`not`, `and`, `or`, `exists`, and `forall` are used instead of the mathematical symbols denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), existential quantification ($\exists$), and universal quantification ($\forall$), respectively.

A general TRC expression is of the form

$$\{ \mathrm{T1}, \mathrm{T2}, \ldots, \mathrm{Tn} \mid \mathcal{F}(\mathrm{T1}, \mathrm{T2}, \ldots, \mathrm{Tn}) \},$$

where $\mathcal{F}$ is a TRC formula describing the properties that are to hold true on the data to be retrieved. The output schema of $\mathcal{F}$ is given by the attributes associated with the tuple variables T1, T2, ..., Tn.

The building block of a TRC formula is an atom consisting of a reference to a relation or a comparison of an attribute of a tuple variable to an attribute of another tuple variable or a domain constant. An atom forms the basis of a formula, which is built up from atoms using logical connectives (`not`, `and`, `or`) and quantification (`exists`, `forall`).

Let

`r` be a relation of degree n
`T` and `Ti` represent tuple variables
`ai` represent an attribute
`c` be a domain constant
$\theta$ be a comparison operator (<, =<, =, >, >=, <>), and assume that its operands are comparable by $\theta$

An atom is of the form

- `r(T)`
  TRUE when T is assigned a value forming a tuple in `r`.
- `Ti.am` $\theta$ `Tj.an`
  TRUE when `Ti.am` $\theta$ `Tj.an` is TRUE and otherwise FALSE.
- `T.ai` $\theta$ `c`
  TRUE when `T.ai` $\theta$ `c` is TRUE and otherwise FALSE.

A formula is composed of atoms using the following rules:

**(1)** An atom is a formula.
Its meaning is given by the truth value of the atom.

**(2)** Let $\mathcal{F}$, $\mathcal{F}1$, and $\mathcal{F}2$ be formulas; then the following are formulas:

- ($\mathcal{F}$)
  TRUE when $\mathcal{F}$ is TRUE and otherwise FALSE.
- `not` $\mathcal{F}$
  TRUE when $\mathcal{F}$ is FALSE and otherwise FALSE.



- $\mathcal{F}1$ and $\mathcal{F}2$
  TRUE when both $\mathcal{F}1$ and $\mathcal{F}2$ are TRUE and otherwise FALSE.
- $\mathcal{F}1$ or $\mathcal{F}2$
  TRUE when either $\mathcal{F}1$ or $\mathcal{F}2$ are TRUE and otherwise FALSE.

**(3)** Let $\mathcal{F}$(T) be a formula in which the variable T appears free. A variable is free if it is not quantified by an existential or universal quantifier. Then the following are formulas:

- `(exists T)`$\mathcal{F}$(T)
  TRUE if there exists a value assigned to T that makes $\mathcal{F}$(T) TRUE and otherwise FALSE.
- `(forall T)`$\mathcal{F}$(T)
  TRUE if any value that is assigned to T makes $\mathcal{F}$(T) TRUE and otherwise FALSE.

A valid TRC expression $\{\mathrm{T1}, \mathrm{T2}, \ldots, \mathrm{Tn} \mid \mathcal{F}(\mathrm{T1}, \mathrm{T2}, \ldots, \mathrm{Tn})\}$ has only the tuple variables T1, T2, ..., Tn free in $\mathcal{F}$. Free variables are global variables with respect to the TRC expression $\mathcal{F}$. Any other variable appearing in $\mathcal{F}$ is a local variable and is bound by its quantified declaration. The scope of the local variable is the quantified formula. The remainder of the chapter illustrates the TRC language by example.

The result of a TRC expression may be saved in an intermediate table. The syntax

```
intermediateTable := queryExpression;
```

assigns the result of the `queryExpression` to the `intermediateTable`. The attribute names for the schema of `intermediateTable` are derived from the `queryExpression`. However, if renaming of attributes is desired, then the syntax

```
intermediateTable(attr1, ..., attrn) := queryExpression;
```

provides for the renaming of the output schema of `queryExpression` to the schema given by the attribute list `attr1, ..., attrn`.

## 4.2 EXPRESSIVE POWER

Like DRC, the TRC language is relationally complete, since any relational algebra query can be expressed in TRC. The TRC language is introduced using the examples over the EMPLOYEE TRAINING enterprise, first illustrating the relational completeness of TRC using the fundamental operators and then describing the additional operators of relational algebra in TRC. In the examples, the names of the tuple variables are usually abbreviated to one or two characters, based on a mnemonic association with the table over which the tuple variable ranges. For the Employee Training enterprise examples, the following tuple variable naming convention is used:

- E refers to the `employee` relation.
- T refers to the `takes` relation.
- A refers to the `technologyArea` relation.
- C refers to the `trainingCourse` relation.

