since the relations are in sorted order, tuples with the same join attribute value are guaranteed to be in consecutive order. If we assume that the join is many-to-many, that is there can be many tuples of both R and S with the same join value, and if we assume that each set of tuples with the same join value can be held in the database buffer at the same time, then each block of each relation need only be read once. Therefore, the cost estimate for the sort–merge join is:

$$\text{nBlocks}(R) + \text{nBlocks}(S)$$

If a relation has to be sorted, R say, we would have to add the cost of the sort, which we can approximate as:

$$\text{nBlocks}(R)*\lceil \log_2(\text{nBlocks}(R))\rceil$$

An outline algorithm for sort–merge join is shown in Figure 21.10.

(4) Hash join

For a Natural join (or Equijoin), a hash join algorithm may also be used to compute the join of two relations R and S on join attribute set A. The idea behind this algorithm is to partition relations R and S according to some hash function that provides uniformity and randomness. Each equivalent partition for R and S should hold the same value for the join attributes, although it may hold more than one value. Therefore, the algorithm has to check equivalent partitions for the same value. For example, if relation R is partitioned into $R_1, R_2, \ldots, R_M$, and relation S into $S_1, S_2, \ldots, S_M$ using a hash function *h()*, then if B and C are attributes of R and S respectively, and $h(R.B) \neq h(S.C)$, then $R.B \neq S.C$. However, if $h(R.B) = h(S.C)$, it does not necessarily imply that $R.B = S.C$, as different values may map to the same hash value.

The second phase, called the *probing phase*, reads each of the R partitions in turn and for each one attempts to join the tuples in the partition to the tuples in the equivalent S partition. If a nested loop join is used for the second phase, the smaller partition is used

```
//
// Sort-merge join of R and S on join attribute A
// Algorithm assumes join is many-to-many.
// Reads are omitted for simplicity.
// First sort R and S (unnecessary if two files are already sorted on join attributes).
sort(R);
sort(S);
// Now perform merge
nextR = 1; nextS = 1;
while (nextR <= nTuples(R) and nextS <= nTuples(S)) {
    join_value = R.tuples[nextR].A;
// scan S until we find a value less than the current join value
    while (S.tuples[nextS].A < join_value and nextS <= nTuples(S)) {
        nextS = nextS + 1;
    }
// May have matching tuple of R and S.
// For each tuple in S with join_value, match it to each tuple in R with join_value.
// (Assumes M:N join).
    while (S.tuples[nextS].A = join_value and nextS <= nTuples(S)) {
        m = nextR;
        while (R.tuples[m].A = join_value and m <= nTuples(R)) {
            add matching tuples S.tuples[nextS] and R.tuples[m] to result;
            m = m + 1;
        }
        nextS = nextS + 1;
    }
// Have now found all matching tuples in R and S with the same join_value.
// Now find the next tuple in R with a different join value.
    while (R.tuples[nextR].A = join_value and nextR <= nTuples(R)) {
        nextR = nextR + 1;
    }
}
```

Figure 21.10 Algorithm for sort–merge join.

as the outer loop, R_i say. The complete partition R_i is read into memory and each block of the equivalent S_i partition is read and each tuple is used to probe R_i for matching tuples. For increased efficiency, it is common to build an in-memory hash table for each partition R_i using a second hash function, different from the partitioning hash function. The algorithm for hash join is outlined in Figure 21.11. We can estimate the cost of the hash join as:

$$3(\text{nBlocks}(R) + \text{nBlocks}(S))$$

This accounts for having to read R and S to partition them, write each partition to disk, and then having to read each of the partitions of R and S again to find matching tuples. This

Figure 21.11
Algorithm for hash join.

```
//
// Hash join algorithm
// Reads are omitted for simplicity.
//
// Start by partitioning R and S.
for i = 1 to nTuples(R) {
    hash_value = hash_function(R.tuple[i].A);
    add tuple R.tuple[i].A to the R partition corresponding to hash value, hash_value;
}
for j = 1 to nTuples(S) {
    hash_value = hash_function(S.tuple[j].A);
    add tuple S.tuple[j].A to the S partition corresponding to hash value, hash_value;
}
// Now perform probing (matching) phase
for ihash = 1 to M {
    read R partition corresponding to hash value ihash;
    RP = Rpartition[ihash];
    for i = 1 to max_tuples_in_R_partition(RP) {
// build an in-memory hash index using hash_function2( ), different from hash_function( )
        new_hash = hash_function2(RP.tuple[i].A);
        add new_hash to in-memory hash index;
    }
// Scan S partition for matching R tuples
    SP = Spartition[ihash];
    for j = 1 to max_tuples_in_S_partition(SP) {
        read S and probe hash table using hash_function2(SP.tuple[j].A);
        add all matching tuples to output;
    }
    clear hash table to prepare for next partition;
}
```

estimate is approximate and takes no account of overflows occurring in a partition. It also assumes that the hash index can be held in memory. If this is not the case, the partitioning of the relations cannot be done in one pass, and a recursive partitioning algorithm has to be used. In this case, the cost estimate can be shown to be:

$$2(\text{nBlocks}(R) + \text{nBlocks}(S))*[\log_{\text{nBuffer}-1}(\text{nBlocks}(S)) - 1] + \text{nBlocks}(R) + \text{nBlocks}(S)$$

For a more complete discussion of hash join algorithms, the interested reader is referred to Valduriez and Gardarin (1984), DeWitt *et al.* (1984), and DeWitt and Gerber (1985). Extensions, including the hybrid hash join, are described in Shapiro (1986), and a more recent study by Davison and Graefe (1994) describe hash join techniques that can adapt to the available memory.

Example 21.5 Cost estimation for Join operation

For the purposes of this example, we make the following assumptions:

- There are separate hash indexes with no overflow on the primary key attributes staffNo of Staff and branchNo of Branch.
- There are 100 database buffer blocks.
- The system catalog holds the following statistics:

nTuples(Staff)	= 6000			
bFactor(Staff)	= 30	⇒	nBlocks(Staff)	= 200
nTuples(Branch)	= 500			
bFactor(Branch)	= 50	⇒	nBlocks(Branch)	= 10
nTuples(PropertyForRent)	= 100,000			
bFactor(PropertyForRent)	= 50	⇒	nBlocks(PropertyForRent)	= 2000

A comparison of the above four strategies for the following two joins is shown in Table 21.3:

J1: Staff $\bowtie_{\text{staffNo}}$ PropertyForRent
J2: Branch $\bowtie_{\text{branchNo}}$ PropertyForRent

In both cases, we know that the cardinality of the result relation can be no larger than the cardinality of the first relation, as we are joining over the key of the first relation. Note that no one strategy is best for both Join operations. The sort–merge join is best for the first join provided both relations are already sorted. The indexed nested loop join is best for the second join.

Table 21.3 Estimated I/O costs of Join operations in Example 21.5.

Strategies	J1	J2	Comments
Block nested loop join	400,200	20,010	Buffer has only one block for R and S
	4282	N/A[a]	(nBuffer – 2) blocks for R
	N/A[b]	2010	All blocks of R fit in database buffer
Indexed nested loop join	6200	510	Keys hashed
Sort–merge join	25,800	24,240	Unsorted
	2200	2010	Sorted
Hash join	6600	6030	Hash table fits in memory

[a] All blocks of R can be read into buffer.
[b] Cannot read all blocks of R into buffer.