1 Exercises

With two exceptions, these exercises are copies of those given at the ends of Chapters 1-7 in *An Introduction to Relational Database Theory.* The exercises using *Rel* given with some of those chapters are also included. The first exception is Exercise 7 for Chapter 7, which I have replaced by a precise, detailed specification for a comprehensive database design. The second is a set of additional exercises using *Rel,* exploring virtual relvars and user-defined type definitions.

In this second edition the only changes are to use the syntax for Version 2 of **Tutorial D**, now supported by *Rel,* and to correct a number of errors in the first edition (including some particularly bad ones in Section 1.4, Exercise 2).

1.1 Exercise for Chapter 1, Introduction

Consider the following table (from Figure 1.5 of the book)

A	B	A
1	2	3
4		5
6	7	8
9	9	?
1	2	3

Give three reasons why it cannot possibly represent a relation.

1.2 Exercises for Chapter 2, Values, Types, Variables, Operators

Complete sentences 1-10 below, choosing your fillings from the following:

=, :=, ::=, argument, arguments, body, bodies, `BOOLEAN`, cardinality, `CHAR`, `CID`, degree, denoted, expressions, false, heading, headings, `INTEGER`, list, lists, literal, literals, operator, operators, parameter, parameters, read-only, set, sets, `SID`, true, type, types, update, variable, variables.

In 1–5, consider the expression `X = 1 OR Y = 2`.

1. In the given expression, = and `OR` are _____ whereas `X` and `Y` are _____ references.

2. `X` and `1` denote _____ to an invocation of _____.

3. The value _____ by the given expression is of _____ `BOOLEAN`.

4. `1` and `2` are both _____ of _____ `INTEGER`.

5. The operators used in the given expression are _____ operators.

In 6–10, consider the expression `RELATION { X SID, Y CID } { }`.

6. It denotes a relation whose _____ is zero and whose _____ is two.

7. It is a relation _____.

8. The declared type of `Y` is _____.

9. In general, the heading of a relation is a possibly empty _____ of attributes and its body is a possibly empty _____ of tuples.

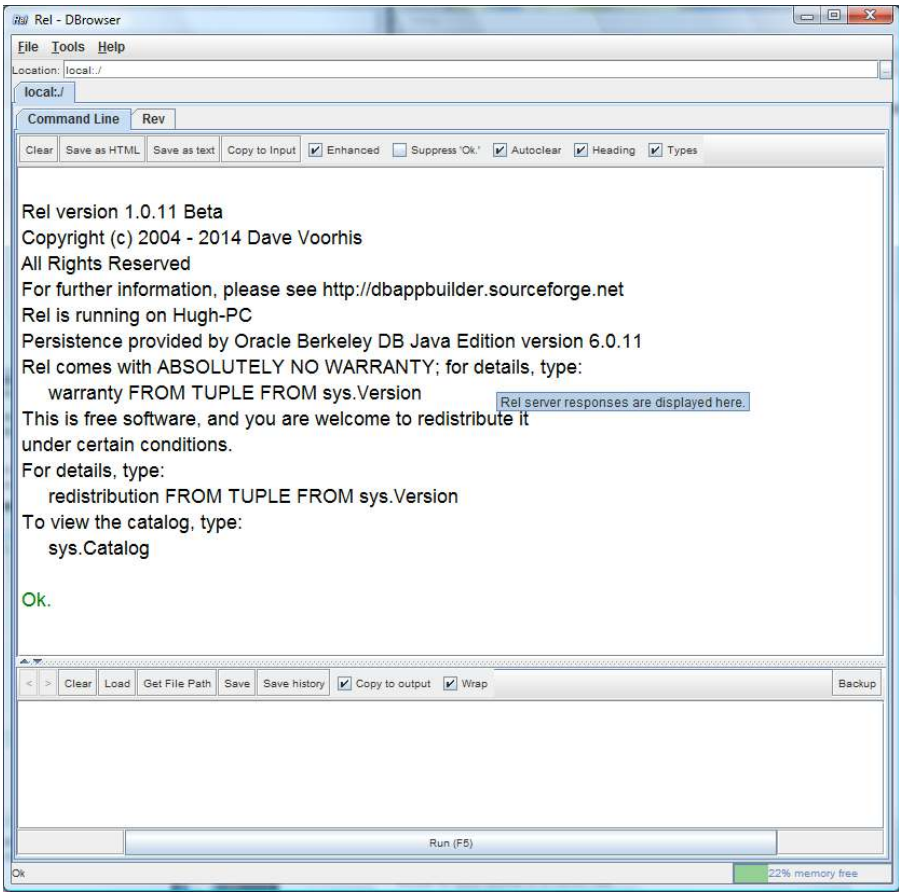
10. It is _____ that the assignment `RV __ RELATION { X SID, Y CID } { }` is legal if the _____ of `RV` is `{ Y CID, X SID }`, _____ that it is legal if the _____ of `RV` is `{ A SID, B CID }`, _____ that it is legal if the _____ of `RV` is `{ X CID, Y SID }`, and _____ that it is legal if the _____ of `RV` is `{ X CHAR, Y CHAR }`.

