 TM254 Managing IT: the why, the what and the how

SQL Implementation  
Block 2 Part 6 Activity 6.1

This activity should take you no more than 2 hours. It will explore the process of translating the Feeding Time scenario’s posted-key relational representation (a logical schema) into PostgreSQL (a physical schema representation).

# Introduction

In this activity, we will walk through the translation of some of the Feeding Time relations (produced in Part 5, Activity 5.2) into PostgreSQL; then you will complete the implementation of the Feeding Time scenario. Prior to this we will cover aspects of using the PostgreSQL DBMS to create a database and add users.

# 1 Preparation for implementing the SQL code

The final task before translation to implementable code is to review the model design. The logical schema can now be checked against the initial conceptual data model to ensure that there has been no drift from the original requirements, or that any drift is adequately documented. In addition, you should check that the relations have been normalised to third normal form, so that the database implemented has limited redundant duplication.

There is often an initial activity to choose appropriate SQL data types and value constraints that will define appropriate domains for the attribute values.

The translation process is the same whether the relational model used the relation-for-relationship or posted-key representation. Choose a relation and convert it to SQL code. Remember that tables referred to by foreign keys must be defined before the foreign key that refers to them (or use the ALTER TABLE statement to declare all foreign keys after the main table definitions).

Of course, there is the problem of ensuring that your chosen SQL dialect contains the features you require, or having a strategy to handle those parts that cannot be directly realised. These limitations need resolution, or a workaround, and this will need documenting. It is usually the case, however, that moving from a relational logical schema to a relational SQL implementation is relatively straightforward – often with a line-for-line translation from relations to SQL definitions.

In a company setting, you will find that there are style guides and standard practices (hopefully best-practice guidance) that you are expected to follow, along with advice on documentation and any supporting files. The decision to use graphical interface tools rather than SQL language statements is another option available. If the development has been taking place within a supporting CASE tool, then the tool may be capable of generating the SQL statements and documenting the database. All of these aim to turn code generated by a ‘quick’ translation into a software engineering artefact that can be safely integrated into a corporate information system.

Before we can start creating SQL tables, we need to gain access to the PostgreSQL environment and create a user who will create and own the tables. At this point in the module, we’ll create a very simple set-up: a user who is able to access a schema in which all the SQL statements can be evaluated.

## 1.1 The PostgreSQL database and schema

We’ll be covering the following in more detail in later parts of the module, but a quick overview of the PostgreSQL database structure is needed here.

You’ll know from the *Databases Software Guide* that a default database postgres is created when PostgreSQL is installed, along with a superuser known as postgres. In the *Databases Software Guide*, we then went on to run a script to create a new user, student, and a new database, study; within that database, we created the economics schema. You saw how to log in to those and check that the installation was successful. The following explains how to create and populate a database with tables in the public schema within that database.

A PostgreSQL schema is a way of allowing multiple users to share a database while keeping their database objects, tables, etc. separate to which users may or may not have access.

Creating a database in PostgreSQL always creates a default schema called public. The public schema is the one where, by default, all new tables are created. It’s also a schema that, as its name suggests, is usually available to all users. However, it is also possible to have more than one schema within a database, and we can control who has access to which schema. The superuser and database owners can create different schemas and grant access to those schemas to different users, as shown in Figure 1.

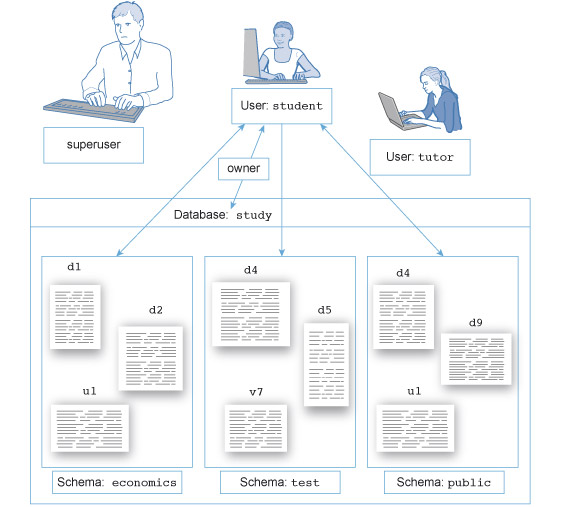


Figure 1  A study database with three schemas and two users

So if we create a new user, we know that all objects created by that user will be, by default, in the public schema. Later in the block we’ll see how to use multiple schemas, but for now we’ll use the public schema as our workspace.

