

## 5. Relational Algebra (Part I)

### 5.1. Relational Algebra and Relational Calculus

Many different DMLs can be designed to express database manipulations. Different DMLs will probably differ in syntax, but more importantly they can differ in the basic operations provided. For those familiar with programming, these differences are not unlike that found in different programming languages. There, the basic constructs provided can greatly differ in syntax and in their underlying approach to specifying computations. The latter can be very contrasting indeed—look at, for example, the differences between procedural (eg. C) and declarative (eg. Prolog) approaches to specifying computations.

Relational Algebra and Relational Calculus are two approaches to specifying manipulations on relational databases. The distinction between them is somewhat analagous to that between procedural and declarative programming. Thus, before looking at the details of these approaches, it is instructive to briefly digress and look at procedural and declarative programming a little more closely. We will then try to put into context how the algebra and calculus help in the design and implementation of DMLs.

Briefly, the procedural approach specifies a computation to be performed as *explicit sequences of operations*. The operations themselves are built from a basic set of primitive operations and structuring/control primitives that build higher level operations. An operation can determine the flow of control (ie. determines the next operation to be executed) and/or cause data to be transformed. Thus programming in the procedural style involves working out the operations and their appropriate order of execution to effect the desired transformation(s) on input data.

The declarative approach, in contrast, would specify the same computation as a description of the logical relationship between input and output data. These relationships are typically expressed as a set of truth-valued sentences about data objects. Neither operations nor their sequences are made explicit by the programmer. Operations are instead implicit in the predetermined set of rules of inference used to derive new sentences from those given (or derived earlier).

In other words, if you used a procedural programming language, you must specify *how* input is to be transformed to desired outputs using the basic operations available in the language. However, if you used a declarative language, you must instead describe *what* relationships must be satisfied between inputs and outputs, but essentially say nothing about how a given input is to be transformed to the desired output (it is for the system to determine how, typically by some systematic application of inference rules).

Amidst such differences, users are right to raise some basic concerns. Are these differences superficial? Or do they mean that there can be computations specifiable in one but not in the other? If the latter were the case, programmers must avoid choosing languages that simply cannot express some desired computation (assuming we know exactly the limitations of each and every programming language)! Fortunately for programmers, this is not the case. It is a well-known result that every *general-purpose* programming language (whether procedural or declarative) is **equivalent** to every other. That is, if something is

computable at all, it can be specified in any of these languages. Such equivalence is established by reference to a *fundamental model of computation*<sup>1</sup> that underlies the notion of computability.

Now what can we say about the different (existing and future) database languages? Can two different languages be said to be equivalent in some sense? To answer this question we must first ask whether, in relational databases, there is a fundamental model of database manipulation? The answer is yes—Relational Calculus defines that model!

Let us first state a fundamental definition:

A language that can define any relation definable in relational calculus is *relationally complete*

This definition provides a benchmark by which any existing (and future) language may be judged as to its power of expression. Different languages may thus be equivalent in the sense of being relationally complete<sup>2</sup>.

Relational Calculus, as we shall see later, provides constructs (well-formed expressions) to specify relations. These constructs are very much in a declarative style. Relational Algebra, on the other hand, also provides constructs for relational database manipulations but in a more procedural style. Moreover, it has also been established that Relational Algebra is equivalent to Relational Calculus, in that every expression in one has an equivalent expression in the other. Thus relational completeness of a database language can also be established by showing that it can define any relation expressible in Relational Algebra.

Why then should we bother with different languages and styles of expression if they are all in some sense equivalent? The answer is that besides equivalence (relational or computational), there are other valid issues that different language designs try to address including the level of abstraction, precision, comprehensibility, economy of expression, ease of writing specifications, efficiency of execution, etc. Declarative constructs, by virtue of their descriptive nature (as opposed to the prescriptive nature of procedural constructs), are closer to natural language and thus easier to write and understand. Designers of declarative languages try to provide constructs that even end-users (with a little training) can use to formulate, for example, ad hoc queries. Declarative constructs, however, execute less efficiently than equivalent procedural ones. Attempts to make them more efficient typically involve first, automatic translation to equivalent procedural form, and second, optimising the resulting expressions according to some predetermined rules of optimisation. In fact, this was the original motivation for the Algebra, ie. providing a set of operations to which expressions of the Calculus could be translated and subsequently optimised for execution. But the efficiency of even this approach cannot match that of carefully hand-coded procedural specifications. Thus for periodical and frequently executed manipulations, it is more efficient to use algebraic forms of database languages.

Because of the fundamental nature and role of the Relational Algebra and Calculus in relational databases, we will look at them in depth in the ensuing chapters. This will provide readers with the basic knowledge of database manipulations possible in the model. We begin in this chapter with Relational Algebra.

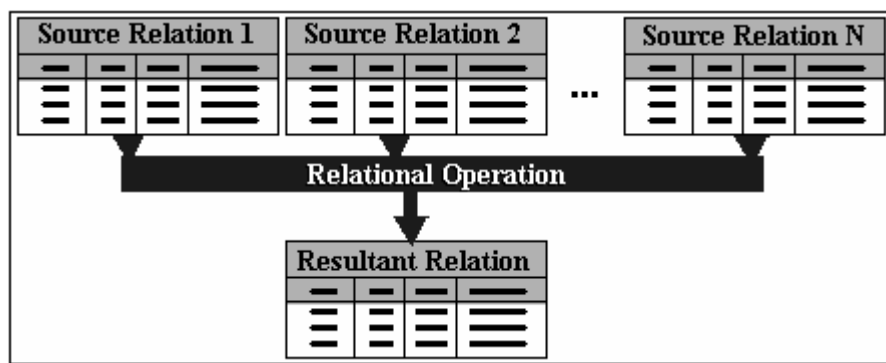
---

<sup>1</sup> The Universal Turing Machine (model of computation) is accepted as defining the class of all computable functions. Every programming language shown to be equivalent to it is therefore equivalent with every other.