Getting Started with *Rel*

After you have downloaded and installed *Rel* from http://dbappbuilder.sourceforge.net/Rel.html, work through the following exercises. From number 7 onwards they involve constructs introduced in Chapter 4. You might prefer to wait until you have studied that chapter but on the other hand a little hands-on experience might help you to understand that chapter when you come to it.

1. Start up *Rel*'s DBrowser. DBrowser is the general-purpose client application provided by *Rel* for evaluating **Tutorial D** expressions and executing **Tutorial D** statements entered by the user.

2. Familiarise yourself with the way of working and the things you can do in *Rel*. You should be looking at a window something like this (which was obtained in Windows Vista):



- Note the layout of the window: a lower pane into which you can type statements to be executed, an upper pane in which results are displayed, and the movable horizontal bar between the panes.
- Note the ▲ and ▼ at the left-hand end of the horizontal bar, allowing you to let one or the other pane occupy the whole window for a while.
- See what is available on the Tools menu and perhaps choose your preferred font.
- Note the < and > to the left of the menu on the input (lower) pane. These are greyed out initially but after you have executed a couple of statements you will be able to use them to recall previously executed statements to the input pane.
- Note the toolbars on both panes. As you do the exercises, decide which options suit you best. Note that you can save the contents of either pane into a local file, and that you can load the contents of a local file into the input area.
- Note the check boxes on the right of the toolbars. They are fairly self-explanatory, apart from “Enhanced”, which we will examine later.

- The box at the top of the upper pane, labelled "Location:", identifies the directory containing the database you are working with. You can switch to another directory by clicking on the little button to the right of the box, labelled with three dots (…).
- The button on the right, labelled "Backup". This produces a *Rel* script that can be used to recreate the entire database in its current state.

3. Type the following into the lower pane:

```
output 2+2 ;
```

Execute what you have typed, either by clicking on Evaluate (F5) shown at the bottom of the window or by pressing F5.

Now delete the semicolon and try executing what remains. (If necessary, use the < button on the lower pane to recall the statement.) You will see how *Rel* handles errors.

Now strike out the word `output` and do not put back the semicolon. What happens when you execute that? (i.e., just `2+2`).

You have now learned:

- that in *Rel* (as in **Tutorial D**) every executable *command* (or statement) is terminated by a semicolon;
- that *Rel* allows you to obtain the result of evaluating an *expression* by using an `output` statement;
- that *Rel* treats an attempt to 'execute' an expression *x* as shorthand for the statement `output x ;` — the absence of the semicolon signals to *Rel* that you are using this convenient shorthand.

4. This exercise is merely to alert you to a certain awkwardness in *Rel* that has no real importance but might cause you to waste a lot of time if you are not warned about it. It's the same as Step 3 except that instead of `2+2` you type `2+2.0`. Look closely at what happens. It doesn't work!

Rel, like some other languages, treats `INTEGER` and `RATIONAL` as distinct types. If you want to do arithmetic on rational numbers, both operands must be rational numbers. Literals denoting rational numbers are distinguished from those denoting integers by the presence of a decimal point, and *Rel* follows the convention in the English-speaking community of using a full stop for this purpose (as opposed to the comma that is used throughout most of Europe, for example).

Now try this: `1/2` (i.e., the integer 1 divided by the integer 2). And then this: `1.0/2.0`.

You have now learned that (a) the operands of dyadic arithmetic operators in *Rel* must be of the same type, and (b) the type of the result of an invocation of such an operator is always of the same type as the operands. **Tutorial D** is silent on such issues, because they are orthogonal to what **Tutorial D** is really intended for (teaching relational theory). But every implementation of **Tutorial D** has to address them somehow.

Fortunately, arithmetic is orthogonal to relational theory and there is no need for us to be bothered by *Rel*'s behaviour here. You have possibly already learned that the same problems do not arise in SQL, where `1/2`, `1/2.0` and `1.0/2.0` are all equivalent, in spite of the fact that `INTEGER` and `REAL` (SQL's counterpart of **Tutorial D**'s `RATIONAL`) are also distinct types in SQL.

5. Now try the following compound statement:

```
begin ;
VAR x integer init(0) ;
x := x + 1 ;
output x ;
end ;
```

Why do we have to write `output x ;` in full here, instead of just `x`?

Now write the fourth line in uppercase: `OUTPUT X ;` What happens?

Try `OUTPUT x ;` instead. What have you learned about *Rel*'s rules concerning *case sensitivity*?

