4 RELATIONAL ALGEBRA AND CALCULUS

> Stand firm in your refusal to remain conscious during algebra. In real life, I assure you, there is no such thing as algebra.
>
> —Fran Lebowitz, *Social Studies*

This chapter presents two formal query languages associated with the relational model. **Query languages** are specialized languages for asking questions, or **queries**, that involve the data in a database. After covering some preliminaries in Section 4.1, we discuss *relational algebra* in Section 4.2. Queries in relational algebra are composed using a collection of operators, and each query describes a step-by-step procedure for computing the desired answer; that is, queries are specified in an *operational* manner. In Section 4.3 we discuss **relational calculus**, in which a query describes the desired answer without specifying how the answer is to be computed; this nonprocedural style of querying is called *declarative*. We will usually refer to relational algebra and relational calculus as algebra and calculus, respectively. We compare the expressive power of algebra and calculus in Section 4.4. These formal query languages have greatly influenced commercial query languages such as SQL, which we will discuss in later chapters.

4.1 PRELIMINARIES

We begin by clarifying some important points about relational queries. The inputs and outputs of a query are relations. A query is evaluated using *instances* of each input relation and it produces an instance of the output relation. In Section 3.4, we used field names to refer to fields because this notation makes queries more readable. An alternative is to always list the fields of a given relation in the same order and to refer to fields by position rather than by field name.

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to 'inherit' field names, for convenience.

We present a number of sample queries using the following schema:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `date`)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

In several examples illustrating the relational algebra operators, we will use the instances $S1$ and $S2$ (of Sailors) and $R1$ (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.1 Instance $S1$ of Sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance $S2$ of Sailors

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 4.3 Instance $R1$ of Reserves

4.2 RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to *compose* operators to form a complex query—a **relational algebra expression** is recursively defined to be a relation, a unary algebra operator applied

to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

4.2.1 Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression

$$\sigma_{rating>8}(S2)$$

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of *terms* that have the form *attribute* `op` *constant* or *attribute1* `op` *attribute2*, where `op` is one of the comparison operators $<, <=, =, \neq, >=$, or $>$. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression

$$\pi_{sname,rating}(S2)$$

evaluates to the relation shown in Figure 4.5. The subscript *sname,rating* specifies the fields to be retained; the other fields are 'projected out.' The schema of the result of a projection is determined by the fields that are projected in the obvious way.

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

evaluates to the relation shown in Figure 4.6. The important point to note is that although three sailors are aged 35, a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. For example, we can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

produces the result shown in Figure 4.7. It is obtained by applying the selection to $S2$ (to get the relation shown in Figure 4.4) and then applying the projection.

age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

4.2.2 Set Operations

The following standard operations on sets are also available in relational algebra: *union* ($\cup$), *intersection* ($\cap$), *set-difference* ($-$), and *cross-product* ($\times$).

- **Union:** $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance R or relation instance S (or both). R and S must be *union-compatible*, and the schema of the result is defined to be identical to the schema of R.

  Two relation instances are said to be **union-compatible** if the following conditions hold:
  - they have the same number of the fields, and
  - corresponding fields, taken in order from left to right, have the same *domains*.

Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of $R \cup S$ inherit names from R, if the fields of R have names. (This assumption is implicit in defining the schema of $R \cup S$ to be identical to the schema of R, as stated earlier.)

- **Intersection:** $R \cap S$ returns a relation instance containing all tuples that occur in *both* R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- **Set-difference:** $R - S$ returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- **Cross-product:** $R \times S$ returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of $R \times S$ contains one tuple $\langle r, s \rangle$ (the concatenation of tuples r and s) for each pair of tuples $r \in R$, $s \in S$. The cross-product opertion is sometimes called **Cartesian product**.

  We will use the convention that the fields of $R \times S$ inherit names from the corresponding fields of R and S. It is possible for both R and S to contain one or more fields having the same name; this situation creates a *naming conflict*. The corresponding fields in $R \times S$ are unnamed and are referred to solely by position.

In the preceding definitions, note that each operator can be applied to relation instances that are computed using a relational algebra (sub)expression.

