2.4 An Algebraic Query Language

In this section, we introduce the data-manipulation aspect of the relational model. Recall that a data model is not just structure; it needs a way to query the data and to modify the data. To begin our study of operations on relations, we shall learn about a special algebra, called *relational algebra*, that consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

Relational algebra is not used today as a query language in commercial DBMS's, although some of the early prototypes did use this algebra directly. Rather, the "real" query language, SQL, incorporates relational algebra at its center, and many SQL programs are really "syntactically sugared" expressions of relational algebra. Further, when a DBMS processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation. Thus, there are several good reasons to start out learning this algebra.

2.4.1 Why Do We Need a Special Query Language?

Before introducing the operations of relational algebra, one should ask why, or whether, we need a new kind of programming languages for databases. Won't conventional languages like C or Java suffice to ask and answer any computable question about relations? After all, we can represent a tuple of a relation by a struct (in C) or an object (in Java), and we can represent relations by arrays of these elements.

The surprising answer is that relational algebra is useful because it is *less* powerful than C or Java. That is, there are computations one can perform in any conventional language that one cannot perform in relational algebra. An example is: determine whether the number of tuples in a relation is even or odd. By limiting what we can say or do in our query language, we get two huge rewards — ease of programming and the ability of the compiler to produce highly optimized code — that we discussed in Section 2.1.6.

2.4.2 What is an Algebra?

An algebra, in general, consists of operators and atomic operands. For instance, in the algebra of arithmetic, the atomic operands are variables like x and constants like 15. The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division. Any algebra allows us to build *expressions* by applying operators to atomic operands and/or other expressions of the algebra. Usually, parentheses are needed to group operators and their operands. For instance, in arithmetic we have expressions such as $(x + y) * z$ or $((x + 7)/(y - 3)) + x$.

Relational algebra is another example of an algebra. Its atomic operands are:

1. Variables that stand for relations.
2. Constants, which are finite relations.

We shall next see the operators of relational algebra.

2.4.3 Overview of Relational Algebra

The operations of the traditional relational algebra fall into four broad classes:

a) The usual set operations — union, intersection, and difference — applied to relations.

b) Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.

c) Operations that combine the tuples of two relations, including "Cartesian product," which pairs the tuples of two relations in all possible ways, and various kinds of "join" operations, which selectively pair tuples from two relations.

d) An operation called "renaming" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

We generally shall refer to expressions of relational algebra as *queries*.

2.4.4 Set Operations on Relations

The three most common operations on sets are union, intersection, and difference. We assume the reader is familiar with these operations, which are defined as follows on arbitrary sets R and S:

- $R \cup S$, the *union* of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.
- $R \cap S$, the *intersection* of R and S, is the set of elements that are in both R and S.
- $R - S$, the *difference* of R and S, is the set of elements that are in R but not in S. Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R.

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.

2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Sometimes we would like to take the union, intersection, or difference of relations that have the same number of attributes, with corresponding domains, but that use different names for their attributes. If so, we may use the renaming operator to be discussed in Section 2.4.11 to change the schema of one or both relations and give them the same set of attributes.

name	*address*	*gender*	*birthdate*
`Carrie Fisher`	`123 Maple St., Hollywood`	`F`	`9/9/99`
`Mark Hamill`	`456 Oak Rd., Brentwood`	`M`	`8/8/88`

Relation R

name	*address*	*gender*	*birthdate*
`Carrie Fisher`	`123 Maple St., Hollywood`	`F`	`9/9/99`
`Harrison Ford`	`789 Palm Dr., Beverly Hills`	`M`	`7/7/77`

Relation S

Figure 2.12: Two relations

Example 2.8: Suppose we have the two relations R and S, whose schemas are both that of relation `MovieStar` Section 2.2.8. Current instances of R and S are shown in Fig. 2.12. Then the union $R \cup S$ is

name	*address*	*gender*	*birthdate*
`Carrie Fisher`	`123 Maple St., Hollywood`	`F`	`9/9/99`
`Mark Hamill`	`456 Oak Rd., Brentwood`	`M`	`8/8/88`
`Harrison Ford`	`789 Palm Dr., Beverly Hills`	`M`	`7/7/77`