<sup>2</sup> Note however, that relational completeness is not the same as computational completeness, ie. Relational Calculus is not equivalent to general-purpose programming languages. It is a specialised calculus intended for the Relational Database Model. Thus while two languages may be relationally complete, each may have features over and above that required for relational completeness (but these need not concern us here).

## 5.2. Overview of Relational Algebra

Relational Algebra comprises a set of basic operations. An operation is the application of an operator to one or more source (or input) relations to produce a new relation as a result. This is illustrated in Figure 5-1 below. More abstractly, we can think of such an operation as a function that maps arguments from specified domains to a result in a specified range. In this case, the domain and range happen to be the same, ie. relations.



**Figure 5-1** A Relational Algebraic Operation

This is no different in principle to, say, operations in arithmetic. For example, the ‘add’ operation in arithmetic takes two numbers and produces a number as a result. In arithmetic, we are used to writing expressions to denote operations and their results. Thus, the ‘add’ operation is usually written using the ‘+’ symbol (the operator) placed between its two arguments, eg.  $145 + 168$ . Moreover, because an expression denotes the result of the operation (which is of the same type as its input arguments), it itself can be written as an argument in another operation, allowing us to construct complex expressions to denote one result, eg.  $145 + 168 - 20 \times 3$ .

Complex expressions that combine different operations are evaluated by a sequence of reductions. The sequence, in the case of arithmetic expressions, is determined by the familiar precedence of operators. Thus, the expression  $145 + 168 - 20 \times 3$  would be reduced as follows:

$$145 + 168 - 20 \times 3 \rightarrow 145 + 168 - 60 \rightarrow 313 - 60 \rightarrow 253$$

This default precedence can be overridden with the explicit use of parenthesis. Thus,

$$(145 + (168 - 20)) \times 3 \rightarrow (145 + 148) \times 3 \rightarrow 393 \times 3 \rightarrow 1179$$

All these would of course be elementary to the reader! The point though is that the basic operations of relational algebra are defined to allow manipulations of relations in much the same way that arithmetic operations manipulate numbers above. With appropriately defined symbols to denote operators and syntax to denote the application of operators to arguments (relations), relational expressions combining multiple operations can be constructed to denote a resultant relation. And as with arithmetic expressions, (algebraic) relational expressions are evaluated by reduction in some specified (default or explicit) order. Thus the earlier statement that relational algebra is basically procedural in nature, ie. operations and their sequencing are explicitly specified to achieve a particular transformation of input(s) to output.

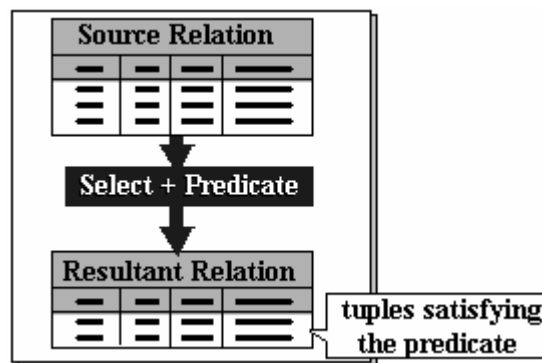
The analogy with arithmetic above has been useful to highlight the basic nature of relational algebra. However, the algebra's basic operations are much more complex than those of arithmetic. They are in fact much more related to operations on sets (viz. intersection, union, difference, etc). This is not surprising as relations are after all special kinds of sets.

As mentioned above, a relational operation maps relations to a relation. As a relation is completely defined by its intension and extension, the complete definition of a relational operation must specify both the schema of the resultant relation and the rules for generating its tuples from the input relation(s). In the following, we will do just that. Moreover, in the interest of clarity as well as precision of presentation, we will define each basic operation both informally and formally.

While notations used to denote the basic operators and operations may differ in the literature, there is no disagreement in their basic logical definitions. It will be necessary for us to use some concrete notation in what follows and, rather than introducing yet more notations, we have chosen in fact to use Codd's original notation.

### 5.3. Selection

Selection is used to extract tuples from a relation. A tuple from the source relation is selected (or not) based on whether it satisfies a specified predicate (condition). A predicate is a truth-valued expression involving tuple component values and their relationships. All tuples satisfying the predicate are then collected into the resultant relation. The general effect is illustrated in Figure 5-2.



**Figure 5-2** The Select Operation

For example, consider the 'Customer' source relation below:

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
2	Martin	Paris	5555910
3	Deen	London	2234391

The result of a selection operation applied to it with the condition that the attribute 'Ccity' must have the value "London" will be:

Result			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
3	Deen	London	2234391

because these are the only tuples that satisfy the stated condition. Procedurally, you may think of the operation as examining each tuple of the source relation in turn (say from top to bottom), checking to see if it met the specified condition before turning attention to the next tuple. If a tuple satisfies the condition, it is included in the resultant relation, otherwise it is ignored.

We shall use the following syntax to express a selection operation:

**select <source-relation-name>**  
**where <predicate>**  
**giving <result-relation-name>**

The <source-relation-name> must already have been defined in the database schema, or is the name of the result of one of previous operations.