We now illustrate these definitions through several examples. The union of $S1$ and $S2$ is shown in Figure 4.8. Fields are listed in order; field names are also inherited from $S1$. $S2$ has the same field names, of course, since it is also an instance of Sailors. In general, fields of $S2$ may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both $S1$ and $S2$ appear only once in $S1 \cup S2$. Also, $S1 \cup R1$ is not a valid operation because the two relations are not union-compatible. The intersection of $S1$ and $S2$ is shown in Figure 4.9, and the set-difference $S1 - S2$ is shown in Figure 4.10.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

Figure 4.8 $S1 \cup S2$

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11. Because $R1$ and $S1$ both have a field named *sid*, by our convention on field names, the corresponding two fields in $S1 \times R1$ are unnamed, and referred to solely by the position in which they appear in Figure 4.11. The fields in $S1 \times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11 *sid* is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 $S1 \times R1$

4.2.3 Renaming

We have been careful to adopt field name conventions that ensure that the result of a relational algebra expression inherits field names from its argument (input) relation instances in a natural way whenever possible. However, name conflicts can arise in some cases; for example, in $S1 \times R1$. It is therefore convenient to be able to give names explicitly to the fields of a relation instance that is defined by a relational algebra expression. In fact, it is often convenient to give the instance itself a name so that we can break a large algebra expression into smaller pieces by giving names to the results of subexpressions.

We introduce a **renaming** operator ρ for this purpose. The expression $\rho(R(\overline{F}), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list* $\overline{F}$, which is a list of

terms having the form *oldname* → *newname* or *position* → *newname*. For ρ to be well-defined, references to fields (in the form of *oldname*s or *position*s in the renaming list) may be unambiguous, and no two fields in the result must have the same name. Sometimes we only want to rename fields or to (re)name the relation; we will therefore treat both R and $\overline{F}$ as optional in the use of ρ. (Of course, it is meaningless to omit both.)

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`, *sid2:* `integer`, *bid:* `integer`, *day:* `dates`).

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the $\cap$ operator is redundant; $R \cap S$ can be defined as $R - (R - S)$.) We will consider these additional operators, and their definition in terms of the basic operators, in the next two subsections.

4.2.4 Joins

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, and it is very important to recognize joins and implement them without materializing the underlying cross-product (by applying the selections and projections 'on-the-fly'). For these reasons, joins have received a lot of attention, and there are several variants of the join operation.[1]

Condition Joins

The most general version of the join operation accepts a *join condition* c and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

$$R \bowtie_c S \ = \ \sigma_c(R \times S)$$

Thus $\bowtie$ is defined to be a cross-product followed by a selection. Note that the condition c can (and typically *does*) refer to attributes of both R and S. The reference to an

[1] There are several variants of joins that are not discussed in this chapter. An important class of joins called *outer joins* is discussed in Chapter 5.

attribute of a relation, say R, can be by position (of the form $R.i$) or by name (of the form $R.name$).

As an example, the result of $S1 \bowtie_{S1.sid<R1.sid} R1$ is shown in Figure 4.12. Because *sid* appears in both $S1$ and $R1$, the corresponding fields in the result of the cross-product $S1 \times R1$ (and therefore in the result of $S1 \bowtie_{S1.sid<R1.sid} R1$) are unnamed. Domains are inherited from the corresponding fields of $S1$ and $R1$.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Figure 4.12 $S1 \bowtie_{S1.sid<R1.sid} R1$

Equijoin

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities (connected by $\wedge$) of the form $R.name1 = S.name2$, that is, equalities between two fields in R and S. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which $S.name2$ is dropped. The join operation with this refinement is called **equijoin**.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} R1$ in Figure 4.13. Notice that only one field called *sid* appears in the result.

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Figure 4.13 $S1 \bowtie_{R.sid=S.sid} R1$

Natural Join

A further special case of the join operation $R \bowtie S$ is an equijoin in which equalities are specified on *all* fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural join*, and it has the nice property that the result is guaranteed not to have two fields with the same name.

The equijoin expression $S1 \bowtie_{R.sid=S.sid} R1$ is actually a natural join and can simply be denoted as $S1 \bowtie R1$, since the only common field is *sid*. If the two relations have no attributes in common, $S1 \bowtie R1$ is simply the cross-product.

4.2.5 Division

The division operator is useful for expressing certain kinds of queries, for example: "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise. However, the division operator does not have the same importance as the other operators—it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the *division* operation A/B as the set of all x values (in the form of unary tuples) such that for *every* y value in (a tuple of) B, there is a tuple $\langle x,y \rangle$ in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, the x value is in the result of A/B.

An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that $Q * B \leq A$. For relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.

Division is illustrated in Figure 4.14. It helps to think of A as a relation listing the parts supplied by suppliers, and of the B relations as listing parts. A/Bi computes suppliers who supply *all* parts listed in relation instance Bi.