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

The intersection $R \cap S$ is

name	*address*	*gender*	*birthdate*
`Carrie Fisher`	`123 Maple St., Hollywood`	`F`	`9/9/99`

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. □

2.4.5 Projection

The *projection* operator is used to produce from a relation R a new relation that has only some of R's columns. The value of expression $\pi_{A_1,A_2,\ldots,A_n}(R)$ is a relation that has only the columns for attributes $A_1, A_2, \ldots, A_n$ of R. The schema for the resulting value is the set of attributes $\{A_1, A_2, \ldots, A_n\}$, which we conventionally show in the order listed.

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890
Wayne's World	1992	95	comedy	Paramount	99999

Figure 2.13: The relation Movies

Example 2.9: Consider the relation Movies with the relation schema described in Section 2.2.8. An instance of this relation is shown in Fig. 2.13. We can project this relation onto the first three attributes with the expression:

$$\pi_{title,year,length}(\texttt{Movies})$$

The resulting relation is

title	*year*	*length*
Star Wars	1977	124
Galaxy Quest	1999	104
Wayne's World	1992	95

As another example, we can project onto the attribute genre with the expression $\pi_{genre}(\texttt{Movies})$. The result is the single-column relation

genre
sciFi
comedy

Notice that there are only two tuples in the resulting relation, since the last two tuples of Fig. 2.13 have the same value in their component for attribute genre, and in the relational algebra of sets, duplicate tuples are always eliminated. □

A Note About Data Quality :-)

While we have endeavored to make example data as accurate as possible, we have used bogus values for addresses and other personal information about movie stars, in order to protect the privacy of members of the acting profession, many of whom are shy individuals who shun publicity.

2.4.6 Selection

The *selection* operator, applied to a relation R, produces a new relation with a subset of R's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R. We denote this operation $\sigma_C(R)$. The schema for the resulting relation is the same as R's schema, and we conventionally show the attributes in the same order as we use for R.

C is a conditional expression of the type with which we are familiar from conventional programming languages; for example, conditional expressions follow the keyword `if` in programming languages such as C or Java. The only difference is that the operands in condition C are either constants or attributes of R. We apply C to each tuple t of R by substituting, for each attribute A appearing in condition C, the component of t for attribute A. If after substituting for each attribute of C the condition C is true, then t is one of the tuples that appear in the result of $\sigma_C(R)$; otherwise t is not in the result.

Example 2.10: Let the relation `Movies` be as in Fig. 2.13. Then the value of expression $\sigma_{length\geq 100}$(`Movies`) is

title	*year*	*length*	*genre*	*studioName*	*producerC#*
`Star Wars`	`1977`	`124`	`sciFi`	`Fox`	`12345`
`Galaxy Quest`	`1999`	`104`	`comedy`	`DreamWorks`	`67890`

The first tuple satisfies the condition $length \geq 100$ because when we substitute for *length* the value 124 found in the component of the first tuple for attribute `length`, the condition becomes $124 \geq 100$. The latter condition is true, so we accept the first tuple. The same argument explains why the second tuple of Fig. 2.13 is in the result.

The third tuple has a `length` component 95. Thus, when we substitute for *length* we get the condition $95 \geq 100$, which is false. Hence the last tuple of Fig. 2.13 is not in the result. $\square$

Example 2.11: Suppose we want the set of tuples in the relation `Movies` that represent Fox movies at least 100 minutes long. We can get these tuples with a more complicated condition, involving the `AND` of two subconditions. The expression is

$$\sigma_{length\geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})$$

The tuple

title	*year*	*length*	*genre*	*studioName*	*producerC#*
`Star Wars`	`1977`	`124`	`sciFi`	`Fox`	`12345`

is the only one in the resulting relation. □

2.4.7 Cartesian Product

The *Cartesian product* (or *cross-product*, or just *product*) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S. This product is denoted $R \times S$. When R and S are relations, the product is essentially the same. However, since the members of R and S are tuples, usually consisting of more than one component, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples. By convention, the components from R (the left operand) precede the components from S in the attribute order for the result.