In its simplest form, the <predicate> is a simple scalar comparison operation, ie. expressions of the form

**<value<sub>1</sub>> <comparator> <value<sub>2</sub>>**

where <comparator> is one of any comparison operator (ie. =, <, >, ≥, ≤, etc). <value<sub>i</sub>> denotes a scalar value and is either a valid attribute name in the source relation, or a constant value. If an attribute name is specified, it denotes the corresponding value in the tuple under consideration. Thus, the example operation above could have been denoted by the following construct:

**select Customer where Ccity = "London"**

valid relation name                      valid attribute name                      a literal value

Of course, arguments to a comparator must reduce to values from the same domain.

The <predicate> may also take the more complex form of a boolean combination of simple comparison operations above, using the boolean operators 'AND', 'OR', and 'NOT'.

The <result-relation-name> is a unique name that is associated with the result relation. It can be viewed as a convenient abbreviation that can be used as <source-relation-name>s in subsequent operations. Thus, the full selection specification corresponding to the example above is:

**select Customer where Ccity = "London" giving Result**

Note that the intension of the resultant relation is identical to that of the source relation. In other words, the result of selection has the same number of attributes (columns) and attribute names (column labels) as the source relation. Its overall effect, therefore, is to derive a 'horizontal' subset of the source relation.

As another example, consider the following relation. Each tuple represents a sales transaction recording the customer number of the customer purchasing the goods (C#), the product number of the goods sold (P#), the date of the transaction (Date) and the quantity sold (Qnt).

Transaction			
C#	P#	Date	Qnt
1	1	21.01	20
1	2	23.01	30
2	1	26.01	25
2	2	29.01	20

Suppose now that we are interested in looking at only those transactions which took place before February 26 with quantities of more than 25 or involving customer number 2. This need would be translated into the following selection:

**select Transaction**  
**where Date < 26.01 AND Qnt > 25 OR C# = 2**  
**giving Result**

and would yield the relation:

Result			
C#	P#	Date	Qnt
1	2	23.01	30
2	1	26.01	25
2	2	29.01	20

This example illustrates the use of boolean combinations in the <predicate>. However, formulating complex predicates is not as simple and straightforward as it may seem. The basic problem is having to deal with ambiguities at two levels.

First, the informal statement (typically in natural language) of the desired condition may itself be ambiguous. The alert reader would have noted that the phrase (in the above example)

“...only those transactions which took place before February 26 with quantities of more than 25 or involving customer number 2...”

has two possible interpretations:

- a transaction is selected if it is before February 26 and its quantity is more than 25, or it is selected if it involves customer number 2
- all selected transactions must be those that are before February 26 but additionally, each must either involve a quantity of more than 25 or a customer number of 2 (or both)

Such ambiguities must be resolved first before construction of the equivalent boolean expression is attempted. In the above example, the first interpretation was assumed.

Second, the formal boolean expressions involving AND, OR and NOT may also be ambiguous. The source of ambiguity is not unlike that for natural language (ambiguity of strength or order of binding).

Thus

**Cname = "Codd" AND Ccity = "London" OR Ccity = "Paris"**

may be interpreted as

- a) a customer Codd who either lives in London or Paris  
(ie. the OR binds stronger and before AND)
- b) a customer Codd who lives in London, or any customer who lives in Paris (ie. the AND binds stronger and before OR)

Because of its more formal nature, however, such ambiguities are easier to overcome. It can in fact be eliminated through a convention of operator precedences and explicit (syntactical) devices to override default precedences. The conventions used are in fact as follows:

1. expressions enclosed within parentheses have greater precedence (ie. binds stronger). Thus,  
**Cname = "Codd" AND (Ccity = "London" OR Ccity = "Paris")**  
can only take interpretation (a) above
2. The order of precedence of the boolean connectives, unless overridden by parentheses, are (from highest to lowest) NOT, AND, OR. Thus,  
**Cname = "Codd" AND Ccity = "London" OR Ccity = "Paris"**  
can only take interpretation (b) above

There is no limit to the level of 'nesting' of parentheses, ie. a parenthesised expression may have within it a parenthesised subexpression, which may in turn have within it a parenthesised sub-subexpression, etc. Given any boolean expression and the above conventions, we can in fact construct a precedence tree that visually depicts its unique meaning. Figure 5-3 illustrates this. A leaf node represents a basic comparison operation, whereas a non-leaf node represents a boolean combination of its children. A node deeper in the tree binds stronger than those above it. Alternatively, the tree can be viewed as a prescription for reducing (evaluating) a boolean expression to a truth-value (**true** or **false**). To reduce a node:

- a) if it is a leaf node, replace it with the result of its associated comparison operation
- b) if it is a non-leaf node, reduce each of its children; then replace it with the result of applying its associated boolean operation on the truth-values of its children

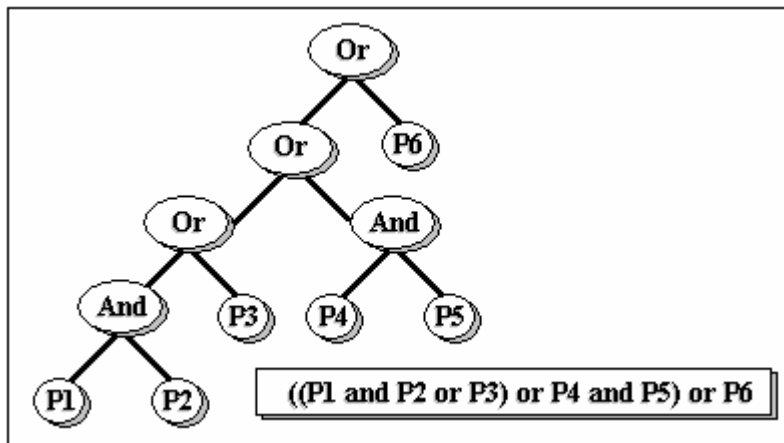


Figure 5-3 A boolean expression and its precedence tree

Using these simple conventions, we can check that expressions we construct indeed carry the intended meanings. (The reader can go back the the last example and ascertain that the intended interpretation was indeed correctly captured in the predicate of the selection statement)