6. Now you can start investigating *Rel*'s support for relations (though not relational databases yet). First, see how *Rel* displays a relation (i.e., the result of evaluating a relation expression) in its upper pane. *Rel* supports two styles of presentation, depending on whether the "Enhanced" option is checked.

With "Enhanced" unchecked (it is usually checked to start with), get *Rel* to evaluate the following relation expression (a literal which we shall call `enrolment`):

```
RELATION {
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
}
```

See Section 2.9. Look closely at the output. Is it identical to the input?

Next, without altering the contents of the lower pane, turn "Enhanced" back on. Note the effect on the display in the output pane.

Now delete all the tuple expressions, leaving just `RELATION { }`. What happens when *Rel* tries to evaluate that?

Now use < to recall the original `RELATION` expression to the input pane and re-evaluate it with "Enhanced" *off*. Use copy-and-paste to copy the result to the input pane, then delete all the `TUPLE` expressions, to leave this:

```
RELATION {StudentId CHARACTER, CourseId CHARACTER,
          Name CHARACTER} { }
```

Study the result of that in the output pane, first with "Enhanced" off, then with it on.

What conclusions do you draw from all this, about *Rel* and **Tutorial D**?

From now on you can run with "Enhanced" either on or off, according to your own preference.

Next, enter the following literal, perhaps by using the < button to recall `enrolment` and editing it:

```
RELATION {
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' }
 }
```

Before you press Evaluate (F5), think about what you expect to happen. Does the result meet your expectation? How do you explain it?

Use < again to recall the `enrolment` literal. Insert `WITH ( enrolment :=` at the beginning and add `) : enrolment` at the end, to give:

```
WITH ( enrolment :=
RELATION {
 TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
 TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
 TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
 TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
 TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} ) : enrolment
```

and evaluate that.

How do you understand what you have just done? (`WITH` isn't described in the book. In case you aren't clear, try this in *Rel*: `WITH ( four := 2+2, eight := four+four ) : eight + four`. Note carefully that the introduced names, `four` and `eight`, are local only.)

By inspection of `enrolment` only, write down all the cases you can find of two students such that there is at least one course they are both enrolled on.

7. For this exercise you will need to continue using < to recall your previous command (now including the definition of the introduced name `enrolment`) and overtype as necessary. Use `enrolment` to investigate the relational operator known as **projection (*see Chapter 4, Section 4.6*).** The projection of a given relation over a specified subset of its attributes yields another relation. In **Tutorial D** a projection is specified by writing a list of attribute names, enclosed in braces `{}` and separated by commas, after the operand relation. The key words `ALL BUT` can optionally precede the list of attribute names, inside the braces.

   How many distinct projections can be obtained from `enrolment`? Obtain as many of these as you wish, trying both the 'inclusion' method and the 'exclusion' method using `ALL BUT`.

8. Still using `enrolment`, investigate the relational operator known as **rename (*see Chapter 4, Section 4.5*).** The renaming of a given relation returns that relation with one or more of its attributes renamed. In **Tutorial D** a renaming is specified by writing `RENAME { old AS new, ... }` after the operand relation.

   At the moment you should have this in your input pane:

```
WITH ( enrolment :=
RELATION {
 TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
 TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
 TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
 TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
 TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} ) : enrolment
```

   Replace the single word (`enrolment`) that follows the colon by a renaming of `enrolment` such that the result has attribute name `SID1` instead of `StudentId`, `N1` instead of `Name`, and is otherwise the same as `enrolment` itself. Replace the : that ends the `WITH` specification by `E1 :=` and add `: E1` at the end. The result should look like this:

```
WITH ( enrolment :=
RELATION {
 TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
 TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
 TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
 TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
 TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} , E1 := <your renaming of enrolment, as specified> ) :
E1
```

Evaluate that to check that you wrote the renaming correctly.

9. Now replace the : by , `E1 :=` and this time add a similar renaming of `enrolment`, using `SID2` and `N2` instead of `SID1` and `N1` for the new attribute names, and add : `E1 JOIN E2` at the end. You are investigating the operator called `JOIN` ***(see Chapter 4, Section 4.4)***.

   How do you interpret the result? How many tuples does it contain? Replace the key word `JOIN` by `COMPOSE` ***(see Chapter 5, Section 5.2)***. How do you interpret *this* result? How many tuples are there now? How do you account for the difference?

10. Add `WHERE NOT ( SID1 = SID2 )` to end of the expression you evaluated in Step 9 (***see Chapter 4, Section 4.7***). Examine the result closely. Now place parentheses around `E1 COMPOSE E2` and evaluate again. Confirm that you get the same result.

    Repeat the experiment, replacing `WHERE NOT ( SID1 = SID2 )` by `{ SID1 }`. Do you get the same results this time? If not, why not?

    What does all this tell you about operator precedence rules in **Tutorial D**?

    Why was it probably a good idea to add that `WHERE` invocation? Did it completely solve the problem? If not, can you think of a better solution?