Expressing A/B in terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all x values in A that are not *disqualified*. An x value is *disqualified* if by attaching a

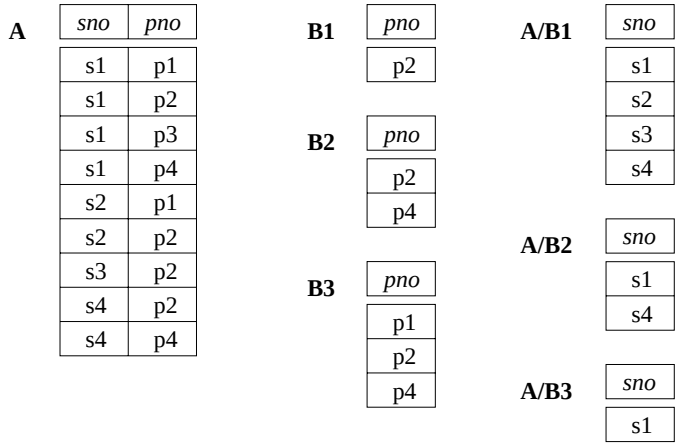

Figure 4.14 Examples Illustrating Division

y value from B, we obtain a tuple $\langle x,y \rangle$ that is not in A. We can compute disqualified tuples using the algebra expression

$$\pi_x((\pi_x(A) \times B) - A)$$

Thus we can define A/B as

$$\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

To understand the division operation in full generality, we have to consider the case when both x and y are replaced by a set of attributes. The generalization is straightforward and is left as an exercise for the reader. We will discuss two additional examples illustrating division (Queries Q9 and Q10) later in this section.

4.2.6 More Examples of Relational Algebra Queries

We now present several examples to illustrate how to write queries in relational algebra. We use the Sailors, Reserves, and Boats schema for all our examples in this section. We will use parentheses as needed to make our algebra expressions unambiguous. Note that all the example queries in this chapter are given a unique query number. The query numbers are kept unique across both this chapter and the SQL query chapter (Chapter 5). This numbering makes it easy to identify a query when it is revisited in the context of relational calculus and SQL and to compare different ways of writing the same query. (All references to a query can be found in the subject index.)

In the rest of this chapter (and in Chapter 5), we illustrate queries using the instances $S3$ of Sailors, $R2$ of Reserves, and $B1$ of Boats, shown in Figures 4.15, 4.16, and 4.17, respectively.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance $B1$ of Boats

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

We first compute the set of tuples in Reserves with $bid = 103$ and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances $R2$ and $S3$, it yields a relation that contains just one field, called *sname*, and three tuples $\langle Dustin \rangle$, $\langle Horatio \rangle$, and $\langle Lubber \rangle$. (Observe that there are two sailors called Horatio, and only one of them has reserved a red boat.)

We can break this query into smaller pieces using the renaming operator ρ:

$$\rho(Temp1, \sigma_{bid=103}Reserves)$$

$$\rho(Temp2, Temp1 \bowtie Sailors)$$
$$\pi_{sname}(Temp2)$$

Notice that because we are only using ρ to give names to intermediate relations, the renaming list is optional and is omitted. $Temp1$ denotes an intermediate relation that identifies reservations of boat 103. $Temp2$ is another intermediate relation, and it denotes sailors who have made a reservation in the set $Temp1$. The instances of these relations when evaluating this query on the instances $R2$ and $S3$ are illustrated in Figures 4.18 and 4.19. Finally, we extract the *sname* column from $Temp2$.

sid	*bid*	*day*
22	103	10/8/98
31	103	11/6/98
74	103	9/8/98

Figure 4.18 Instance of $Temp1$

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	103	10/8/98
31	Lubber	8	55.5	103	11/6/98
74	Horatio	9	35.0	103	9/8/98

Figure 4.19 Instance of $Temp2$

The version of the query using ρ is essentially the same as the original query; the use of ρ is just syntactic sugar. However, there are indeed several distinct ways to write a query in relational algebra. Here is another way to write this query:

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

In this version we first compute the natural join of Reserves and Sailors and then apply the selection and the projection.

This example offers a glimpse of the role played by algebra in a relational DBMS. Queries are expressed by users in a language such as SQL. The DBMS translates an SQL query into (an extended form of) relational algebra, and then looks for other algebra expressions that will produce the same answers but are cheaper to evaluate. If the user's query is first translated into the expression

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

a good query optimizer will find the equivalent expression

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

Further, the optimizer will recognize that the second expression is likely to be less expensive to compute because the sizes of intermediate relations are smaller, thanks to the early use of selection.

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi_{sname}((\sigma_{color='red'}Boats) \bowtie Reserves \bowtie Sailors)$$

This query involves a series of two joins. First we choose (tuples describing) red boats. Then we join this set with Reserves (natural join, with equality specified on the *bid* column) to identify reservations of red boats. Next we join the resulting intermediate relation with Sailors (natural join, with equality specified on the *sid* column) to retrieve the names of sailors who have made reservations of red boats. Finally, we project the sailors' names. The answer, when evaluated on the instances $B1$, $R2$ and $S3$, contains the names Dustin, Horatio, and Lubber.