At this point, we should say a few words about the notation, particularly in the context of the analogy to arithmetic expressions in the last section. Strictly speaking, the full selection syntax above is not an expression that can be used as an argument to another operation. This does not contradict the analogy, however. The selection syntax, in fact, has an expression component comprising the select- and where-clauses only, ie. without the giving-clause:

select <source-relation-name> where <predicate> giving <result-relation-name>

Expression

Thus,

select Customer where Ccity = "London"

is an expression that completely specifies a selection operation while denoting also its result, in much the same way that '2+3' completely specifies an addition operation while also denoting the resultant number (ie. 5). The expression, therefore, can syntactically occur where a relation is expected and it would then be valid to write:

select (select Customer where Ccity = "London" ) where C# < 3

Strictly speaking, this is all we need to define the selection operation. So, of what use is the giving-clause? The answer, in fact, was alluded to earlier when we described the clause as allowing us to introduce a convenient *abbreviation*. It is convenient, and useful, especially in simplifying and making more readable what may otherwise be unwieldy and confusing expressions. Even the simple double selection expression above may already look unwieldy to the reader (imagine what the expression would look like if it involved 10 algebraic operations, say!). It would be clearer to write:

select Customer where Ccity = "London" giving Result;  
select Result where C# < 3 giving Result2



(of course, this operation could have been more simply and clearly written as a single selection operation involving a boolean combination of comparison operations; the reader should attempt to write it as an exercise)

Mathematical notation too have various devices to introduce abbreviations to simplify and make expressions more readable. What we are doing here with the giving-clause is analogous to, for example, writing:

```
let x = 2+3
let y = 7-2
let z = (x-y) × (x+y)
```

instead of the unabbreviated “ $((2+3)-(7-2)) \times ((2+3)+(7-2))$ ”. The giving-clause is thus mainly a stylistic device. It is important to note that that is all it is - introducing a *temporary* abbreviation to be used in another operation. In particular, it is not to be interpreted as permanently modifying the database schema with the addition of a new relation name.

In this book, we will favour this notational style because we think it leads to a simpler and more comprehensible notation. The reader should note, however, that while other descriptions in the literature may favour and use only the expression forms, the differences are superficial.

### Formal Definition

If  $\sigma$  denotes a relation, then let

$S(\sigma)$  denote the finite set of attribute names of  $\sigma$  (ie. its intension)  
 $T(\sigma)$  denote the finite set of tuples of  $\sigma$  (ie. its extension)  
 $\text{dom}(\alpha)$ , where  $\alpha \in S(\sigma)$ , denote the set of allowable values for  $\alpha$   
 $\tau \bullet \alpha$ , where  $\tau \in T(\sigma)$  and  $\alpha \in S(\sigma)$ , denote the value of attribute  $\alpha$  in tuple  $\tau$

The **selection** operation takes the form

select  $\sigma$  where  $\pi$  giving  $\rho$

where  $\pi$  is a predicate expression. The syntax of a predicate expression is given by the following BNF grammar (this should be viewed as an abstract syntax not necessarily intended for an end-user language):

```
pred_exp ::= comp_exp | bool_exp | ( pred_exp )
bool_exp ::= negated_exp | binary_exp
negated_exp ::= NOT pred_exp
binary_exp ::= pred_exp bool_op pred_exp
bool_op ::= AND | OR
comp_exp ::= argument comparator argument
comparator ::= > | < | ≥ | ≤ | =
argument3 ::= attribute_name | literal
```

---

<sup>3</sup> The syntax of ‘attribute\_name’ and ‘literal’ are unimportant in what follows and we leave it unspecified

$\pi$  is well-formed iff it is syntactically correct and

- for every attribute\_name  $\alpha$  in  $\pi$ ,  $\alpha \in S(\sigma)$
- for every comp\_expr  $\alpha_1 * \alpha_2$  (where '\*' denotes a comparator) such that  $\alpha_1, \alpha_2 \in S(\sigma)$ , either  $\text{dom}(\alpha_1) \subseteq \text{dom}(\alpha_2)$  or  $\text{dom}(\alpha_2) \subseteq \text{dom}(\alpha_1)$
- for every comp\_expr  $\alpha * \kappa$ , or  $\kappa * \alpha$  (where '\*' denotes a comparator) such that  $\alpha \in S(\sigma)$  and  $\kappa$  is a literal,  $\kappa \in \text{dom}(\alpha)$

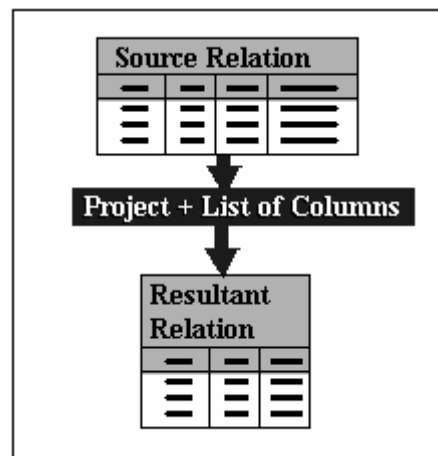
Further, let  $\pi(\tau)$  denote the application of a well-formed predicate expression  $\pi$  to a tuple  $\tau \in T(\sigma)$ .  $\pi(\tau)$  reduces  $\pi$  in the context of  $\tau$ , ie. the occurrence of any  $\alpha \in S(\sigma)$  in  $\pi$  is first replaced by  $\tau \bullet \alpha$ . The resulting expression is then reduced to a truth-value according to the accepted semantics of comparators and boolean operators.