    What connection, if any, do you see between this exercise and Exercise 6?

11. Load the file `OperatorsChar.d`, provided in the Scripts subdirectory of the *Rel* program directory, and execute it. Now you have the operators used in Example 2.4, among others. Give appropriate type definitions for types `NAME` and `CID`. Notice that the operator `TO_UPPER_CASE` is available for converting a given string to its upper-case counterpart. You might like to try using this operator to define a constraint for type `NAME` to ensure that all names begin with a capital letter.

12. Close *Rel* by clicking on File/Exit.

1.3 Exercises for Chapter 3, Predicates and Propositions

Consider again the relation shown as the current value of ENROLMENT in Figure 1.2:

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

For each of the following propositions, state whether it is true or false, basing your conclusions on this relation:

1. There exists a course *CourseId* such that some student named Anne is enrolled on *CourseId*.

2. Every student with StudentId S1 who is enrolled on some course is named Anne.

3. Every student who is enrolled on course C4 is named Anne.

4. Some student who is enrolled on course C4 is named Anne.

5. There are exactly 5 students who are enrolled on some course.

6. It is not the case that there is no course on which no student who is enrolled on some course but is not named Boris is not enrolled.

7. There are exactly 10 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.

8. There are exactly 3 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.

9. If a student named Eve is enrolled on course C1, then student S1 is named Adam.

10. If student S1 is named Anne, then S1 is enrolled on course C2.

1.4 Exercises for Chapter 4, Relational Algebra – The Foundation

1. Recall that *r1* `TIMES` *r2* requires *r1* and *r2* to have no common attributes, in which case it is equivalent to *r1* `JOIN` *r2*. Why would it be a bad idea to *require* `TIMES` to be used in place of `JOIN` in such cases?

2. Given the following relvars:

```
VAR Cust BASE RELATION {C# CHAR, Discount RATIONAL} KEY {C#};
VAR Orders BASE RELATION {O# CHAR, C# CHAR, Date DATE}
                        KEY {O#};
VAR OrderItem BASE RELATION {O# CHAR, P# CHAR, Qty INTEGER }
                           KEY {O#, P#};
VAR Product BASE RELATION {P# CHAR, Unit_price RATIONAL}
                          KEY {P#};
```

The price of an order item can be calculated by the formula:

```
CAST_AS_RATIONAL(Qty)*Unit_price*(1.0-(Discount/100.0))
```

Write down a relation expression to yield a relation with attributes `O#, P#,` and `PRICE`, giving the price of each order item.

3. Given:

```
VAR Exam_Marks BASE RELATION { StudentId SID,
                               CourseId CID,
                               Mark INTEGER}
                            KEY { StudentId, CourseId };
```

Write down a relational expression to give, for each pair of students sitting the same exam, the absolute value of the difference between their marks. Assume you can write `ABS(`x`)` to obtain the absolute value of x.

4. State the value of

   (a) *r* NOT MATCHING TABLE_DEE
   (b) *r* NOT MATCHING TABLE_DUM
   (c) *r* NOT MATCHING *r*
   (d) (*r* NOT MATCHING *r*) NOT MATCHING *r*
   (e) *r* NOT MATCHING (*r* NOT MATCHING *r*)

   Is NOT MATCHING associative? Is it commutative?

5. (Repeated from the body of the chapter) Which operator, in the list given in Section 4.11, **Concluding Remarks**, can be dispensed with without sacrificing relational completeness? How can it be defined in terms of the other operators?

6. (Repeated from the body of the chapter) Investigate the completeness of an algebra that includes MINUS in place of NOT MATCHING by attempting to define NOT MATCHING in terms of MINUS and the other operators.

7. The chapter briefly mentions the operator MATCHING but defers its detailed description to Chapter 5. Before you read that chapter, define *r1* MATCHING *r2* in terms of the operators described in Chapter 4.

Working with a Database in *Rel*

1. Start up *Rel*.

2. Figure 4.13 shows the supplier-and-parts database from Chris Date's *Introduction to Database Systems (8th edition),* as shown on the inside back cover of that book (except that the attribute names there are in upper case).

S

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	Pname	Color	Weight	City
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Figure 4.13: The suppliers-and-parts database

Execute a **Tutorial D** `VAR` statement for each of `S, P` and `SP`. Use `INTEGER` as the declared type for `STATUS` and `QTY, RATIONAL` for `WEIGHT,` and `CHAR` for all the other attributes. Feel free to use lower case or mixed case to suit your own taste for attribute and relvar names, but do not otherwise change any of the given names.

Tutorial D requires at least one key constraint to be specified for each relvar. One key for each for `S`, `P` and `SP` is shown by underlining the relevant attribute names in the table. No other key constraints are needed.

