CMSC 661 — RDB Tutorial

1 Introduction

This document is intended as a simple introduction to the RDB system, and contains a tutorial as well as some optional exercises. RDB was written by Rishyur Nikhil at the University of Pennsylvania. This document is adapted from one written by Professor Susan Davidson of the University of Pennsylvania.

This note is designed to take you step by step through a session with RDB; all you have to do is read this booklet and follow the directions. Anytime we want you to do something, we will precede the suggestion with the symbol "$\Longrightarrow$". For example:

$\Longrightarrow$ Pay particular attention to the next sentence.

2 Using RDB: The Basics

To start RDB, just type "rdb" at the command line on UMBC8. RDB is interactive, so that once you are in RDB it will ask you to type something by printing "rdb>" on the terminal and will answer you with a command or an expression representing a relation. When you type something, it will respond with either an error message (if it can't figure out what you said), a picture of a relation (if you gave it an expression which describes a relation, i.e., a query), or any one of various other messages (if you typed a command).

Now, let's look a little more closely at the sort of things we can ask RDB to do. RDB expects you to type in either an *expression* or a *command.* A command is a request to do something in the system like load a database (we'll do this in a moment), save some relations in permanent storage, look at a list of the available relations, create a new name for some relation, or log off. For example, $\Longrightarrow$ brace yourself and then try typing "help" followed by a semicolon, like this:

```
rdb> help;
```

This will list all of the commands and expressions available. (Although this listing is difficult for the novice to understand, it's useful as a quick reference once you are familiar with RDB. We will not be using some of the commands listed. Complete details can be found in the manual handed out with this booklet.)

All commands and expressions typed into RDB must be terminated by a semicolon, as in the "**help**;" command above; this is what tells RDB where the end of the command or expression is. If you forget the semicolon, or if you are typing something which takes more than one line, the system will give you a prompt which looks like this:

```
...>
```

This prompt tells you that you haven't finished the command, and that RDB expects you to type the rest.

One of the most important things to learn about any system is how to recover after you've made an error. When RDB can't figure out what you've just said, it will print an error message. $\Longrightarrow$ Type "**let 5**" (with no semicolon) and hit the RETURN to see an example for an error message:

```
rdb> let 5
```

Notice that after the message, you see the "...>" prompt. This is because RDB tries to figure out each line as soon as you hit RETURN, and it couldn't understand the "5", which was illegal (for reasons which aren't important right now), but it didn't see a ";", so it's still trying to read your command. *Hit ";" to get back to "rdb>" anytime you get the prompt "...>" after an error message.* Error messages occur after you've typed in a command or expression incorrectly; try to figure out what went wrong and then retype the command (after getting back to the "rdb>" prompt).

One common command is **use**, which tells RDB that we want to use a particular database (i.e., set of relations). For this database tutorial, we will use the Suppliers and Parts database, which contains information about suppliers, parts, and how much of a part is supplied by a supplier. $\Longrightarrow$ To load this database into RDB, we must type **use** followed by the name of the file which contains the database on this system:

```
rdb> use 'sp.rdb`;
relation S bound
relation P bound
relation SP bound
```

This is the database which we will use for the duration of this note [1]

The other thing which we can type into RDB is an *expression*. An expression is a request to print the value of something, such as a relation. Typically, we will type in a **relation-expression** which represents some temporary relation created by performing various operations on the relations stored in the database. RDB will evaluate the relation-expression and print out the result. (In fact, RDB can evaluate more than just relation-expressions. It can also do arithmetic and many other things. $\Longrightarrow$ Try typing the expression "**23+2*5**;".) A simple example of a relation-expression is just the name of a relation. RDB evaluates the expression "**S**;" by returning the value of **S**, which is just the relation itself. $\Longrightarrow$ For example, look at the relation **S** by typing "**S**;":

```
rdb> S;
-------------------------
|snum|sname|status|city  |
|----|-----|------|------|
|s1  |Smith|    20|London|
|s2  |Jones|    10|Paris |
|s3  |Blake|    30|Paris |
|s4  |Clark|    20|London|
|s5  |Adams|    30|Athens|
--------------------------

rdb>
```

RDB evaluated the expression "**S**;" and printed its value, which is the relation **S** stored in the database. $\Longrightarrow$ In the same way, look at the relations **P** and **SP**.

3 Querying the Database

Now let's look at expressions which manipulate the database. By typing a relation-expression, you will ask RDB to print some result relation, that is, you will *query* the database. Recall the operations of *projection* (which picks out columns of relation) and *selection* (which picks out rows). The RDB syntax for projecting out a list of attributes (i.e., column names) over some relation is:

```
project <relation-expression> over <attribute-list>;}
```

In place of the **<relation-expression>** you can type any other expression which results in a relation, and in place of **<attribute-list>** you can type a list of attributes (i.e., column names) separated by commas. For example, since a name of a relation in the database is an expression, we can project out the **snum** and **sname** columns of the **S** relation by typing:

```
rdb> project S over snum, sname;
```

$\Longrightarrow$ Try this and compare it with the original relation.