Then  $\rho$ , the resultant relation of the selection operation, is characterised by the following:

- $S(\rho) \equiv S(\sigma)$
- $T(\rho) \equiv \{ \tau \mid \tau \in T(\sigma) \wedge \pi(\tau) \}$

## 5.4.Projection

Whereas a selection operation extracts rows of a relation meeting specified conditions, a projection operation extracts specified *columns* of a relation. The desired columns are simply specified by name. The general effect is illustrated in Figure 5-4.



**Figure 5-4:** The projection operation

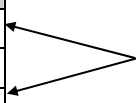
We could think of selection as eliminating rows (tuples) not meeting the specified conditions. In like manner, we can think of a projection as eliminating columns not named in the operation. However, an additional step is required for projection because removing columns may result in duplicate rows, which are not allowed in relations. Quite simply, any duplicate occurrence of a row must be removed so that the result is a relation (a desired property of relational algebra operators).

For example, using again the customer relation:

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
2	Martin	Paris	5555910
3	Deen	London	2234391

its projection over the attribute 'Ccity' would yield (after eliminating all columns other than 'Ccity'):

Result	Ccity
London	
Paris	
London	



duplicates

Note the duplication of row 1 in row 3. Projection can result in duplication because the resultant tuples have a smaller degree whereas the uniqueness of tuples in the source relation is only guaranteed for the original degree of the relation. For the final result to be a relation, duplicated occurrences must be removed, ie.

Result	Ccity
London	
Paris	

The form of a projection operation is:

**project** <source-relation-name>  
**over** <list-of-attribute-names>  
**giving** <result-relation-name>

Thus the above operation would be written as:


**project** Customer  
**over** Ccity  
**giving** Result

As with selection, <source-relation-name> must be a valid relation—a relation name defined in the database schema or the name of the result of a previous operation. <list-of-attribute-names> is a comma-separated list of at least one identifier. Each identifier appearing in the list must be a valid attribute name of <source-relation-name>. And finally, <result-relation-name> must be a unique identifier used to name the resultant relation.

Why would we want to project a relation over some attributes and not others? Quite simply, we sometimes are interested in only a subset of an entity's attributes given a particular situation. Thus, if we needed to

telephone all customers to inform them of some new product line, data about a customer's number and the city of residence are superfluous. The relevant data, and only the relevant data, can be presented using:

**project Customer**  
**over Cname, Cphone**  
**giving Result**



Result	
Cname	Cphone
Codd	2263035
Martin	5555910
Deen	2234391

Extending this example, suppose further that we have multiple offices sited in major cities and the task of calling customers is distributed amongst such offices, ie. the office in London will call up customers resident in London, etc. Now the simple projection above will not do, because it presents customer names and phone numbers without regard to their place of residence. If it was used by each office, customers will receive multiple calls and you will probably have many annoyed customers on your hands, not to mention the huge phone bills you unnecessarily incurred!

The desired relation in this case must be restricted to only customers from a given city. How can we specify this? The simple answer is that we cannot - not with just the projection operation. However, the alert reader would have realised that the requirement to restrict resultant rows to only those from a given city is exactly the sort of requirement that the selection operation is designed for! In other words, here we have an example of a situation that needs a composition of operations to compute the desired relation. Thus, for the office in London, the list of customers and phone numbers relevant to it is computed by first selecting customers from London, then projecting the result over customer names and phone numbers. This is illustrated in Figure 5-5. For offices in other cities, only the predicate of the selection needs to be appropriately modified.

Note that the order of the operations is significant, ie. a selection followed by a projection. It would not work the other way around (you can verify this by trying it out yourself).

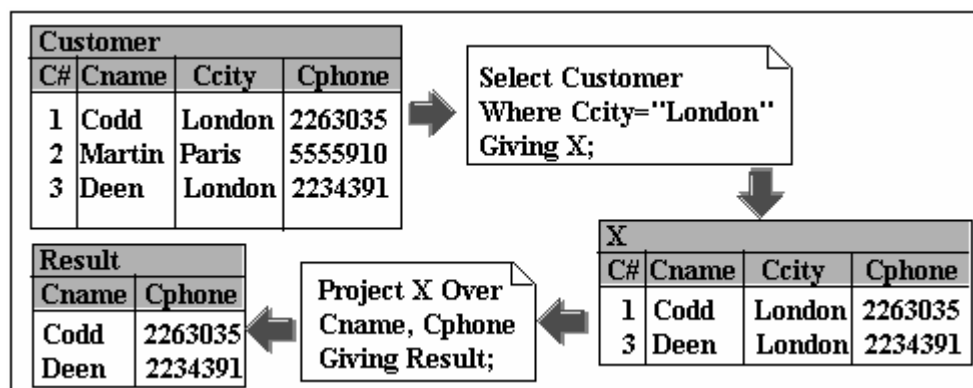


Figure 5-5 Combining operators to compute a desired relation

## Formal Definition

If  $\sigma$  denotes a relation, then let

$S(\sigma)$  denote the finite set of attribute names of  $\sigma$  (ie. its intension)

$T(\sigma)$  denote the finite set of tuples of  $\sigma$  (ie. its extension)

$\tau \bullet \alpha$ , where  $\tau \in T(\sigma)$  and  $\alpha \in S(\sigma)$ , denote the value of attribute  $\alpha$  in tuple  $\tau$

The projection operation takes the form

**project  $\sigma$  over  $\delta$  giving  $\rho$**

where  $\delta$  is a comma-separated list of attribute names. Formally,  $\delta$  (as a discrete structure) may be considered a tuple, but having a concrete enumeration syntax (comma-separated list).