An equivalent expression is:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}Boats) \bowtie Reserves) \bowtie Sailors)$$

The reader is invited to rewrite both of these queries by using ρ to make the intermediate relations explicit and to compare the schemas of the intermediate relations. The second expression generates intermediate relations with fewer fields (and is therefore likely to result in intermediate relation instances with fewer tuples, as well). A relational query optimizer would try to arrive at the second expression if it is given the first.

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'}Sailors) \bowtie Reserves \bowtie Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances $B1$, $R2$, and $S3$, the query will return the colors gren and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \bowtie Reserves)$$

The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple 'attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same *sid* value, that is, the sailor has made some reservation. The answer, when evaluated on the instances $B1$, $R2$ and $S3$, contains the three tuples $\langle Dustin \rangle$, $\langle Horatio \rangle$, and $\langle Lubber \rangle$. Even though there are two sailors called Horatio who have reserved a boat, the answer contains only one copy of the tuple $\langle Horatio \rangle$, because the answer is a *relation*, i.e., a *set* of tuples, without any duplicates.

At this point it is worth remarking on how frequently the natural join operation is used in our examples. This frequency is more than just a coincidence based on the set of queries that we have chosen to discuss; the natural join is a very natural and widely used operation. In particular, natural join is frequently used when joining two tables on a foreign key field. In Query Q4, for example, the join equates the *sid* fields of Sailors and Reserves, and the *sid* field of Reserves is a foreign key that refers to the *sid* field of Sailors.

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

We identify the set of all boats that are either red or green (Tempboats, which contains boats with the *bid*s 102, 103, and 104 on instances $B1$, $R2$, and $S3$). Then we join with Reserves to identify *sid*s of sailors who have reserved one of these boats; this gives us *sid*s 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of *sid*s) with Sailors to find the names of Sailors with these *sid*s. This gives us the names Dustin, Horatio, and Lubber on the instances $B1$, $R2$, and $S3$. Another equivalent definition is the following:

$$\rho(Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

Let us now consider a very similar query:

(Q6) Find the names of sailors who have reserved a red and a green boat. It is tempting to try to do this by simply replacing $\cup$ by $\cap$ in the definition of Tempboats:

$$\rho(Tempboats2, (\sigma_{color='red'} Boats) \cap (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats2 \bowtie Reserves \bowtie Sailors)$$

However, this solution is incorrect—it instead tries to compute sailors who have reserved a boat that is both red and green. (Since *bid* is a key for Boats, a boat can be only one color; this query will always return an empty answer set.) The correct approach is to find sailors who have reserved a red boat, then sailors who have reserved a green boat, and then take the intersection of these two sets:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

The two temporary relations compute the *sid*s of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances $B1$, $R2$, and $S3$, the *sid*s of sailors who have reserved a red boat are 22, 31, and 64. The *sid*s of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.

This formulation of Query Q6 can easily be adapted to find sailors who have reserved red *or* green boats (Query Q5); just replace $\cap$ by $\cup$:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cup Tempgreen) \bowtie Sailors)$$

In the above formulations of Queries Q5 and Q6, the fact that *sid* (the field over which we compute union or intersection) is a key for Sailors is very important. Consider the following attempt to answer Query Q6:

$$\rho(Tempred, \pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors))$$
$$\rho(Tempgreen, \pi_{sname}((\sigma_{color='green'} Boats) \bowtie Reserves \bowtie Sailors))$$
$$Tempred \cap Tempgreen$$

This attempt is incorrect for a rather subtle reason. Two distinct sailors with the same name, such as Horatio in our example instances, may have reserved red and green boats, respectively. In this case, the name Horatio will (incorrectly) be included in the answer even though no one individual called Horatio has reserved a red boat and a green boat. The cause of this error is that *sname* is being used to identify sailors (while doing the intersection) in this version of the query, but *sname* is not a key.

(Q7) Find the names of sailors who have reserved at least two boats.

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$
$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$
$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$
$$\pi_{sname1}\sigma_{(sid1=sid2)\wedge(bid1\neq bid2)} Reservationpairs$$

First we compute tuples of the form ⟨*sid,sname,bid*⟩, where sailor *sid* has made a reservation for boat *bid*; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: In order to show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances $B1$, $R2$, and $S3$, the sailors with *sid*s 22, 31, and 64 have each reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

Notice that we included *sid* in Reservations because it is the key field identifying sailors, and we need it to check that two Reservations tuples involve the same sailor. As noted in the previous example, we can't use *sname* for this purpose.

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$$\pi_{sid}(\sigma_{age>20} Sailors) -$$
$$\pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors. We first identify sailors aged over 20 (over instances $B1$, $R2$, and $S3$, *sid*s 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who

have reserved a red boat (*sid*s 22, 31, and 64), to obtain the answer (*sid*s 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their *sid*s (as shown above), and then join with Sailors and project the *sname* values.