Although more descriptive tuple variable names are possible, this naming convention for tuple variables provides concise, yet readable, examples.

### 4.2.1 Fundamental Operators

Table 4.1 summarizes the TRC expressions for operations involving the fundamental relational algebra operators, which identify the required operations for effective retrieval of information from a relational database.

The examples over the Employee Training enterprise that illustrate the expression of the fundamental operators in TRC are summarized in Table 4.2.

**TABLE 4.1** TRC summary of fundamental relational algebra operators.

| Algebra | TRC |
|---|---|
| $\sigma_\theta$(r) | {R \| r(R) and $\theta$ } |
| $\pi_A$(r) | {R.ai . . . R.aj \| r(R)} |
| r ∪ s | {T \| r(T) or s(T) } |
| r − s | {T \| r(T) and not s(T) } |
| q × r | {Q, R \| q(Q) and r(R) } |

**TABLE 4.2** TRC summary of fundamental Employee Training queries.

| Query | √TRC |
|---|---|
| $Q_\sigma$ | `{ E | employee(E) and E.eSalary > 100000 };` |
| $Q_\pi$ | `{ E.eLast, E.eFirst, E.eTitle | employee(E)};` |
| $Q_\cup$ | `managers := { E.eID | employee(E) and E.eTitle='Manager' };`<br>`coaches := { E.eID | employee(E) and E.eTitle='Coach' };`<br>`{ T | managers(T) or coaches(T) };` |
| $Q_-$ | `managers := { E.eID | employee(E) and E.eTitle='Manager' };`<br>`takenCourse := { T.eID | takes(T) };`<br>`{ T | managers(T) and not takenCourse(T) };` |
| $Q_\times$ | `{ E.eID, C.cID | employee(E) and trainingCourse(C) };` |



Query $Q_\sigma$:

```
✓ { E | employee(E) and E.eSalary > 100000 };
```

The tuple variable E is the only variable that appears in the TRC formula. This variable is a free or global variable, since it is not quantified within the TRC formula. The atom `employee(E)` binds the variable E to a tuple in the `employee` relation. The atom `E.eSalary > 100000` then checks whether the employee's salary is greater than 100000. The output schema of the TRC expression consists of the attributes associated with the tuple variable E, which are the attributes of the table `employee(eID, eLast, eFirst, eTitle, eSalary)`.

Query $Q_\pi$:

```
✓ { E.eLast, E.eFirst, E.eTitle | employee(E)};
```

Dot notation projects the desired attributes of the tuple variable E that ranges over the `employee` table.

Query $Q_\cup$:

```
✓  managers :=
     { E.eID | employee(E) and E.eTitle='Manager' };
   coaches :=
     { E.eID | employee(E) and E.eTitle='Coach' };
   { T | managers(T) or coaches(T) };
```

The first expression finds the identification number of employees who are managers, using the global or free tuple variable E to range over the employee table, and then the value of the `eTitle` determines whether the employee is a manager. Similarly, the second expression finds the identification number of employees who are coaches. The third expression unions the compatible intermediate tables `managers` or `coaches`. The tuple variable T is free in the union TRC expression, getting its bindings from the operands of the disjunction. Therefore, the disjuncts of a valid disjunctive expression must have the same free variables.

Although the previous specification of the query used the general template for expressing a union in TRC, the query can be specified in one step:

```
✓ { E.eID | employee(E) and
     (E.eTitle = 'Manager' or E.eTitle = 'Coach') };
```

Query $Q_-$:

```
✓  managers :=
     { E.eID | employee(E) and E.eTitle='Manager' };
   takenCourse := { T.eID | takes(T) };
   { T | managers(T) and not takenCourse(T) };
```

The first intermediate table `managers` contains the `eID` of employees having the title `Manager`. The second intermediate table `takenCourse` contains the `eID` of employees that have taken a course, as given by the `takes` tables. The answer to the query returns the set of tuples `T` such that `T` forms a tuple in the `managers` table and does not form a tuple in the `takenCourse` table. An alternative specification of the query can be given as a single query:

```
√  { E.eID | employee(E) and E.eTitle='Manager' and
     not (exists T) (takes(T) and T.eID=E.eID) };
```

In this version of the query, the tuple variable `E` is a global variable, and `T` is a local variable, defined only within the scope of the negated formula.