Let  $S_{\text{tuple}}(x)$  denote the set of elements in the tuple  $x$ . Then,  $\delta$  must observe the following constraint:

$$S_{\text{tuple}}(\delta) \subset S(\sigma)$$

ie. every name occurring in  $\delta$  must be a valid attribute name in the relation  $\sigma$ .

Furthermore, if  $\tau \in T(\sigma)$  and  $\tau'$  denotes a tuple, we define:

$$R(\tau, \delta, \tau') \equiv \forall \alpha \bullet \alpha \in S_{\text{tuple}}(\delta) \Leftrightarrow \tau \bullet \alpha \in S_{\text{tuple}}(\tau')$$

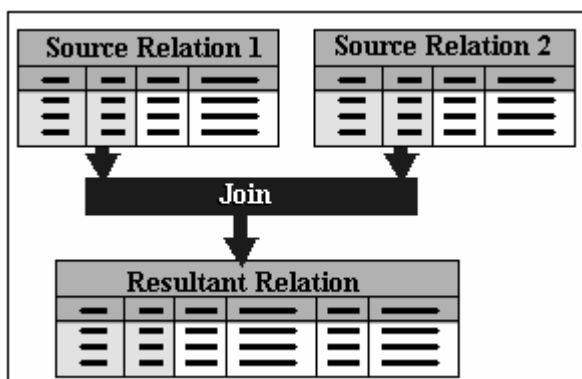
ie. a tuple element  $\tau \bullet \alpha$  is in the tuple  $\tau'$  if and only if the attribute name  $\alpha$  occurs in  $\delta$ .

Then  $\rho$ , the resultant relation of the projection, is characterised by the following:

- $S(\rho) \equiv S_{\text{tuple}}(\delta)$
- $T(\rho) \equiv \{ \tau' \mid \tau \in T(\sigma) \wedge R(\tau, \delta, \tau') \}$

## 5.5.Natural Join

The next operation we will look at is the Natural Join (hereafter referred to simply as Join). This operation takes two source relations as inputs and produces a relation whose tuples are formed by concatenating tuples from each input source. It is basically a cartesian product of the extensions of each input source. However, not all possible combinations of tuples necessarily end up in the result. This is because it implicitly selects from among all possible tuple combinations only those that have identical values in attributes *shared* by both relations.

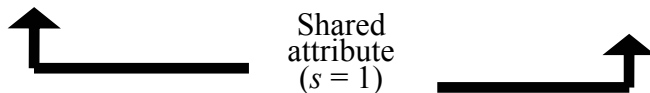


**Figure 5-6** The Join combines two relations over one or more common domains

Thus, in a typical application of a Join, the intensions of the input sources share at least one attribute name or domain (we assume here that attribute names are global to a schema, ie. the same name occurring in different relations denote the same attribute and value domain). The Join is said to occur over such domain(s). Figure 5-6 illustrates the general effect. The shaded left-most two columns of the inputs are notionally the shared attributes. The result comprise these and the concatenation of the other columns from each input. More precisely, if the degree of the input sources were  $m$  and  $n$  respectively, and the number of shared attributes was  $s$ , then the degree of the resultant relation is  $(m+n-s)$ .

As an example, consider the two relations below:

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
2	Martin	Paris	5555910
3	Deen	London	2234391



Transaction			
C#	P#	Date	Qnt
1	1	21.01	20
1	2	23.01	30
2	1	26.01	25
2	2	29.01	20

These relations share the attribute 'C#', as indicated. To compute the join of these relations, consider in turn every possible pair of tuples formed by taking one tuple from each relation, and examine the values of their shared attribute. So if the pair under consideration was

$\langle 1, \text{Codd}, \text{London}, 2263035 \rangle$  and  $\langle 1, 1, 21.01, 20 \rangle$

we would find that the values match exactly. In such a case, we concatenate them and add the concatenation to the resultant relation. It doesn't matter if the second tuple is concatenated to the end of the first, or the first to the second, as long as we are consistent about it. By convention, we use the former. Additionally, we omit the second occurrence of the shared attribute in the result (repeated occurrence is superfluous). This gives us the tuple

$\langle 1, \text{Codd}, \text{London}, 2263035, 1, 21.01, 20 \rangle$

If, on the other hand, the pair under consideration was

$\langle 3, \text{Deen}, \text{London}, 2234391 \rangle$  and  $\langle 1, 1, 21.01, 20 \rangle$

we would ignore it because the values of their shared attributes do not match exactly.

Thus, the resultant relation after considering all pairs would be:

Result						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
2	Martin	Paris	5555910	2	29.01	20

The foregoing description is in fact general enough to admit operations on relations that do not share any attributes at all ( $s = 0$ ). The join, in such a case, is simply the cartesian product of the input sources' extensions (the condition that tuple combinations have identical values over shared attributes is vacuously true since there are no shared attributes!). However, such uses of the operation are atypical.