The relation schema for the resulting relation is the union of the schemas for R and S. However, if R and S should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes. To disambiguate an attribute A that is in the schemas of both R and S, we use $R.A$ for the attribute from R and $S.A$ for the attribute from S.

Example 2.12: For conciseness, let us use an abstract example that illustrates the product operation. Let relations R and S have the schemas and tuples shown in Fig. 2.14(a) and (b). Then the product $R \times S$ consists of the six tuples shown in Fig. 2.14(c). Note how we have paired each of the two tuples of R with each of the three tuples of S. Since B is an attribute of both schemas, we have used $R.B$ and $S.B$ in the schema for $R \times S$. The other attributes are unambiguous, and their names appear in the resulting schema unchanged. □

2.4.8 Natural Joins

More often than we want to take the product of two relations, we find a need to *join* them by pairing only those tuples that match in some way. The simplest sort of match is the *natural join* of two relations R and S, denoted $R \bowtie S$, in which we pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S. More precisely, let $A_1, A_2, \ldots, A_n$ be all the attributes that are in both the schema of R and the schema of S. Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, \ldots, A_n$.

If the tuples r and s are successfully paired in the join $R \bowtie S$, then the result of the pairing is a tuple, called the *joined tuple*, with one component for each of the attributes in the union of the schemas of R and S. The joined tuple

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

(c) Result $R \times S$

Figure 2.14: Two relations and their Cartesian product

agrees with tuple r in each attribute in the schema of R, and it agrees with s in each attribute in the schema of S. Since r and s are successfully paired, the joined tuple is able to agree with both these tuples on the attributes they have in common. The construction of the joined tuple is suggested by Fig. 2.15. However, the order of the attributes need not be that convenient; the attributes of R and S can appear in any order.

Example 2.13: The natural join of the relations R and S from Fig. 2.14(a) and (b) is

A	B	C	D
1	2	5	6
3	4	7	8

The only attribute common to R and S is B. Thus, to pair successfully, tuples need only to agree in their B components. If so, the resulting tuple has components for attributes A (from R), B (from either R or S), C (from S), and D (from S).

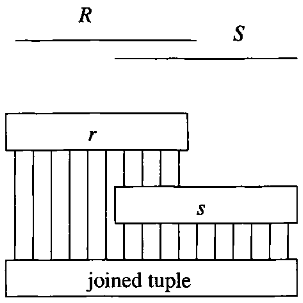

Figure 2.15: Joining tuples

In this example, the first tuple of R successfully pairs with only the first tuple of S; they share the value 2 on their common attribute B. This pairing yields the first tuple of the result: $(1,2,5,6)$. The second tuple of R pairs successfully only with the second tuple of S, and the pairing yields $(3,4,7,8)$. Note that the third tuple of S does not pair with any tuple of R and thus has no effect on the result of $R \bowtie S$. A tuple that fails to pair with any tuple of the other relation in a join is said to be a *dangling tuple*. $\square$

Example 2.14: The previous example does not illustrate all the possibilities inherent in the natural join operator. For example, no tuple paired successfully with more than one tuple, and there was only one attribute in common to the two relation schemas. In Fig. 2.16 we see two other relations, U and V, that share two attributes between their schemas: B and C. We also show an instance in which one tuple joins with several tuples.