Query $Q_{\times}$:

```
√  { E.eID, C.cID | employee(E) and trainingCourse(C) };
```

The simplifying assumption for the cartesian product operator in relational algebra requires that the operand relations do not have any attributes in common. In the TRC language, the same assumption for the result of a cartesian product must hold. The resulting relation schema must not contain duplicate attribute names. The specification of the query $Q_{\times}$ represents a cartesian product of the `employee` and `trainingCourse` tables in TRC. The `eID` and `cID` attributes of the tuple variables `E` and `C`, respectively, are specified as the result of the query to be consistent with the schema of the corresponding relational algebra example.

### 4.2.2 Additional Operators

Table 4.3 summarizes the illustrative examples over the EMPLOYEE TRAINING enterprise using the additional binary operators of relational algebra ($\cap$, $\bowtie_\theta$, $\bowtie$), which

**TABLE 4.3** TRC summary of additional EMPLOYEE TRAINING queries.

| Query | √TRC |
|---|---|
| $Q_{\cap}$ | `managers := { E.eID \| employee(E) and E.eTitle='Manager' };`<br>`takenCourse := { T.eID \| takes(T) };`<br>`{ T \| managers(T) and takenCourse(T) };` |
| $Q_{\bowtie_\theta}$ | `{ E, A \| employee(E) and technologyArea(A) and E.eID=A.aLeadID };` |
| $Q_{\bowtie}$ | `{ C.cTitle, T.tYear, T.tMonth, T.tDay \|`<br>`trainingCourse(C) and takes(T) and C.cID=T.cID };` |



are frequently used combinations of the fundamental operators. The division (÷) binary operator is not given in the table but delegated to a more detailed discussion later in this section.

Query $Q_{\cap}$:

```
√   managers :=
      { E.eID | employee(E) and E.eTitle='Manager' };
    takenCourse := { T.eID | takes(T) };
    { T | managers(T) and takenCourse(T) };
```

The intersection operation is achieved by using the same tuple variable name in both atoms referencing the operand relations. The query can also be specified in one step:

```
√  { E.eID | employee(E) and E.eTitle='Manager' and
     (exists T) (takes(T) and T.eID=E.eID) };
```

In this version of the query, the tuple variable `E` is a global variable, and `T` is a local variable defined only within the scope of the existential formula.

Query $Q_{\bowtie_\theta}$:

```
√  { E, A | employee(E) and technologyArea(A) and
     E.eID=A.aLeadID };
```

The global variables `E` and `A` range over the tables to be joined. The join condition `E.eID=A.aLeadID` joins the `employee` and `technologyArea` tables such that the employee `eID` is equal to the `aLeadID` of the technology area.

Query $Q_{\bowtie}$:

```
√  { C.cTitle, T.tYear, T.tMonth, T.tDay |
     trainingCourse(C) and takes(T) and C.cID=T.cID };
```

In DRC, using the same domain variable name in the positions that are to be joined results in a natural join. Since there is no shortcut available for the natural join in TRC, the attributes to be joined must be done so explicitly (e.g., `C.cID=T.cID`).

**Division.** Division is a complex binary relational algebra operator that finds those values in the first operand relation that are related to *all* of the values in the second operand relation. The same abstract division example (`abTable ÷ bTable`), where the schemas of the operand relations are `abTable(a,b)` and `bTable(b)`, demonstrates the expression of division in TRC. This illustration

...division in TRC using both universal and existential quantification, along with their corresponding truth tables.2

The abstract division example includes an a value in the result of the division, provided that the a value is in the abTable and for all possible B values, if B is a tuple in the bTable, then the a and b values are related by the abTable:

```
{ T.a | abTable(T) and (forall B) (bTable(B) implies
    (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) };
```

Why does the TRC specification look more complicated than its corresponding specification in the DRC language?

```
{ A | abTable(A, _) and (forall B) (bTable(B) implies
    abTable(A,B) ) };
```

This DRC specification is utilizing shortcuts of the DRC language. Without the shortcut, the DRC specification more closely resembles the TRC:

```
{ A | abTable(A, _) and (forall B) (bTable(B) implies
    (exists A1,B1) (abTable(A1,B1) and A1=A and B1=B) ) };
```

Since the implies operator is not formally defined in this version of TRC, the logically equivalent specification using not p or q instead of p implies q is

```
✓ { T.a | abTable(T) and (forall B) (not bTable(B) or
    (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) };
```

Table 4.4 shows the corresponding truth table. The forall formula must be true for all values of the variable B, including those B values that do not form a tuple in the bTable.