## 1.2 Creating a new user in psql

Initially we’re going to create our tables in the default database postgres, in the public schema. We’ll create a new user with the name r2d2 who will be given access rights to the postgres database, and by default will be given access to the public schema. Any tables, domains, etc. created by r2d2 will appear in the public schema, and any SQL queries will look to the public schema for the tables they reference.

If you have followed the *Databases Software Guide* instructions for installing PostgreSQL, you should be able to start the psql tool and connect to the default postgres database. In psql, accept the default values by hitting Enter, then supply the password you used when installing the system. This logs you in to the default postgres database as the superuser known as postgres. Notice that psql shows a prompt that ends in a # as a reminder that you have SuperUser privileges; if you did not have SuperUser privileges, the prompt would end with >.

You could choose your own username – if you do so, you’ll need to remember it – but for this activity I’ll create a user called r2d2 and declare that the user is permitted to log in.

At the command prompt in psql, type:

CREATE USER r2d2 WITH LOGIN;

With my security hat on, I also need to set a password for r2d2, which I can trigger using a command-line directive in psql. So, at the psql prompt, after evaluating the above SQL statement, type:

\password r2d2

This will prompt for a new password for the r2d2 user. Note that the characters you enter will not be displayed, so be careful as you type. You will be prompted to enter your password again as a check that it is correct. (Do not forget the password you choose! Although if you do forget a user password, then a user with superuser privileges can reset it.)

Also, with my security hat on, we need to give the new user we have created the necessary permission access the public schema. (Originally Postgresql the default setting that all new users could access the postgres database public schema, but to make it more secure, the superuser now needs to grant that access explicitly).

So, at the sql prompt, type:

GRANT ALL ON SCHEMA public TO r2d2;

(We cover more on the GRANT command in Part 9: Managing data access)

If you ever need to remove a user, they can be DROPped by the superuser or any other user who has been given appropriate privileges (DROP user r2d2;). (If you forget a username, you can produce a list of current usernames defined on a server by typing \dg at the psql prompt.)

Warning

SQL is case insensitive; however, passwords are case sensitive.

SQL converts usernames to lower case – so if we had entered CREATE USER R2D2 WITH LOGIN; this would have created the username r2d2, and at the psql prompt you would have had to enter r2d2 as the username (psql does not convert to lower case). However, the connection password remains case sensitive – so if you create the password BEEP, then psql requires the password BEEP; beep would not be accepted.

If you are having problems connecting to a database, try going lower case for the username and think carefully about the case of the original password characters.

Now that you’ve created the new user and given that user an initial password, you can stop working with superuser privileges, disconnect from the database and connect again as r2d2. So, use \q to leave psql.

From this point onwards in this activity, you can connect to the default database using the username r2d2 (or the username you chose) and the password you set. Any domains and tables you create will be created in the public schema of the default postgres database.

## 1.3 Working with SQL statements

When developing a database, it’s usually not a good idea to work with an interactive interface such as psql. It’s hard to correct missteps and mistakes, and after a long interactive debugging session it can be hard to recreate a sequence of working statements accurately.

It is preferable to work with a text editor to create the script file. You can then either copy and paste lines from the script file, or have psql read the content of the entire script file (as described in the *Databases Software Guide*). I tend to work using the Windows operating system, so I use Notepad to create my text files (but any text editor will do). Getting into the habit of copying and pasting into psql has the advantage that you can copy and paste many of the example SQL statements in the module material to see how they behave.

Ideally, the text file(s) will eventually be edited to contain a script or scripts that can be executed, without error, to create the required table and supporting objects for the database.

If the script starts to get complex, splitting it into different files can help to manage the complexity. So, I often have a script for any user, database and schema creation (often one that I run with superuser status). I have a separate script of domain declarations, and a script of table and view declarations (which can be run without superuser status). I also tend to keep data insertion scripts (to initialise table content) separate from database implementation scripts. However, you may have other preferences, or you may work in a setting where there is a standard expectation of how to partition and name script files.

For this activity I’d suggest keeping it simple, with all implementation statements in one script.