"Populate" (as they say) each relvar with the values shown in Date's tables. There are several ways of achieving this. Choose whichever you prefer from the following:

a. Include an `INIT ( ... )` specification in the `VAR` statement, after the heading and before the `KEY` specification. Inside the parens, write a `RELATION { ... }` expression, using a `TUPLE` expression for each required tuple, as in the enrolment literal used in the *Rel* exercises for Chapter 2.

b. Execute the `VAR` statement without an `INIT ( ... )` specification. The implied initial value is the empty relation of the appropriate type. You can see this by asking *Rel* for the current value of the relvar. For example, to get the current value of `S`, just type `S` into the lower pane and click `Run (F5)`.
Now use an assignment statement of the form
`varname := relation-expression`
to populate the relvar. Check that *Rel* has indeed assigned the correct value to it.

c. Use *Rel* `INSERT` statements to populate the relvar piecemeal, perhaps one tuple at a time. Having typed in the first `INSERT` statement. Here is the general form of an `INSERT` statement to insert a single tuple:

```
INSERT varname RELATION { TUPLE { ... } } ;
```

Note that the source operand is still a relation, not just a tuple, hence the need to enclose the `TUPLE` expression inside `RELATION { }`.

3. Informally, we refer to `S` as suppliers, `P` as parts and `SP` as shipments. Predicates for these relvars are:

`S:` Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*.
`P:` Part *P#* is named *Pname*, is coloured *Color*, weighs *Weight* and is located in city *City*.
`SP:` Supplier *S#* ships part *P#* in quantities of *Qty*.
What, then, is the predicate for the expression `S JOIN SP JOIN P`?
What do you expect to be the result of that expression?
What is its degree?
Does *Rel* give the result you expected? Explain what you see.

4. Attempt to insert a tuple into `SP` with supplier number S1, part number P1 and quantity 100. Explain the result of your attempt.

5. Get *Rel* to evaluate each of the following expressions. For each one, write down the corresponding predicate and also give an informal interpretation of the query in the style of those given in Exercise 6 below.

a. `SP WHERE P# = 'P2'`

b. `S { ALL BUT Status }`

c. `SP { S#, Qty }`

d. `P NOT MATCHING ( SP WHERE S# = 'S2' )`

e. `S MATCHING ( SP WHERE P# = 'P2' )`

f. `S { City } UNION P { City }`

g. `S { City } MINUS P { City }`

h.
```
( ( S RENAME { City AS SC } ) { SC } ) JOIN
( ( P RENAME { City AS PC } ) { PC } )
```

6. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:

   a. Get all shipments.
   b. Get supplier numbers for suppliers who supply part P1.
   c. Get suppliers with status in the range 15 to 25 inclusive.
   d. Get part numbers for parts supplied by a supplier in Paris.
   e. Get part numbers for parts not supplied by any supplier in Paris.
   f. Get city names for cities in which at least two suppliers are located.
   g. Get all pairs of part numbers such that some supplier supplies both of the indicated parts.
   h. Get supplier numbers for suppliers with a status lower than that of supplier S1.
   i. Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

1.5 Exercises for Chapter 5, Building on The Foundation

1. (Repeated from the body of the chapter) What can you say about the result of *r1* `COMPOSE` *r2* when *r1* and *r2* have identical headings? For example, what is the result of `IS_CALLED COMPOSE IS_CALLED`?

2. (Repeated from the body of the chapter) Is `COMPOSE` associative? In other words, is (*r1* `COMPOSE` *r2*) `COMPOSE` *r3* equivalent to *r1* `COMPOSE` (*r2* `COMPOSE` *r3*)? If so, prove it; if not, show why.

3. What can you say about the result of *r1* `MATCHING (` *r2* `MATCHING` *r1* `)`?

4. (Repeated from the body of the chapter) Does the aggregate operator `AVG` have a basis operator? If so, define it.

5. Suppose an aggregate operator `PRODUCT` is defined, with arithmetic multiplication as its basis operator. What is the result of `PRODUCT(`*r,x*`)` if *r* is empty?

6. (Repeated from the body of the chapter) Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations being ungrouped on?

7. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:

   a. Get the total number of parts supplied by supplier S1.
   b. Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.
   c. Get part numbers for parts supplied by all suppliers in London.
   d. Get supplier numbers and names for suppliers who supply all the purple parts.
   e. Get all pairs of supplier numbers, S*x* and S*y* say, such that S*x* and S*y* supply exactly the same set of parts each.
   f. Write a truth-valued expression to determine whether all supplier names are unique in `S`.
   g. Write a truth-valued expression to determine whether all part numbers appearing in `SP` also appear in `P`.

1.6 Exercises for Chapter 6, Constraints and Updating

1. (Repeated from the body of the chapter).

   a. An implication of `KEY { ALL BUT }` is that no other key can possibly exist for the relvar it applies to. Why is this so?

   b. An implication of `KEY { }` is that no other key can possibly exist for the relvar it applies to. Why is this so?