(Q9) Find the names of sailors who have reserved all boats. The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The intermediate relation Tempsids is defined using division, and computes the set of *sid*s of sailors who have reserved every boat (over instances $B1$, $R2$, and $S3$, this is just *sid* 22). Notice how we define the two relations that the division operator (/) is applied to—the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sid*s such that there is a tuple ⟨*sid,bid*⟩ in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sid*s; for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute only *bid*s of boats named *Interlake* in defining the second argument to the division operator. Over instances $B1$, $R2$, and $S3$, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.

4.3 RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus that we present in detail is called the tuple relational calculus (TRC). Variables in TRC take on tuples as values. In another variant, called the domain relational calculus (DRC), the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE. We discuss DRC in Section 4.3.2.[2]

[2]The material on DRC is referred to in the chapter on QBE; with the exception of this chapter, the material on DRC and TRC can be omitted without loss of continuity.

4.3.1 Tuple Relational Calculus

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form { $T \mid p(T)$ }, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to `true` with $T = t$. The language for writing formulas $p(T)$ is thus at the heart of TRC and is essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

(Q11) Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test $S.rating > 7$ is applied. The answer contains those instances of S that pass this test. On instance $S3$ of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, R and S be tuple variables, a an attribute of R, and b an attribute of S. Let `op` denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$. An **atomic formula** is one of the following:

- $R \in Rel$
- $R.a$ `op` $S.b$
- $R.a$ `op` *constant*, or *constant* `op` $R.a$

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and $p(R)$ denotes a formula in which the variable R appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

In the last two clauses above, the **quantifiers** $\exists$ and $\forall$ are said to **bind** the variable R. A variable is said to be **free** in a formula or *subformula* (a formula contained in a

larger formula) if the (sub)formula does not contain an occurrence of a quantifier that binds it.[3]

We observe that every variable in a TRC formula appears in a subformula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type*, in the programming language sense. Informally, an atomic formula $R \in Rel$ gives R the type of tuples in Rel, and comparisons such as $R.a$ `op` $S.b$ and $R.a$ `op` *constant* induce type restrictions on the field $R.a$. If a variable R does not appear in an atomic formula of the form $R \in Rel$ (i.e., it appears only in atomic formulas that are comparisons), we will follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We will not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and that this is the domain associated with each field of each relation.)

A **TRC query** is defined to be expression of the form $\{T \mid p(T)\}$, where T is the only free variable in the formula p.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The **answer** to a TRC query $\{T \mid p(T)\}$, as we noted earlier, is the set of all tuples t for which the formula $p(T)$ evaluates to `true` with variable T assigned the tuple value t. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to `true`.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to (or simply 'is') `true` if one of the following holds:

- F is an atomic formula $R \in Rel$, and R is assigned a tuple in the instance of relation Rel.

[3]We will make the assumption that each variable in a formula is either free or bound by exactly one occurrence of a quantifier, to avoid worrying about details such as nested occurrences of quantifiers that bind some, but not all, occurrences of variables.

- F is a comparison $R.a$ `op` $S.b$, $R.a$ `op` *constant*, or *constant* `op` $R.a$, and the tuples assigned to R and S have field values $R.a$ and $S.b$ that make the comparison `true`.
- F is of the form $\neg p$, and p is not `true`; or of the form $p \wedge q$, and both p and q are `true`; or of the form $p \vee q$, and one of them is `true`, or of the form $p \Rightarrow q$ and q is `true` whenever[4] p is `true`.
- F is of the form $\exists R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$, including the variable R,[5] that makes the formula $p(R)$ `true`.
- F is of the form $\forall R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$ that makes the formula $p(R)$ `true` no matter what tuple is assigned to R.

Examples of TRC Queries

We now illustrate the calculus through several examples, using the instances $B1$ of Boats, $R2$ of Reserves, and $S3$ of Sailors shown in Figures 4.15, 4.16, and 4.17. We will use parentheses as needed to make our formulas unambiguous. Often, a formula $p(R)$ includes a condition $R \in Rel$, and the meaning of the phrases *some tuple* R and *for all tuples* R is intuitive. We will use the notation $\exists R \in Rel(p(R))$ for $\exists R(R \in Rel \wedge p(R))$. Similarly, we use the notation $\forall R \in Rel(p(R))$ for $\forall R(R \in Rel \Rightarrow p(R))$.

(Q12) Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of P that are mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form $P \in Relname$. The result of this query is a relation with two fields, *name* and *age*. The atomic formulas $P.name = S.sname$ and $P.age = S.age$ give values to the fields of an answer tuple P. On instances $B1$, $R2$, and $S3$, the answer is the set of tuples $\langle Lubber, 55.5\rangle$, $\langle Andy, 25.5\rangle$, $\langle Rusty, 35.0\rangle$, $\langle Zorba, 16.0\rangle$, and $\langle Horatio, 35.0\rangle$.