For tuples to pair successfully, they must agree in both the B and C components. Thus, the first tuple of U joins with the first two tuples of V, while the second and third tuples of U join with the third tuple of V. The result of these four pairings is shown in Fig. 2.16(c). $\square$

2.4.9 Theta-Joins

The natural join forces us to pair tuples using one specific condition. While this way, equating shared attributes, is the most common basis on which relations are joined, it is sometimes desirable to pair tuples from two relations on some other basis. For that purpose, we have a related notation called the *theta-join.* Historically, the "theta" refers to an arbitrary condition, which we shall represent by C rather than θ.

The notation for a theta-join of relations R and S based on condition C is $R \bowtie_C S$. The result of this operation is constructed as follows:

1. Take the product of R and S.

2. Select from the product only those tuples that satisfy the condition C.

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

Figure 2.16: Natural join of relations

As with the product operation, the schema for the result is the union of the schemas of R and S, with "R." or "S." prefixed to attributes if necessary to indicate from which schema the attribute came.

Example 2.15: Consider the operation $U \bowtie_{A<D} V$, where U and V are the relations from Fig. 2.16(a) and (b). We must consider all nine pairs of tuples, one from each relation, and see whether the A component from the U-tuple is less than the D component of the V-tuple. The first tuple of U, with an A component of 1, successfully pairs with each of the tuples from V. However, the second and third tuples from U, with A components of 6 and 9, respectively, pair successfully with only the last tuple of V. Thus, the result has only five tuples, constructed from the five successful pairings. This relation is shown in Fig. 2.17. □

Notice that the schema for the result in Fig. 2.17 consists of all six attributes, with U and V prefixed to their respective occurrences of attributes B and C to distinguish them. Thus, the theta-join contrasts with natural join, since in the latter common attributes are merged into one copy. Of course it makes sense to

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Figure 2.17: Result of $U \bowtie_{A<D} V$

do so in the case of the natural join, since tuples don't pair unless they agree in their common attributes. In the case of a theta-join, there is no guarantee that compared attributes will agree in the result, since they may not be compared with $=$.

Example 2.16: Here is a theta-join on the same relations U and V that has a more complex condition:

$$U \bowtie_{A<D \texttt{ AND } U.B \neq V.B} V$$

That is, we require for successful pairing not only that the A component of the U-tuple be less than the D component of the V-tuple, but that the two tuples disagree on their respective B components. The tuple

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

is the only one to satisfy both conditions, so this relation is the result of the theta-join above. □

2.4.10 Combining Operations to Form Queries

If all we could do was to write single operations on one or two relations as queries, then relational algebra would not be nearly as useful as it is. However, relational algebra, like all algebras, allows us to form expressions of arbitrary complexity by applying operations to the result of other operations.

One can construct expressions of relational algebra by applying operators to subexpressions, using parentheses when necessary to indicate grouping of operands. It is also possible to represent expressions as expression trees; the latter often are easier for us to read, although they are less convenient as a machine-readable notation.

Example 2.17: Suppose we want to know, from our running `Movies` relation, "What are the titles and years of movies made by Fox that are at least 100 minutes long?" One way to compute the answer to this query is:

1. Select those `Movies` tuples that have *length* ≥ 100.

2. Select those `Movies` tuples that have $studioName$ = `'Fox'`.

3. Compute the intersection of (1) and (2).

4. Project the relation from (3) onto attributes `title` and `year`.

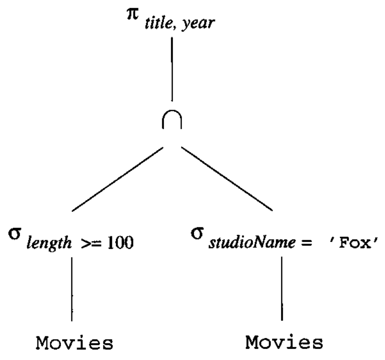

Figure 2.18: Expression tree for a relational algebra expression

In Fig. 2.18 we see the above steps represented as an expression tree. Expression trees are evaluated bottom-up by applying the operator at an interior node to the arguments, which are the results of its children. By proceeding bottom-up, we know that the arguments will be available when we need them. The two selection nodes correspond to steps (1) and (2). The intersection node corresponds to step (3), and the projection node is step (4).