2. Suppose the relvar definition for `COURSE` is extended to include an attribute `MaxExamMark`, whose value in each tuple is the maximum mark obtainable for that course's exam. `{StudentId, CourseId}` is a foreign key in `EXAM_MARK`, referencing `IS_ENROLLED_ON`. A constraint is needed to ensure that no student is awarded a mark greater than the relevant maximum.

   a. Write a **Tutorial D** `CONSTRAINT` statement to address this requirement, where the constraint condition is an invocation of `IS_EMPTY`.
   b. Complete the following statement to make it equivalent to the one you wrote for part (a):

```
CONSTRAINT … AND(EXAM_MARK, … ) ;
```

3. Now suppose that instead of there being a recorded maximum mark of each exam the maximum score for each question in each exam is recorded in the following relvar:

```
VAR EXAM_QUESTION BASE RELATION { CourseId CID,
      Question# INTEGER, MaxMark INTEGER }
      KEY { CourseId, Question# } ;
```

For each course, the exam questions are supposed to be numbered sequentially, starting at 1.

a. Write a **Tutorial D** `CONSTRAINT` statement to address this requirement.
b. Suppose the questions are subdivided into parts, a, b, c and so on, up to a maximum of six parts, and maximum marks are given for each part rather than for each question. Again, the parts for each question must be "numbered" sequentially, starting at a. Write a **Tutorial D** `CONSTRAINT` statement to address *this* requirement.
c. Devise shorthands, in the style of **Tutorial D**, for expressing constraints of the kinds found in your solutions to a. and b.

4. Using *Rel*, with the suppliers-and-parts database set up for the *Rel* exercises given at the end of Chapter 4, write **Tutorial D** integrity constraints to express the following requirements:

a. Every shipment tuple must have a supplier number matching that of some supplier tuple.
b. Every shipment tuple must have a part number matching that of some part tuple.
c. All London suppliers must have status 20.
d. No two suppliers can be located in the same city.
e. At most one supplier can be located in Athens at any one time.
f. There must exist at least one London supplier.
g. The average supplier status must be at least 10.
h. Every London supplier must be capable of supplying part P2.

1.7 Exercises for Chapter 7, Database Design I: Projection-Join Normalization

1. (Repeated from the body of the chapter). The predicate for `WIFE_OF_HENRY_VIII` is "The first name of the *Wife#*-th wife of Henry VIII is *FirstName* and her last name is *LastName* and *Fate* is what happened to her." Write an appropriate predicate for the following expression:

```
WIFE_OF_HENRY_VIII { Wife#, FirstName }
JOIN
WIFE_OF_HENRY_VIII { LastName, Fate }
```

2. Consider the following declarations:

```
VAR C1_EXAM_MARK BASE
    INIT ( EXAM_MARK WHERE CourseId = CID('C1') )
     KEY { StudentId } ;

CONSTRAINT C1_only
    AND ( C1_EXAM_MARK, CourseId = CID('C1') ) ;
```

(Recall that AND is the aggregate operator mentioned in Chapter 5, evaluating to TRUE if and only if the given condition evaluates to TRUE for every tuple of the given relation.)

a. Explain why C1_EXAM_MARK is not in BCNF.
b. Assume that similar relvars are defined for every course, except that this time there are no CourseId attributes. Describe how a query could be expressed to give the course identifier and mark for every exam taken by student S1.

3. In Section 7.5 of the chapter, under the heading **Functional Dependencies**, the following eight theorems are given concerning FDs.

1. Reflexivity: If *B* is a subset of *A*, then $A \rightarrow B$
2. Augmentation: If $A \rightarrow B$, then $A \cup C \rightarrow B \cup C$
3. Transitivity: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
4. Self-determination: $A \rightarrow A$
5. Decomposition: If $A \rightarrow B$ and *C* is a subset of *B*, then $A \rightarrow C$ and $A \rightarrow B - C$
6. Union: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow B \cup C$
7. Composition: If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup C \rightarrow B \cup D$
8. Unification: If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup (C - B) \rightarrow B \cup D$

Taking the first three as axioms, prove theorems 4 to 8.

4. (Repeated from the body of the chapter). Consider relvar SCDF with attributes S (for student), C (for course), D (for department), and F (for faculty). Assuming that the set {{C} → {D}, {D} → {F}} is a minimal cover for the FDs in SCDF, prove that {S,C} is a key of SCDF.

5. (Repeated from the body of the chapter). Assume that {{W,X} → {Y,Z}, {Y} → {X}} is a minimal FD cover for the FDs in relvar WXYZ. Prove that {W,X} and {W,Y} are both keys of WXYZ.