Syntactically, we will write the Join operation as follows:

**join** <source-relation-name><sub>1</sub> AND <source-relation-name><sub>2</sub>  
**over** <attribute-name-list>  
**giving** <result-relation-name>

where again

- <source-relation-name><sub>1</sub> is a valid relation name (in the schema or the result of a previous operation)
- <attribute-name-list> is a comma-separated non-empty list of attribute names, each of which must occur in both input sources, and
- <result-relation-name> is a unique identifier denoting the resultant relation

With this syntax, particularly with the over-clause, we have in fact taken the liberty

- (1) to insist that the join must be over at least one shared attribute, ie. we disallow expressions of pure cartesian products of two relations that do not share any attribute. This restriction is of no *practical* consequence, however, as in practice a Join is used to bring together information from different relations related through some common value.
- (2) to allow a join over a subset of shared attributes, ie. we relax (generalise) the restriction that a Join is over all shared attributes.

If a Join is over a proper subset of shared attributes, then shared attributes not specified in the over-clause will each have its own column in the result relation. But in such cases, the respective column labels will be qualified names. We will adopt the convention of writing a qualified name as ' $\rho.\alpha$ ', where  $\alpha$  is the column label and  $\rho$  the relation name in which  $\alpha$  appears. As an illustration, consider the relations below:

R1		
A1	A2	X
1	2	abc
1	3	def
2	4	ijk

R2		
A1	A2	Y
2	3	pqr
2	2	xyz

The operation

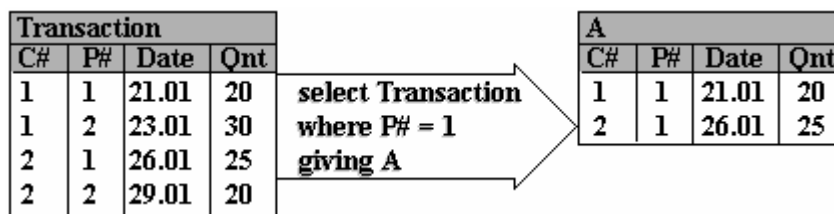
**join R1 AND R2 over A1 giving Result**

will yield

Result				
A1	R1.A2	X	R2.A2	Y
2	4	ijk	3	pqr
2	4	ijk	2	xyz

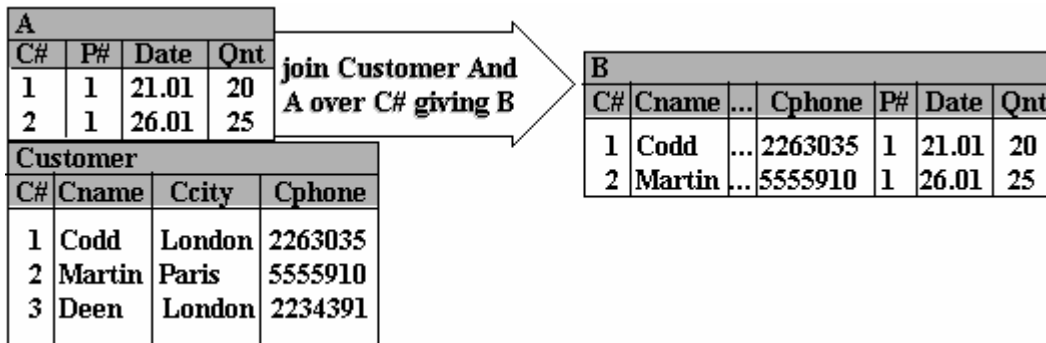
To see why Join is a necessary operation in the algebra, consider the following situation (assume as context the Customer and Transaction relations above): the company decided that customers who purchased product number 1 ( $P\# = 1$ ) should be informed that a fault has been discovered in the product and that, as a sign of good faith and of how it values its customers, it will replace the product with a brand new fault-free one. To do this, we need to list, therefore, the names and phone numbers of all such customers.

First, we need to identify all customers who purchased product number 1. This information is in the Transaction relation and, using the following selection operation, it is easy to limit its extension to only such customers:

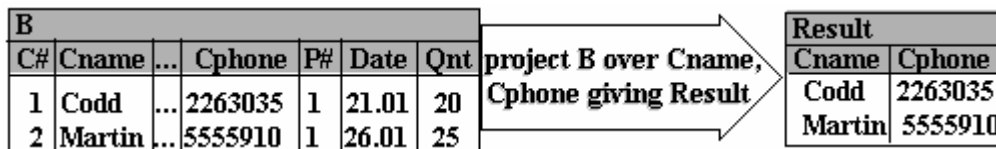


Next, we note that the resultant relation only identifies such customers by their customer numbers. What we need, though, are their names and phone numbers. In other words, we would like to extend each tuple in A with the customer name and phone number corresponding to the customer number. As such items are found in the relation Customer which shares the attribute 'C#' with A, the join is a natural operation to perform:





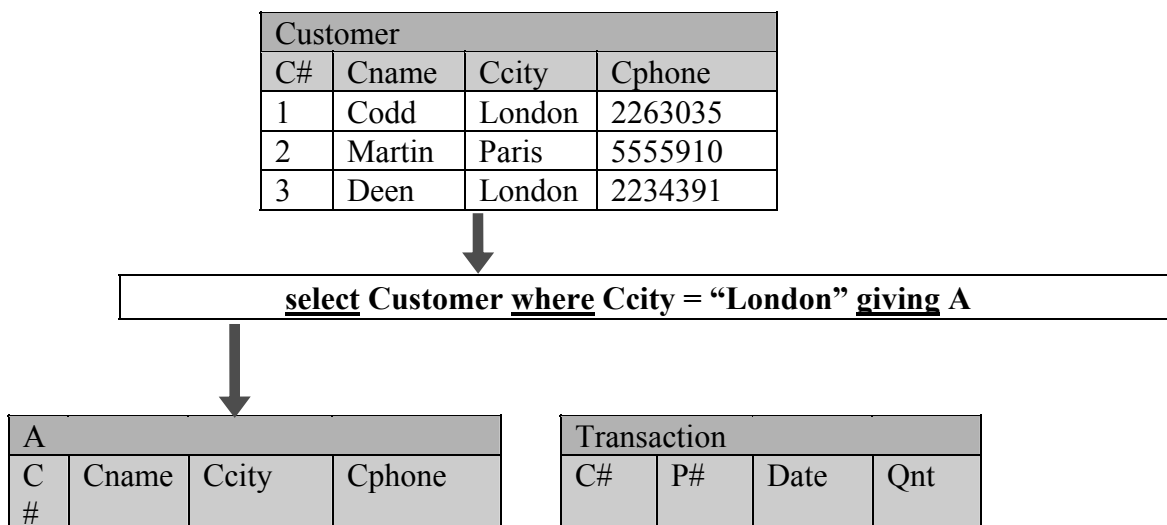
With B, we have practically derived the information we need—in fact, more than we need, since we are interested only in the customer name (the ‘Cname’ column) and phone number (the ‘Cphone’ column). But as we’ve learned, the irrelevant columns may be easily removed using projection, as shown below.



As a final example, let us also assume we have the Product relation, in addition to the Customer and Transaction relations:

Product		
P#	Pname	Pprice
1	CPU	1000
2	VDU	1200

The task is to “get the names of products sold to customers in London”. Once again, this task will require a combination of operations which must involve a Join at some point because not all the information required are contained in one relation. The sequence of operations required is shown below.



1	Codd	London	2263035
3	Deen	London	2234391

1	1	21.01	20
1	2	23.01	30
2	1	26.01	25
2	2	29.01	20

**join A AND Transaction over C# giving B**

B						
C#	Cname	Ccity	...	P#	Date	Qnt
1	Codd	London	...	1	21.01	20
1	Codd	London	...	2	23.01	30

Product		
P#	Pname	Pprice
1	CPU	1000
2	VDU	1200

**join B AND Product over P# giving C**

C								
C #	Cname	Ccity	Cphone	P#	Date	Qnt	Pname	Pprice
1	Codd	London	2263035	1	21.01	20	CPU	1000
1	Codd	London	2263035	2	23.01	30	VDU	1200

**project C over Pname giving Result**

Result
Pname
CPU
VDU

## Formal Definition

As before, if  $\sigma$  denotes a relation, then let

$S(\sigma)$  denote the finite set of attribute names of  $\sigma$  (ie. its intension)

$T(\sigma)$  denote the finite set of tuples of  $\sigma$  (ie. its extension)

$\tau \bullet \alpha$ , where  $\tau \in T(\sigma)$  and  $\alpha \in S(\sigma)$ , denote the value of attribute  $\alpha$  in tuple  $\tau$

Further, if  $\tau_1$  and  $\tau_2$  are tuples, let  $\tau_1 \wedge \tau_2$  denote the tuple resulting from appending  $\tau_2$  to the end of  $\tau_1$ .

We will also have need to use the terminology introduced in defining projection above, in particular,  $S_{\text{tuple}}$  and the definition:

$$R(\tau, \delta, \tau') \equiv \forall \alpha \bullet \alpha \in S_{\text{tuple}}(\delta) \Leftrightarrow \tau \bullet \alpha \in S_{\text{tuple}}(\tau')$$

The (natural) join operation takes the form

**join  $\sigma$  AND  $\nu$  over  $\delta$  giving  $\rho$**

As with other operations, the input sources  $\sigma$  and  $\nu$  must denote valid relations that are either defined in the schema or are results of previous operations, and  $\rho$  must be a unique identifier to denote the result of the join.  $\delta$  is a tuple of attribute names such that:

$$S_{\text{tuple}}(\delta) \subseteq (S(\sigma) \cap S(\nu))$$

Let  $\varepsilon = (S(\sigma) \cap S(\nu)) - S_{\text{tuple}}(\delta)$ , ie. the set of shared attribute names not specified in the over-clause. We next define, for any relation  $r$ :

$$\text{Rename}(r, \varepsilon) \equiv \{ \alpha \mid \alpha \in S(r) - \varepsilon \vee (\alpha = 'r.p' \wedge p \in S(r) \cap \varepsilon) \}$$

In the case that  $\varepsilon = \{\}$  or  $S(r) \cap \varepsilon = \{\}$ ,  $\text{Rename}(r, \varepsilon) = S(r)$ .

The Join operation can then be characterised by the following:

- $S(\rho) \equiv \text{Rename}(\sigma, \varepsilon) \cup \text{Rename}(\nu, \varepsilon)$
- $T(\rho) \equiv \{ \tau_1 \wedge \tau_2 \mid \tau_1 \in T(\sigma) \wedge \tau_2 \in T(\nu) \wedge R(\tau_1, \Delta, \tau_2) \wedge \forall \alpha \bullet \alpha \in S_{\text{tuple}}(\delta) \Rightarrow \tau_1 \bullet \alpha = \tau_2 \bullet \alpha \}$

where

$$S_{\text{tuple}}(\Delta) = S(\nu) - S_{\text{tuple}}(\delta)$$