Alternatively, we could represent the same expression in a conventional, linear notation, with parentheses. The formula

$$\pi_{title,year}\Big(\sigma_{length\geq 100}(\texttt{Movies}) \cap \sigma_{studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

represents the same expression.

Incidentally, there is often more than one relational algebra expression that represents the same computation. For instance, the above query could also be written by replacing the intersection by logical **AND** within a single selection operation. That is,

$$\pi_{title,year}\Big(\sigma_{length\geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

is an equivalent form of the query. □

Equivalent Expressions and Query Optimization

All database systems have a query-answering system, and many of them are based on a language that is similar in expressive power to relational algebra. Thus, the query asked by a user may have many ***equivalent expressions*** (expressions that produce the same answer whenever they are given the same relations as operands), and some of these may be much more quickly evaluated. An important job of the query "optimizer" discussed briefly in Section 1.2.5 is to replace one expression of relational algebra by an equivalent expression that is more efficiently evaluated.

2.4.11 Naming and Renaming

In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations. We shall use the operator $\rho_{S(A_1,A_2,\ldots,A_n)}(R)$ to rename a relation R. The resulting relation has exactly the same tuples as R, but the name of the relation is S. Moreover, the attributes of the result relation S are named $A_1, A_2, \ldots, A_n$, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say $\rho_S(R)$.

Example 2.18: In Example 2.12 we took the product of two relations R and S from Fig. 2.14(a) and (b) and used the convention that when an attribute appears in both operands, it is renamed by prefixing the relation name to it. Suppose, however, that we do not wish to call the two versions of B by names $R.B$ and $S.B$; rather we want to continue to use the name B for the attribute that comes from R, and we want to use X as the name of the attribute B coming from S. We can rename the attributes of S so the first is called X. The result of the expression $\rho_{S(X,C,D)}(S)$ is a relation named S that looks just like the relation S from Fig. 2.14, but its first column has attribute X instead of B.

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Figure 2.19: $R \times \rho_{S(X,C,D)}(S)$

When we take the product of R with this new relation, there is no conflict of names among the attributes, so no further renaming is done. That is, the result of the expression $R \times \rho_{S(X,C,D)}(S)$ is the relation $R \times S$ from Fig. 2.14(c), except that the five columns are labeled A, B, X, C, and D, from the left. This relation is shown in Fig. 2.19.

As an alternative, we could take the product without renaming, as we did in Example 2.12, and then rename the result. The expression

$$\rho_{RS(A,B,X,C,D)}(R \times S)$$

yields the same relation as in Fig. 2.19, with the same set of attributes. But this relation has a name, RS, while the result relation in Fig. 2.19 has no name. □

2.4.12 Relationships Among Operations

Some of the operations that we have described in Section 2.4 can be expressed in terms of other relational-algebra operations. For example, intersection can be expressed in terms of set difference:

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

That is, if R and S are any two relations with the same schema, the intersection of R and S can be computed by first subtracting S from R to form a relation T consisting of all those tuples in R but not S. We then subtract T from R, leaving only those tuples of R that are also in S.

The two forms of join are also expressible in terms of other operations. Theta-join can be expressed by product and selection:

$$R \bowtie_C S = \sigma_C(R \times S)$$

The natural join of R and S can be expressed by starting with the product $R \times S$. We then apply the selection operator with a condition C of the form

$$R.A_1 = S.A_1 \text{ AND } R.A_2 = S.A_2 \text{ AND} \cdots \text{AND } R.A_n = S.A_n$$

where $A_1, A_2, \ldots, A_n$ are all the attributes appearing in the schemas of both R and S. Finally, we must project out one copy of each of the equated attributes. Let L be the list of attributes in the schema of R followed by those attributes in the schema of S that are not also in the schema of R. Then

$$R \bowtie S = \pi_L\Big(\sigma_C(R \times S)\Big)$$

Example 2.19: The natural join of the relations U and V from Fig. 2.16 can be written in terms of product, selection, and projection as:

$$\pi_{A,U.B,U.C,D}\Big(\sigma_{U.B=V.B \text{ AND } U.C=V.C}(U \times V)\Big)$$

That is, we take the product $U \times V$. Then we select for equality between each pair of attributes with the same name — B and C in this example. Finally, we project onto all the attributes except one of the B's and one of the C's; we have chosen to eliminate the attributes of V whose names also appear in the schema of U.