The first exercise uses simple projections. You must figure out the proper expressions to type in to construct (i.e., print out) the specified relations from the relations in the database.

$\Longrightarrow$ **Optional Exercise 1.** Write expressions which return these relations:

[1] Note that the system responded with a message which tells us that three relations, S,P, and SP, were initialized as a result of the **use** command. We will print out these relations in a moment, after discussing the concept of relational-expressions.

- The **citys** of **S**.
- The **pnums** and **colors** of **P**.

The RDB syntax for selecting rows of a relation which satisfy some condition is:

```
restrict <relation-expression> with <condition>
```

The "**<condition>**" can be any boolean expression (i.e., one that is either true or false, such as 3<3 or city = 'paris') which involves the attributes of the relation. For example, to select the rows of the relation **P** which have the value '**screw**' in the column **pname** we would type:

```
rdb> restrict P with pname = 'screw';
```

$\Longrightarrow$ Try this and compare it with the original relation. Note carefully that a value which is a word *must* be enclosed in single quotes; this is how RDB distinguishes between column names and actual values. To find all the rows of **SP** where the **qty** attribute is greater than or equal to **300** *or* where the attribute **snum** equals **'s1'**, we would type

```
rdb> restrict SP with (qty > = 300) or (snum = 's1');
```

$\Longrightarrow$ Try this and compare it with the original relation; make sure you understand how the rows of the resulting relation satisfy the condition.

We can combine selections and projections by nesting these expressions, in other words, by replacing the **"<relation-expression>"** by another (possibly very complex) expression. For example, to find all the rows of **S** where the attribute **snum** has the value **'s1'**, and then print the **sname** and **city** attributes, we would type:

```
rdb> project (restrict S with snum = 's1')
...>         over sname, city;
```

$\Longrightarrow$ Try this and compare it with the original relation. Now we are ready for the next exercise.

$\Longrightarrow$ **Optional Exercise 2**. Construct expressions which will print out the following relations:

1. All the rows in the relation S where the city is paris and the status is greater than 20.
2. The pnames of P whose weight exceeds 16.
3. The snames and citys of S with a status of 30.

The cross-product of two relations **A** and **B** is the composite relation consisting of all rows one could create by concatenating a row from **A** with a row from **B**. For example, to see the cross-product of the relations **S** and **SP**, $\Longrightarrow$ type

```
rdb> join S with sp;
```

(In fact this is an unfortunate piece of syntax, since this is *not* the join of the two relations). $\Longrightarrow$ Verify that this result relation has created every possible concatenation of the rows from the two relations. Note also how the column names correspond to the two relations. You can verify if you wish, that the result relation has 60 rows, that is, 5 times 12, the number of rows in **S** and **SP** respectively.

Recall that the *θ-join* operation on, for example, relations **A** and **B** is a selection of rows from the cross-product of **A** and **B** where a column from **A** stands in the relationship θ with a column from **B**, with θ being some relation such as less-than (<), greater-than (>), etc. In particular, the *equi-join* requires that the specified two columns be equal. We can express equi-joins in RDB as

```
join <relation-expression>
     with <relation-expression>
     on <attribute-list>;
```

An example will clarify this difficult concept. ⟹ Type the following expression, which produces the equi-join of **S** and **SP** on the column **snum**.

```
rdb> join S with SP on snum;
```

The result relation is presented with the columns labeled **snum** in the two relations merged into one column, since they would be equal in any case. Try to identify the rows from the original relation (the first three columns will correspond to the rows from **S**, and the first, fourth, and fifth columns will correspond to the rows from **SP**). This relation shows how valuable the join expression is: information from two different tables which is related, such as the supplier names **(sname)** and part numbers **(pnum)** for the various supplier numbers **(snum)**, is brought together into one relation.

Thus, suppose you wanted to perform the query: "Find the part numbers of all parts which supplier **smith** makes." Intuitively, we know by looking at the relation **S** that this supplier has an **snum** of **'s1'**, and by looking at the relation **SP** we see that supplier **'s1'** makes parts **'p1'** through **'p6'**. The information we used to conceptually "join" these two relations is in the column **snum**, so the query which extracts this information from the database is an equi-join on this field. The following expression would perform this query:

```
rdb> project (restrict (join S with SP on sum)
...>                   with snum = 's1')
...>          over pnum;
```

In other words, we joined **S** and **SP** on the attribute **snum** because these two relations contained the information we needed and **snum** contained the data which linked the two, we selected only those rows which contained information about suppliers **'s1'**, and then we projected out the exact data we wanted. ⟹ Perform this query and verify that the result follows this intuition. Make sure you understand the rational behind the equi-join before you go on with the third set of exercises, all of which use the equi-join.

⟹ **Optional Exercise 3.** Here we will pose the problems as questions to be answered. Construct expressions which will return relations which have exactly the information necessary to answer the question.