## 1.4 Comments in SQL scripts

It’s always good to be able to comment code files. SQL has two standard notations for comments, one using double dashes and the other using slashes and stars:

-- Lines beginning with the double dash are SQL comment lines.

SELECT \* FROM table; –- Comments can appear at the end of lines

-- and run to the end of the line.

/\* A multi-line comment can appear

between these begin (slash star)

and end (star slash) markers. \*/

## 1.5 DROP scripts

While developing a new database implementation, it’s very easy to construct a complex collection of interdependent objects in your SQL implementation – domains, tables, foreign keys, views, etc. SQL will not allow you to remove an object if some other object depends on it. (This is a consequence of declaration before use – if you are using it, you can’t delete it!) So, if you make a mistake in a domain declaration, say, and have already used this in a table declaration, then you cannot simply delete the domain and recreate it. SQL will tell you that the object you are trying to delete is being used by another object, and refuse to remove it.

To correct this problematic domain, you will need to delete and recreate any tables that depend on it; yet to delete those tables, you need to delete any tables that depend on those tables, and so on. Often it is easier to edit the implementation scripts (say, to correct the domain mistake), delete the entire contents of the databases and then rerun the installation scripts. (**Warning**: never do this with a database containing ‘live’ data unless you have a strategy for saving and re-importing the data!) To simplify this process, developers often create DROP scripts. The purpose of these is to DROP (delete) the SQL objects they have been working on, but to do it in an order that ensures any dependencies are respected.

So, if you make a mistake running a CREATE script (and you will), you need to know how to undo the things you have created. It should become a habit to write the DROP script at the same time as you develop the CREATE scripts – the idea is that running the DROP script will undo anything in the related CREATE scripts and allow you to rerun the CREATE scripts. When writing the DROP script, the dependencies mean that you usually end up dropping things in the reverse order to the scripts that create the objects.

In the sample scripts at the end of this activity (you can see this in the file Feeding\_Time\_DROP\_script.txt) I’ve coded the DROP statements using the IF EXISTS version of DROP:

DROP TABLE IF EXISTS feeds;

This ensures that the script doesn’t stop when it tries to drop an object that has not been created. This provides an easy way to entirely replace the fragments of the databases created if an implementation script breaks partway through, or at any time after we’ve created and populated the tables with data.

Warning

While this is good practice in a development or teaching environment, it is a **very bad thing** when the organisation depends on the data in the tables that have been dropped. So **be very careful** with DROP scripts after your database goes live!

If you are working interactively, one statement at a time in psql, you may make a mistake when creating a table – getting its name wrong, missing an attribute or foreign key, etc. When you are more experienced with SQL you may find ways to fix the error without needing to recreate the table – but the easiest way to fix an immediate error is to execute DROP TABLE <name of table just created>; then re-type the CREATE statement.

# 2 Translating the relational representation to SQL

This activity has a similar format to the previous activities that produced the logical schema from the conceptual data model. I’ll present the translation of one of the relationships (OccursAt) from its relational representation and produce an SQL code fragment that implements the relevant tables. You will then be able to see how the Feeds, Supervises and Classifies relationships are implemented in SQL tables, before trying to write your own SQL code for the remaining relationships.

We’ll use the posted-key representation of the relational model, as this contains examples of both posted-key and relation-for-relationship representations. As a reminder of the relational model developed at the end of the Part 5 Activities (Logical Schema 1, and Logical Schema 2), you can look at the document Solution for Logical Schema 2 on the module website.

The output of this process will be a text file containing executable SQL code that can be imported and run in the PostgreSQL DBMS through the psql interface tools. I’ll include comments in the SQL code text to explain what the code represents – but Part 6 covers the CREATE TABLE statement and the structural elements needed to represent relationships.

There are two possible approaches to preparing the SQL code that will produce the implemented tables:

1 For each table, define its attributes and primary keys, then go through and add into that definition the foreign keys and constraints to the CREATE TABLE statements. Then check for the ordering dependency between two tables in case of circular definitions, and convert to the ‘CREATE, then ALTER’ order described in the text.

Or

2 For each table, define the attributes and primary keys needed for that table. Then write an ALTER TABLE statement for each foreign key and each constraint required. This should avoid any circular definition issues; also, for some people, the ability to keep together the ALTER TABLE statements that represent a relationship or define a constraint makes it easier to read the SQL code – especially if the script file has useful comments to describe the content.