Consider a logically equivalent specification of division that uses only existential quantification. An a value is included in the result of the division, provided that the a value is in the abTable and it is not the case that there exists a b value that is in the bTable and is *not* related to the a value by the abTable:

```
✓ { T.a | abTable(T) and not (exists B) (bTable(B) and
    not (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) };
```

2 Although this exposition closely resembles the similar description in DRC, it is included to follow its derivation in terms of TRC.



**TABLE 4.4** TRC division: universal truth table.

| abTable(T) | b | not bTable(B) | (exists AB) | not bTable(B) or (exists AB) | forall B |
|---|---|---|---|---|---|
| (a1) | b1 | F | T | T | |
| | b2 | F | T | T | T |
| | b3 | T | F | T | |
| (a2) | b1 | F | F | F | |
| | b2 | F | T | T | F |
| | b3 | T | F | T | |
| (a3) | b1 | F | T | T | |
| | b2 | F | T | T | T |
| | b3 | T | T | T | |

This logically equivalent existential specification can be derived from the universal specification using the following logical equivalences:

- Given the universal specification

```
✓ { T.a | abTable(T) and (forall B) (not bTable(B) or
    (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) };
```

- Apply the logical equivalence: (forall D) $\mathcal{F}$(D) ≡ not (exists D) not $\mathcal{F}$(D)

```
✓ { T.a | abTable(T) and not (exists B) not (not bTable(B) or
    (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) };
```

- Apply DeMorgan's Law: not (p or q) ≡ not p and not q

```
✓ { T.a | abTable(T) and not (exists B) (bTable(B) and
    not (exists AB) (abTable(AB) and AB.a=T.a
    and AB.b=B.b) ) };
```

Table 4.5 shows the corresponding truth table.

### 4.2.3 Safety

By illustrating how the operators of relational algebra can be expressed in TRC, the previous examples have demonstrated the relational completeness of TRC. Although the result of a relational algebra expression is finite, not all TRC expressions produce a finite result. Consider the TRC expression that attempts to find employees who do not lead a technology area, and assume that there exists an intermediate table leads containing the IDs of the employees that lead technology areas:

```
{E | not leads(E)}
```

TABLE 4.5 TRC division: existential truth table.

| abTable(T) | b | bTable(B) | not (exists AB) | bTable(B) and not (exists AB) | not (exists B) |
|---|---|---|---|---|---|
| (a1) | b1 | T | F | F | |
| | b2 | T | F | F | |
| | b3 | F | T | F | T |
| (a2) | b1 | T | T | T | |
| | b2 | T | F | F | |
| | b3 | F | T | F | F |
| (a3) | b1 | T | F | F | |
| | b2 | T | F | F | |
| | b3 | F | F | F | T |

There are infinitely many tuple values that are not in the `leads` table. To fi[nd] those employees who do not lead a technology area, first limit the tuple varia[ble] `E` to values of the `employee` table and then check whether the employee ID val[ue] appears in the `leads` table:

```
✓   { E | employee(E) and not (exists L)
        (leads(L) and L.eID=E.eID) };
```

A *safe* TRC expression guarantees a finite result by limiting (either directly [or] indirectly) the values of a tuple variable (T) by its appearance in a positive at[om] (`r(T)`). The EMPLOYEE TRAINING examples illustrating the relational completeness [of] the TRC language are safe. Only *safe* TRC expressions can be realized in relatio[nal] algebra. Therefore, relational algebra and safe TRC (and safe DRC) are equivale[nt] in expressive power.

## 4.3 EXAMPLE QUERIES

The example queries over the EMPLOYEE TRAINING enterprise illustrate vari[ous] types of relevant queries and the use of the TRC language to express these quer[ies]. Similar to DRC, the TRC language examples use the convention to limit a vari[able] before its value is referenced in a left-to-right reading of the relational calc[ulus] expression. Since safe TRC is equivalent in expressive power to relational alge[bra] and safe DRC, TRC does not support aggregation. Therefore, queries Q4 and [Q5] must be answered creatively. The inventive solutions for these queries are sim[ilar] in concept to the DRC solutions. By finding employees who took two courses [in] two different technology areas, query Q4 retrieves the employees who took co[urses] in more than one technology area. Query Q5 finds the minimum salary emplo[yee] by selecting those employees that do not have a salary that is greater than an[other] ... Q6 finds the employees who have tak[en]



least one course in the database technology area and have taken all database courses. The verification of taking at least one database course is required to produce correct results when the `dbCourse` table is empty.