(Q13) Find the sailor name, boat id, and reservation date for each reservation.

$$\{P \mid \exists R \in Reserves \;\; \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

For each Reserves tuple, we look for a tuple in Sailors with the same *sid*. Given a pair of such tuples, we construct an answer tuple P with fields *sname, bid*, and *day* by

[4] *Whenever* should be read more precisely as 'for all assignments of tuples to the free variables.'
[5] Note that some of the free variables in $p(R)$ (e.g., the variable R itself) may be bound in F.

copying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances $B1$, $R2$, and $S3$ is shown in Figure 4.20.

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

Figure 4.20 Answer to Query Q13

(Q1) Find the names of sailors who have reserved boat 103.

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with $bid = 103$." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, *sname*.

(Q2) Find the names of sailors who have reserved a red boat.

$$\{P \mid \exists S \in Sailors\ \ \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname$$
$$\wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.color = 'red'$." Another way to write this query, which corresponds more closely to this reading, is as follows:

$$\{P \mid \exists S \in Sailors\ \ \exists R \in Reserves\ \ \exists B \in Boats$$
$$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors\ \ \exists R1 \in Reserves\ \ \exists R2 \in Reserves$$
$$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors \ \ \forall B \in Boats$$
$$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Notice how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply the sailor having reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor's name, we have avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances $B1$, $R2$, and $S3$, the answer contains the Sailors tuples with *sid*s 22 and 31.

We can write this query without using implication, by observing that an expression of the form $p \Rightarrow q$ is logically equivalent to $\neg p \vee q$:

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color \neq 'red' \vee (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

4.3.2 Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{\langle x_1, x_2, \ldots, x_n\rangle \mid p(\langle x_1, x_2, \ldots, x_n\rangle)\}$, where each x_i is either a *domain variable* or a constant and $p(\langle x_1, x_2, \ldots, x_n\rangle)$ denotes a **DRC formula** whose only free variables are the variables among the x_i, $1 \leq i \leq n$. The result of this query is the set of all tuples $\langle x_1, x_2, \ldots, x_n\rangle$ for which the formula evaluates to `true`.

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$ and let X and Y be domain variables. An **atomic formula** in DRC is one of the following:

- $\langle x_1, x_2, \ldots, x_n \rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \leq i \leq n$ is either a variable or a constant.
- X op Y
- X op *constant*, or *constant* op X

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and $p(X)$ denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

The reader is invited to compare this definition with the definition of TRC formulas and see how closely these two definitions correspond. We will not define the semantics of DRC formulas formally; this is left as an exercise for the reader.

Examples of DRC Queries

We now illustrate DRC through several examples. The reader is invited to compare these with the TRC versions.

(Q11) Find all sailors with a rating above 7.

$$\{\langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in Sailors \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition $\langle I, N, T, A \rangle \in Sailors$ ensures that the domain variables I, N, T, and A are restricted to be fields of the *same* tuple. In comparison with the TRC query, we can say $T > 7$ instead of $S.rating > 7$, but we must specify the tuple $\langle I, N, T, A \rangle$ in the result, rather than just S.

(Q1) Find the names of sailors who have reserved boat 103.

$$\{\langle N \rangle \mid \exists I, T, A(\langle I, N, T, A \rangle \in Sailors \wedge \exists Ir, Br, D(\langle Ir, Br, D \rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

Notice that only the *sname* field is retained in the answer and that only N is a free variable. We use the notation $\exists Ir, Br, D(\ldots)$ as a shorthand for $\exists Ir(\exists Br(\exists D(\ldots)))$. Very often, all the quantified variables appear in a single relation, as in this example. An even more compact notation in this case is $\exists\langle Ir, Br, D\rangle \in Reserves$. With this notation, which we will use henceforth, the above query would be as follows:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists\langle Ir, Br, D\rangle \in Reserves(Ir = I \wedge Br = 103))\}$$

The comparison with the corresponding TRC formula should now be straightforward. This query can also be written as follows; notice the repetition of variable I and the use of the constant 103:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists D(\langle I, 103, D\rangle \in Reserves))\}$$

(Q2) Find the names of sailors who have reserved a red boat.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists\langle I, Br, D\rangle \in Reserves \wedge \exists\langle Br, BN, 'red'\rangle \in Boats)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 \neq Br2)$$

Notice how the repeated use of variable I ensures that the same sailor has reserved both the boats in question.