1. What are the names of the suppliers who supply some part in quantity greater than 200?
2. Which cities have suppliers who supply some part? (Note that some suppliers don't supply any parts.)
3. (*Optional*) What are the names and cities of the suppliers who supply a red part? (Hint: you'll need to do two joins to get this one.)

---

4 Defining Functions and Views

We saw briefly in the last section that RDB is a special kind of database language in that it allows the user to do arithmetic as well as write queries against a database. In fact, RDB has many of the features of functional programming languages such as LISP. In this section we will use this capability (which few database languages possess) to define functions over database values and also special types of functions called *views*.

We can give values to variables, functions, or relation names in RDB by using **let**, which has the following syntax:

```
let <identifier> = <any-expression>;
```

For example, $\Longrightarrow$ verify that the dialog below is correct by typing, as usual, the **boldface** commands. The comments following the "%" are explanations of what's going on; don't type these.

```
rdb> 34;
34
rdb>3 + 5;                % Any arithmetic expression
8                         % can be evaluated
rdb>let a = 54;           % Give a the value 54
integer a bound
rdb>a;
54
rdb>let b = 3 + a;
integer b bound
rdb>b;
57
rdb>let d = 'hi there';   % Any type of value can be
hi there                  % given to any variable
rdb>let f(x) = x+1;       % This is the way to
function f bound          % define functions
rdb>f(3);                 % calling the function f
4
rdb>f(f(10));
12
```

The last three examples might need a bit of explanation if you are not familiar with the concept of functions. When we define the function **f** by typing

```
rdb> let f(x) = x+1;
```

we are telling RDB that when we type **f** with some integer in place of the **x** in **f(x)**

```
rdb> f(3);
```

we want RDB to substitute **3** for **x** in the expression **x+1** and evaluate that instead. Thus a function is a way of defining how values are to be substituted for variables in some expression [2] $\Longrightarrow$ See if you can explain how the expression f(f(10)) is evaluated.

We can also define functions on relations. $\Longrightarrow$ Define the function **xprod** which finds the cross product of two relations:

```
rdb > let xprod (a,b) = join a with b;
function xprod bound
rdb> xprod (S,P);
```

Here when we call **xprod** on the arguments **S** and **P** we are asking RDB to substitute these relation names for **a** and **b** in the join expression and evaluate it. We could also use the same method to substitute column names or actual values. We will use the **let** command in the next set of exercises.

$\Longrightarrow$ **Optional Exercise 4.** Define the following functions.

1. Construct a function **whereis(sn)** which returns the (single column) relation of **citys** of **S** whose **sname** is **sn**. (Remember that **sn** is like a placeholder for the value we will give the function when it is called.) Find the city where **smith** is located (i.e., by evaluating **whereis('smith');).**

2. Construct a function **overWeight(w)** which returns a relation consisting of the **pnames of P** whose weight exceeds w. Compare this with Exercise 2.2; perform the same query found there by typing "**overWeight(16)**".

[2] For those with some programming experience, we should mention that we can define recursive functions in RDB as well. Try typing in the definition of the factorial function: let fact(x) = if x = 0 then 1 else (x * fact(x-1); now evaluate fact(10).

A *view* of a database is a conceptual relation, that is, a relation which is not actually stored in the database in the form in which you see it, but is constructed by the system every time you access it. We can define views in RDB by simply defining a function on relations with no arguments. (It will look just like a relation name with an empty set of parentheses at the end). The important difference is that, unlike a relation explicitly constructed and stored in the database as a set of rows, a view will be constructed anew every time we access it, so that it will always give us the present state of the component relations; in other words, updates to the component relations will be reflected in the view.

For example, define a relation **sc** which gives the names of all cities where a supplier is located:

```
rdb> let sc = project S over city;
relation sc bound
```

⟹ Look at sc. Now ⟹ define the *view* **scview ()** which gives the names of all cities where a supplier is located:

```
rdb > let scview () = project S over city;
function scview bound
```

Note carefully that the presence of the parenthesis "()" indicates to RDB that the second definition is a function. ⟹ Look at **scview ().** It is obviously the same as the relation **sc**. But now we will update the relation **S** by adding a new row [3].

⟹ Add a row to the relation **S** by typing:

```
rdb > let S = S + { snum sname status city
...>                [ s6 Lewis 25 Phila ] };
relation S bound
```

⟹ Now look at the relations **S, sc,** and the view **scview** (). Can you explain what happened?

⟹ **Optional Exercise 5.** Define the view **spq ()** of **sname, pnum,** and **qty** from **S** and **SP.** (Hint: you'll need a join.) Update the relations **S** and **SP** as follows and then examine the view.

```
rdb> let S = S - {snum sname status city
...>              [s5 adams 30 athens] };
```

In order to log off the system, type

```
rdb> exit;
bye...
```

[3] We have not discussed how to update relations. Briefly, a row is represented explicitly by enclosing the appropriate values, in the correct order and without commas, between square brackets. Words are represented without quotes. We can simply add or subtract rows from the relation by reassigning the name of the relation, as in the example. The RDB manual contains more information about creating and modifying relations.