---

√Q1: What training courses are offered in the 'Database' technology area?

```
(cID, cTitle, cHours)

dbCourse :=
 { T.cID, T.cTitle, T.cHours | trainingCourse(T) and (exists A)
   (technologyArea(A) and A.aID = T.areaID and
   A.aTitle = 'Database') };
```

---

√Q2: Which employees have taken a training course offered in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)

dbEmployee :=
{ E.eID, E.eLast, E.eFirst, E.eTitle | employee(E) and
        (exists T,D) (takes(T) and dbCourse(D) and
        T.eID=E.eID and T.cID=D.cID) };
```

---

√Q3: Which employees have not taken any training courses?

```
(eID, eLast, eFirst, eTitle)

{ E.eID, E.eLast, E.eFirst, E.eTitle | employee(E) and
    not (exists T) (takes(T) and T.eID=E.eID) };
```

---

√Q4: Which employees took courses in more than one technology area?

```
(eID, eLast, eFirst, eTitle)

{ E.eID, E.eLast, E.eFirst, E.eTitle |
    employee(E) and (exists T1,T2,C1,C2)
        (takes(T1) and T1.eID=E.eID and
        takes(T2) and T2.eID=E.eID and
        trainingCourse(C1) and T1.cID=C1.cID and
        trainingCourse(C2) and T2.cID=C2.cID and
        C1.areaID <> C2.areaID) };
```

√Q5: Which employees have the minimum salary?

```
(eID, eLast, eFirst, eTitle, eSalary)

{ E | employee(E) and
  not (exists S) (employee(S) and S.eSalary < E.eSalary) };
```

√Q6: Which employees took all of the training courses offered in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)

{ E.eID, E.eLast, E.eFirst, E.eTitle | employee(E) and
    (exists B)(dbEmployee(B) and B.eID=E.eID) and
    not (exists D)(dbCourse(D) and
        not (exists T) (takes(T) and T.eID=E.eID and
        T.cID=D.cID) ) };
```

## 4.4 SUMMARY

TRC is a relationally complete language that declaratively specifies the properties of the data to be retrieved, not how to retrieve the data. The variables of the TRC language range over tuples of relations, and the names of attributes associated with the tuple are referenced using dot notation. The TRC *by-name* syntax forms the foundation of SQL, which is described in the next chapter.

## DISCUSSION

Some may prefer the TRC *by-name* syntax over the positional syntax offered by the theoretical DRC. However, TRC appears somewhat verbose when compared with DRC. DRC offers pragmatic shortcuts for equality constraints at the domain level such as natural joins and domain constants. The only shortcut available in both TRC and DRC is the ability to use the same variable names in compatible operations such as union, difference, and intersection.

## EXERCISES

*Don't forget that you may use intermediate tables to break down a query into multiple steps. Since TRC and DRC are similar languages, the first exercise presents new queries over the* EMPLOYEE TRAINING *schema to encourage the development of TRC expressions that are not based on existing DRC solutions.*





√ Use the WinRDBI educational tool to *check* the answers to these marked exercises.

√**4.1** Answer the following queries in TRC over the EMPLOYEE TRAINING schema:

```
employee(eID, eLast, eFirst, eTitle, eSalary)
technologyArea(aID, aTitle, aURL, aLeadID)
trainingCourse(cID, cTitle, cHours, areaID)
takes(eID, cID, tYear, tMonth, tDay)
```

**(a)** Which employees took training courses in a given month?

Look at the database instance and choose a month and year that will not yield an empty result:

```
(eID, eLast, eFirst, eTitle)
```

**(b)** Which employees have not taken all of the training courses in the 'Database' technology area?

```
(eID, eLast, eFirst, eTitle)
```

**(c)** Which employee leads have taken more than one training course in the technology area that they lead?

```
(eID, eLast, eFirst, eTitle)
```

**(d)** Which training courses in the 'Database' technology area have the maximum number of hours?

```
(cID, cTitle, cHours)
```

**(e)** Which employees have taken *all* training courses offered in the year 2001?

```
(eID, eLast, eFirst, eTitle)
```

√**4.2** Answer Exercise 3.1 in TRC. Do not translate the DRC solutions to TRC! Verify that you obtained the same set of results on the same database instance for each query. Compare your TRC solutions with your DRC solutions. Do you prefer one relational calculus language to the other? Why?

√**4.3** Answer Exercise 2.1 in TRC. Verify your results and compare your solutions.

√**4.4** Answer Exercise 2.2 in TRC. Verify your results and compare your solutions.

**4.5** The most common mistake in constructing a division query in TRC is the lack of use of the implication within the universally quantified formula. Consider the following specification of the division query:

```
{ T.a | abTable(T) and (forall B) (bTable(B) and
    (exists AB) (abTable(AB) and AB.a=T.a and AB.b=B.b) ) }
```