6. The heading of relvar R1 consists of attributes named a, b, c, d, e, f, g, and h. The following set of FDs is a cover for those that hold in R1:

FD1: {a,b} → {c}
FD2: {a,b} → {d}
FD3: {b} → {e}
FD4: {c} → {f}
FD5: {g} → {h}
FD6: {d} → {b, e}

a. Describe the single change required to derive an irreducible cover from the given set.
b. Describe the single change required to derive a minimal cover from your answer to a.
c. Explain why `R1` is not in Boyce-Codd normal form (BCNF).
d. Decompose `R1` into an equivalent set of BCNF relvars. Name your relvars `R2`, `R3`, and so on and for each one list its attribute names and state its key(s). For example: `R3{c,d,e} KEY{d} KEY{c,e}` if you think this relvar with those two keys is part of the solution.

7. *This replaces the Exercise 7 given in Chapter 7, giving you a more precise and detailed specification to work from. It's best done after a study of Chapter 8, in particular Section 8.4, Representing "Entity Subtypes".*

This exercise is based heavily on what I see when I use internet banking with a bank at which I have several accounts. You are to develop a **Tutorial D** database definition, using `VAR` and `CONSTRAINT` statements, to implement the specification given below. Assume that types `DATE` and `TIME` are available for dates and times of day, and that the usual comparison operators are defined for these types. Otherwise, use type `RATIONAL` for currency amounts and `CHAR` for everything else (we are not concerned, here, with constraints defining formats for customer numbers, phone numbers, e-mail addresses, and so on).

Scenario, Requirements, and Business Rules

The bank has **customers**. Each new customer is assigned a unique **customer number**. Each customer's **name** and **address** must be recorded.

BR1 Every customer is uniquely identified by a **customer number** and has exactly one **name** and exactly one **address**.

Optionally, one **e-mail address** for a customer can be recorded, as well as up to three **phone numbers**—but no more than one phone number of each type (home, business, mobile).

BR2 A customer can additionally have at most one **e-mail address**.
BR3 A **phone number** is of exactly one type, home, business, or mobile.
BR4 A customer can additionally have at most one home phone number, at most one business phone number, and at most one mobile phone number.

To become a customer of the bank, one most open at least one **account**. Of course it is possible for the same customer to have several accounts (for example, current, savings, mortgage, and so on). Each account is uniquely identified by an **account number**. For each account, the account holder's **customer number** must be recorded and also the **account type**, and the **date** on which the account was opened. The same customer is permitted to hold several accounts of the same type.

BR5 Every account is uniquely identified by an **account number** and has exactly one **customer number**, exactly one **type**, and exactly one **date** on which it was opened.

BR6 The customer number of an account is that of an existing customer.

The recording of details of payments into and out of an account is complicated by the various different methods of payment. Each transaction against a particular account is automatically assigned a **transaction number** upon reception by the bank's computer. Transaction numbers are unique within an account.

BR7 A transaction is uniquely identified by the combination of its **account number** (identifying an existing account) and **transaction number**.

Every transaction is for an **amount** of money.

BR8 Every transaction has exactly one **amount**.

Every transaction is somehow dated and the relevant date of no transaction can precede the date on which the relevant account was opened.

In addition to the amount, the information associated with each transaction number varies according to the kind of transaction, as follows.

Payments in:

For a payment into an account, the **account number**, **date, time**, and **source**.

BR9 Every payment in has exactly one **account number**, exactly one **date**, exactly one **time** of day, and exactly one **source**.

Payment by cheque:

A cheque on a particular account is uniquely identified within that account by its cheque number. For a payment by cheque, the **account number, cheque number, date written** (as shown on the cheque)**, date processed** (by the bank)**, payee,** and **amount** are recorded.

BR10 Every payment by cheque has exactly one **account number,** exactly one **cheque number,** exactly one **date written** (as shown on the cheque)**,** exactly one **date processed** (by the bank)**,** and exactly one **payee.**

It is possible for more than one payment to be made by cheque to the same payee with the same date written and date processed. It is assumed that cheque books are not issued to a customer until the relevant account has been opened. A cheque cannot be processed before the date written as shown on the cheque.

BR11 The **date processed** of a cheque cannot precede its **date written.**

Payment by direct debit:

For a payment by direct debit, the **account number, date, time, payee,** and **amount** are recorded. Note that is theoretically possible for more than one payment to be made by direct debit to the same payee at exactly the same time on the same day.

BR12 Every payment by direct debit has exactly one **account number,** exactly one **date,** exactly one **time,** and exactly one **payee.**

Payment by debit card:

This includes cash withdrawals from ATMs. A customer can be issued with any number of debit cards. Each debit card is for a particular account and several debit cards can be issued for the same account, perhaps for use by various family members. Each debit card is identified by a **card number**. For every debit card, the relevant **account number**, **cardholder's name,** and **expiry date** are recorded. For a payment by debit card, the **card number, date, time, payee** (which might refer to an ATM)**,** and **amount** are recorded. It is not possible for the same debit card to be used more than once at exactly the same time on the same day. It is not possible to use a debit card for any payment on a date after the expiry date.