(Q9) Find the names of sailors who have reserved all boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \forall B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \vee (\exists\langle Ir, Br, D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

This query can be read as follows: "Find all values of N such that there is some tuple $\langle I, N, T, A\rangle$ in Sailors satisfying the following condition: for every $\langle B, BN, C\rangle$, either this is not a tuple in Boats or there is some tuple $\langle Ir, Br, D\rangle$ in Reserves that proves that Sailor I has reserved boat B." The $\forall$ quantifier allows the domain variables B, BN, and C to range over all values in their respective attribute domains, and the pattern '$\neg(\langle B, BN, C\rangle \in Boats)\vee$' is necessary to restrict attention to those values that appear in tuples of Boats. This pattern is common in DRC formulas, and the notation $\forall\langle B, BN, C\rangle \in Boats$ can be used as a shorthand instead. This is similar to

the notation introduced earlier for $\exists$. With this notation the query would be written as follows:

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats \\ (\exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B)))\}$$

(Q14) Find sailors who have reserved all red boats.

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats \\ (C =' red' \Rightarrow \exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

Here, we find all sailors such that for every red boat there is a tuple in Reserves that shows the sailor has reserved it.

4.4 EXPRESSIVE POWER OF ALGEBRA AND CALCULUS *

We have presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus as we have presented it.

Consider the query $\{S \mid \neg(S \in Sailors)\}$. This query is syntactically correct. However, it asks for all tuples S such that S is not in (the given instance of) Sailors. The set of such S tuples is obviously infinite, in the context of infinite domains such as the set of all integers. This simple example illustrates an *unsafe* query. It is desirable to restrict relational calculus to disallow unsafe queries.

We now sketch how calculus queries are restricted to be safe. Consider a set I of relation instances, with one instance per relation that appears in the query Q. Let $Dom(Q,I)$ be the set of all constants that appear in these relation instances I or in the formulation of the query Q itself. Since we only allow finite instances I, $Dom(Q,I)$ is also finite.

For a calculus formula Q to be considered safe, at a minimum we want to ensure that for any given I, the set of answers for Q contains only values that are in $Dom(Q,I)$. While this restriction is obviously required, it is not enough. Not only do we want the set of answers to be composed of constants in $Dom(Q,I)$, we wish to *compute* the set of answers by only examining tuples that contain constants in $Dom(Q,I)$! This wish leads to a subtle point associated with the use of quantifiers $\forall$ and $\exists$: Given a TRC formula of the form $\exists R(p(R))$, we want to find all values for variable R that make this formula `true` by checking only tuples that contain constants in $Dom(Q,I)$. Similarly,

given a TRC formula of the form $\forall R(p(R))$, we want to find any values for variable R that make this formula `false` by checking only tuples that contain constants in $Dom(Q, I)$.

We therefore define a *safe* TRC formula Q to be a formula such that:

1. For any given I, the set of answers for Q contains only values that are in $Dom(Q, I)$.
2. For each subexpression of the form $\exists R(p(R))$ in Q, if a tuple r (assigned to variable R) makes the formula `true`, then r contains only constants in $Dom(Q, I)$.
3. For each subexpression of the form $\forall R(p(R))$ in Q, if a tuple r (assigned to variable R) contains a constant that is not in $Dom(Q, I)$, then r must make the formula `true`.

Note that this definition is not *constructive*, that is, it does not tell us how to check if a query is safe.

The query $Q = \{S \mid \neg(S \in Sailors)\}$ is unsafe by this definition. Dom(Q,I) is the set of all values that appear in (an instance I of) Sailors. Consider the instance $S1$ shown in Figure 4.1. The answer to this query obviously includes values that do not appear in $Dom(Q, S1)$.

Returning to the question of expressiveness, we can show that every query that can be expressed using a *safe* relational calculus query can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be **relationally complete**. A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

4.5 POINTS TO REVIEW

- The inputs and outputs of a query are relations. A query takes *instances* of each input relation and produces an instance of the output relation. **(Section 4.1)**
- A *relational algebra* query describes a procedure for computing the output relation from the input relations by applying relational algebra operators. Internally, database systems use some variant of relational algebra to represent query evaluation plans. **(Section 4.2)**
- Two basic relational algebra operators are selection (σ), to select subsets of a relation, and projection (π), to select output fields. **(Section 4.2.1)**

- Relational algebra includes standard operations on sets such as union ($\cup$), intersection ($\cap$), set-difference ($-$), and cross-product ($\times$). **(Section 4.2.2)**
- Relations and fields can be renamed in relational algebra using the renaming operator (ρ). **(Section 4.2.3)**
- Another relational algebra operation that arises commonly in practice is the join ($\bowtie$) —with important special cases of equijoin and natural join. **(Section 4.2.4)**
- The division operation ($/$) is a convenient way to express that we only want tuples where all possible value combinations—as described in another relation—exist. **(Section 4.2.5)**
- Instead of describing a query by how to compute the output relation, a *relational calculus* query describes the tuples in the output relation. The language for specifying the output tuples is essentially a restricted subset of first-order predicate logic. In *tuple relational calculus*, variables take on tuple values and in *domain relational calculus*, variables take on field values, but the two versions of the calculus are very similar. **(Section 4.3)**
- All relational algebra queries can be expressed in relational calculus. If we restrict ourselves to safe queries on the calculus, the converse also holds. An important criterion for commercial query languages is that they should be *relationally complete* in the sense that they can express all relational algebra queries. **(Section 4.4)**

EXERCISES

Exercise 4.1 Explain the statement that relational algebra operators can be *composed*. Why is the ability to compose operators important?