The individual CREATE statements are very similar, so I’ll show the two versions in the worked example for OccursAt, but not for the Feeds, Supervises and Classifies relationships. You can choose which you prefer for your SQL generation. When you have finished, you should be able to execute and debug your code to develop a working database for Feeding Time.

The starting relational model, representing the logical schema, is shown in Figure 2. This was developed during the activities for Part 5 resulting in the figure in the Solution for Logical Schema 2 document (Activity 5.2), and should be accompanied by the data dictionary describing the data types and constraints reported in the activities in Part 4 (see Solution for E-R Model 2, Activity 4.4).

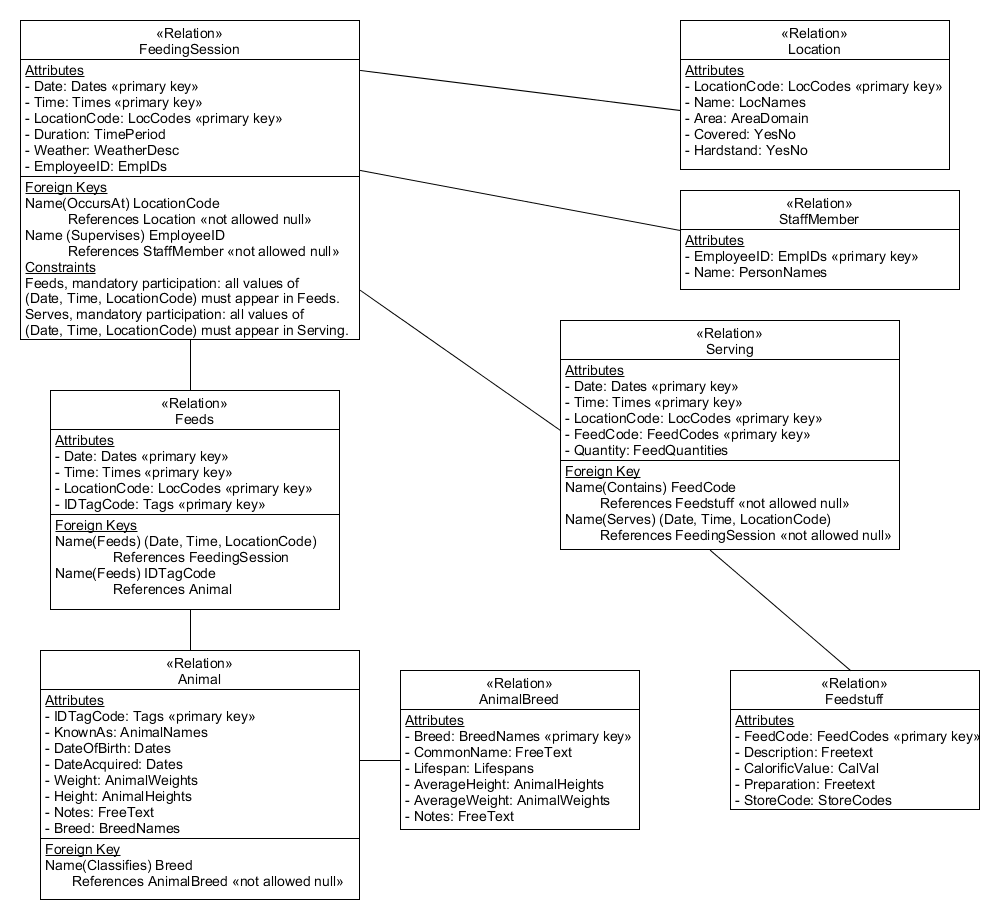


Figure 2  Relational model, using posted keys, for the Feeding Time scenario

## 2.1 Checking the relational model is in third normal form

We start by checking that the relations have been normalised to each of first, second and third normal form, as follows:

* All the attributes are single valued, and all the relations have primary keys declared (forcing unique rows), so all the relations are in first normal form.
* None of the attributes are uniquely determined by part of the primary key in any relation; they all depend on the full primary key. So all the relations are in second normal form.
* There are no transitive dependencies, so all the relations are in third normal form.