BR13 Every debit card is uniquely identified by a **card number** and has exactly one **account number** (identifying an existing account)**,** exactly one **cardholder's name,** exactly one **expiry date,** and exactly one **payee.**

BR14 Every payment by debit card is uniquely identified by the combination of its **transaction number** and **card number** of an existing card, also by the combination of its **card number**, **date**, and **time**, and has exactly one **payee**.

BR15 The **date** of a payment by debit card cannot be later than the expiry date of the relevant card.

BR16 Every transaction is either a payment in, or a payment by cheque, or a payment by direct debit, or a payment by debit card.

BR17 No transaction is of more than one of the types mentioned in **BR16**.

BR18 A transaction other than a payment by cheque cannot be dated earlier than the date on which the relevant account was opened.

8. Based on your experiences with Exercise 7, suggest enhancements to **Tutorial D** to make it easier to express any constraints you declared that struck you as being of a common enough kind to warrant an additional shorthand.

9. (For students familiar with SQL). Consider the following SQL definitions:

```
CREATE TABLE SF ( StudentId CHAR(4),
                  Faculty VARCHAR(50),
       PRIMARY KEY ( StudentId )
       UNIQUE ( StudentId, Faculty ) ;

CREATE TABLE CF ( CourseId CHAR(4),
                  Faculty VARCHAR(50),
       PRIMARY KEY ( CourseId )
       UNIQUE ( CourseId, Faculty );

CREATE TABLE SCF ( StudentId CHAR(4),
                   CourseId CHAR(4),
                   Faculty VARCHAR(50),
        PRIMARY KEY ( StudentId, CourseId ),
        FOREIGN KEY ( StudentId, Faculty )
                   REFERENCES SF ( StudentId, Faculty ),
        FOREIGN KEY ( CourseId, Faculty )
                   REFERENCES CF ( CourseId, Faculty ) ;
```

a. What problem was the designer solving here?
b. What possible problem remains in this solution?
c. Describe and comment on the particular features of SQL that make this solution possible.

1.8 Additional Exercises Using Rel

1. Explore *Rel*'s catalogue. It consists of a relvar named `sys.Catalog`. Use the following trick to see `sys.Catalog`'s heading only:

```
sys.Catalog WHERE FALSE
```

From their names, you might be able to guess which attributes are of most interest (possibly `Name, Owner,` and `isVirtual`?).

Create a virtual relvar named `myvars` giving the `Name, Owner,` and `isVirtual` of every relvar not owned by 'Rel'. Virtual relvars are created like this:

```
VAR name VIRTUAL relation-expression ;
```

Test your virtual relvar definition by entering the queries

```
myvars
myvars WHERE isVirtual
(myvars WHERE NOT isVirtual){ALL BUT isVirtual}
```

2. If you haven't already done so, load the file `OperatorsChar.d`, provided in the Scripts subdirectory of the *Rel* program directory, and execute it. One of the relvars mentioned in `sys.Catalog` is named `sys.Operators`. Display the contents of that relvar. How many attributes does it have? What is the declared type of the attribute named `Implementations`?

3. Evaluate the expression

```
(sys.Operators ungroup (Implementations)
where Language = 'JavaF')
{ ALL BUT Language, CreatedByType, Owner, CreationSequence}
```

What are the "ReturnsTypes" of `LENGTH`, `IS_DIGITS`, and `SUBSTRING`?

4. Note that if *s* is a value of type `CHAR`, then `LENGTH(`*s*`)` gives the number of characters in *s*, `IS_DIGITS(`*s*`)` gives `TRUE` if and only if every character of *s* is a decimal digit. `SUBSTRING(`*s*`,0,1)` gives the string consisting of the first *1* characters of *s* (note that strings are considered to start at position 0, not 1). `SUBSTRING(`*s*`,`*f*`)` gives the string consisting of all the characters of *s* from position *f* to the end.

What is the result of `IS_DIGITS('')`? Is it what you expected? Is it consistent with the definition given above?

5. Using operators defined by `OperatorsChar.d`, define types for supplier numbers and part numbers, following Example 2.4 from Chapter 2.

Define relvars `Srev`, `Prev`, and `SPrev` as replacements for `S`, `P` and `SP`, using the types you have just defined as the declared types of attributes `S#` and `P#`.

Write relvar assignments to copy the contents of `S`, `P` and `SP` to `Srev`, `Prev`, and `SPrev`, respectively. Note that if `SNO` is the type name for supplier numbers in `S` and `Srev`, then `SNO(S#)` "converts" an `S#` value in `S` to one for use in `Srev`.

6. Using the relvars defined in Exercise 5, repeat Exercise 6 from the set headed "Working with A Database in *Rel*" given with the exercises for Chapter 4. Which of your solutions need revisions beyond the obvious changes in relvar names?