Exercise 4.2 Given two relations $R1$ and $R2$, where $R1$ contains N1 tuples, $R2$ contains N2 tuples, and N2 > N1 > 0, give the minimum and maximum possible sizes (in tuples) for the result relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for $R1$ and $R2$ that are needed to make the expression meaningful:

(1) $R1 \cup R2$, (2) $R1 \cap R2$, (3) $R1 - R2$, (4) $R1 \times R2$, (5) $\sigma_{a=5}(R1)$, (6) $\pi_a(R1)$, and (7) $R1/R2$

Exercise 4.3 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

1. Find the *name*s of suppliers who supply some red part.
2. Find the *sid*s of suppliers who supply some red or green part.
3. Find the *sid*s of suppliers who supply some red part or are at 221 Packer Ave.
4. Find the *sid*s of suppliers who supply some red part and some green part.
5. Find the *sid*s of suppliers who supply every part.
6. Find the *sid*s of suppliers who supply every red part.
7. Find the *sid*s of suppliers who supply every red or green part.
8. Find the *sid*s of suppliers who supply every red part or supply every green part.
9. Find pairs of *sid*s such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
10. Find the *pid*s of parts that are supplied by at least two different suppliers.
11. Find the *pid*s of the most expensive parts supplied by suppliers named Yosemite Sham.
12. Find the *pid*s of parts supplied by every supplier at less than $200. (If any supplier either does not supply the part or charges more than $200 for it, the part is not selected.)

Exercise 4.4 Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

1. $\pi_{sname}(\pi_{sid}(\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)$
2. $\pi_{sname}(\pi_{sid}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$
3. $(\pi_{sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$
4. $(\pi_{sid}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sid}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$
5. $\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sid,sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)))$

Exercise 4.5 Consider the following relations containing airline flight information:

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`,
distance: `integer`, *departs:* `time`, *arrives:* `time`)
Aircraft(*aid:* `integer`, *aname:* `string`, *cruisingrange:* `integer`)
Certified(*eid:* `integer`, *aid:* `integer`)
Employees(*eid:* `integer`, *ename:* `string`, *salary:* `integer`)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.

Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed. (See the exercises at the end of Chapter 5 for additional queries over the airline schema.)

1. Find the *eid*s of pilots certified for some Boeing aircraft.
2. Find the *name*s of pilots certified for some Boeing aircraft.
3. Find the *aid*s of all aircraft that can be used on non-stop flights from Bonn to Madras.
4. Identify the flights that can be piloted by every pilot whose salary is more than $100,000. (*Hint:* The pilot must be certified for at least one plane with a sufficiently large cruising range.)
5. Find the names of pilots who can operate some plane with a range greater than 3,000 miles but are not certified on any Boeing aircraft.
6. Find the *eid*s of employees who make the highest salary.
7. Find the *eid*s of employees who make the second highest salary.
8. Find the *eid*s of pilots who are certified for the largest number of aircraft.
9. Find the *eid*s of employees who are certified for exactly three aircraft.
10. Find the total amount paid to employees as salaries.
11. Is there a sequence of flights from Madison to Timbuktu? Each flight in the sequence is required to depart from the city that is the destination of the previous flight; the first flight must leave Madison, the last flight must reach Timbuktu, and there is no restriction on the number of intermediate flights. Your query must determine whether a sequence of flights from Madison to Timbuktu exists for *any* input Flights relation instance.

Exercise 4.6 What is *relational completeness*? If a query language is relationally complete, can you write any desired query in that language?

Exercise 4.7 What is an *unsafe* query? Give an example and explain why it is important to disallow such queries.

BIBLIOGRAPHIC NOTES

Relational algebra was proposed by Codd in [156], and he showed the equivalence of relational algebra and TRC in [158]. Earlier, Kuhns [392] considered the use of logic to pose queries. LaCroix and Pirotte discussed DRC in [397]. Klug generalized the algebra and calculus to include aggregate operations in [378]. Extensions of the algebra and calculus to deal with aggregate functions are also discussed in [503]. Merrett proposed an extended relational algebra with quantifiers such as *the number of*, which go beyond just universal and existential quantification [460]. Such generalized quantifiers are discussed at length in [42].