For another example, the theta-join of Example 2.16 can be written

$$\sigma_{A<D \texttt{ AND } U.B \neq V.B}(U \times V)$$

That is, we take the product of the relations U and V and then apply the condition that appeared in the theta-join. □

The rewriting rules mentioned in this section are the only "redundancies" among the operations that we have introduced. The six remaining operations — union, difference, selection, projection, product, and renaming — form an independent set, none of which can be written in terms of the other five.

2.4.13 A Linear Notation for Algebraic Expressions

In Section 2.4.10 we used an expression tree to represent a complex expression of relational algebra. An alternative is to invent names for the temporary relations that correspond to the interior nodes of the tree and write a sequence of assignments that create a value for each. The order of the assignments is flexible, as long as the children of a node N have had their values created before we attempt to create the value for N itself.

The notation we shall use for assignment statements is:

1. A relation name and parenthesized list of attributes for that relation. The name `Answer` will be used conventionally for the result of the final step; i.e., the name of the relation at the root of the expression tree.

2. The assignment symbol `:=`.

3. Any algebraic expression on the right. We can choose to use only one operator per assignment, in which case each interior node of the tree gets its own assignment statement. However, it is also permissible to combine several algebraic operations in one right side, if it is convenient to do so.

Example 2.20: Consider the tree of Fig. 2.18. One possible sequence of assignments to evaluate this expression is:

```
R(t,y,l,i,s,p) := σ_{length≥100}(Movies)
S(t,y,l,i,s,p) := σ_{studioName='Fox'}(Movies)
T(t,y,l,i,s,p) := R ∩ S
Answer(title, year) := π_{t,y}(T)
```

The first step computes the relation of the interior node labeled $\sigma_{length\geq 100}$ in Fig. 2.18, and the second step computes the node labeled $\sigma_{studioName='\texttt{Fox}'}$. Notice that we get renaming "for free," since we can use any attributes and relation name we wish for the left side of an assignment. The last two steps compute the intersection and the projection in the obvious way.

It is also permissible to combine some of the steps. For instance, we could combine the last two steps and write:

$$\begin{aligned}&\texttt{R(t,y,l,i,s,p)} := \sigma_{length\geq 100}(\texttt{Movies})\\&\texttt{S(t,y,l,i,s,p)} := \sigma_{studioName='\texttt{Fox}'}(\texttt{Movies})\\&\texttt{Answer(title, year)} := \pi_{t,y}(\texttt{R} \cap \texttt{S})\end{aligned}$$

We could even substitute for R and S in the last line and write the entire expression in one line. □

2.4.14 Exercises for Section 2.4

Exercise 2.4.1: This exercise builds upon the products schema of Exercise 2.3.1. Recall that the database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

Some sample data for the relation `Product` is shown in Fig. 2.20. Sample data for the other three relations is shown in Fig. 2.21. Manufacturers and model numbers have been "sanitized," but the data is typical of products on sale at the beginning of 2007.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 2.4.13 if you wish. For the data of Figs. 2.20 and 2.21, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

a) What PC models have a speed of at least 3.00?

b) Which manufacturers make laptops with a hard disk of at least 100GB?

c) Find the model number and price of all products (of any type) made by manufacturer B.

d) Find the model numbers of all color laser printers.

e) Find those manufacturers that sell Laptops, but not PC's.

! f) Find those hard-disk sizes that occur in two or more PC's.