## 2.2 Implementing the required domains

The first stage to produce the SQL, assuming you already have a username and access to a database in which you can create new objects, is to write the CREATE DOMAIN statements for the required domains. I’ve put the SQL domain descriptions into a fragment of the data dictionary – see the document Feeding Time Domain Declarations, which you can read in Word, on the module website. They also appear in the Feeding\_Time\_DOMAINS\_script.txt file. You can open and read the content of this file in any text editor.

The *Databases Software Guide* explains how to take input from a script file using the \i command, which we can use to read the domain definitions into the public schema. I suggest creating a folder for this activity to keep your working files and scripts together, then copying the Feeding\_Time\_DOMAINS\_script.txt file to that folder. Finally, log in to PostgreSQL as user r2d2 (if you recall, this user will by default use the postgres database and the public schema), then use the \i <pathname>/Feeding\_Time\_DOMAINS\_script.txt command to add the domains to the public schema. (If you want to evaluate the example CREATE and ALTER TABLE statements that follow, you will need to run the DOMAINS script before you start.)

## 2.3 Working through the individual relations

Working from the logical schema diagram, you would normally work systematically through each relation, writing the CREATE TABLE statements as required. For the OccursAt example, I’m going to show this with just the relevant bits from the FeedingSession and Location relations that relate to the OccursAt relationship. Note that not all attributes in the FeedingSession relation are relevant to this relationship; those that are appear in Figure 3.

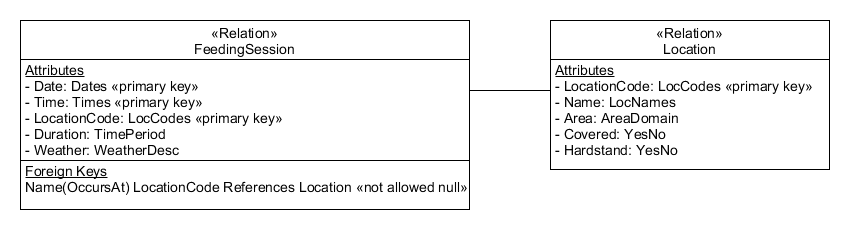


Figure 3  FeedingSession and Location relations, with only the attributes relevant to the OccursAt relationship

### 2.3.1 Using the ‘do-it-all-in-one-statement’ approach

Location needs to be declared first, so that the foreign key reference is valid when FeedingSession is created (recall, declaration before use).

My preference is to work through the relational representation as shown in Figure 3, then go through the general constraints described in the data dictionary (which could have been included in the relational diagram notation). In this case we have general constraints for the uniqueness of the location name, and the feeding sessions recording historic sessions on or before today’s date, and later than the first records were kept (note that this constraint helps to avoid accidentally entering meaningless dates for this context, but cannot prevent someone entering a meaningful but wrong date!).

The end result is the same, as all the constraints appear in one CREATE TABLE statement for the relevant table. My examples assume that the domains have been declared by evaluating the Feeding\_Time\_DOMAINS\_script.txt file.

Note: Layout of SQL is largely a personal preference – I prefer to indent column declarations but not constraints. I also keep the final closing bracket and semicolon on their own line so it is clear where each multi-line statement ends.

CREATE TABLE location(

locationcode LocCodes,

PRIMARY KEY (locationcode),

name LocNames,

area AreaDomain,

covered YesNo,

hardstand YesNo,

UNIQUE(name) -- name has a general constraint

);

CREATE TABLE feedingsession(

date DATE,

time TIME,

locationcode LocCodes,

PRIMARY KEY (date, time, locationcode),

duration TimePeriod,

weather WeatherDesc,

-- relationship OccursAt, mandatory participation  
CONSTRAINT occurs\_at FOREIGN KEY (locationcode) REFERENCES location,

-- FeedingSession is a historic record, not referencing a date in

-- the future, or before records began on 31 May 2002.  
CONSTRAINT session\_date\_is\_historic CHECK ((date <= CURRENT\_DATE)

AND (date > '2002-05-31'))

);

Notice that this is a line-for-line conversion from the relational model to SQL. In this example, because I’m only working with a fragment of the full model, I haven’t implemented the second foreign key in the full FeedingSession (representing Supervises). This was simply to allow me to focus on the OccursAt relationship. I could, of course, add the additional foreign key using ALTER TABLE later, but if I was following the ‘do-it-all-in-one-statement’ approach, I’d work out which order to declare the tables so that I used a minimum number of ALTER TABLE statements.

### 2.3.2 Using the ALTER TABLE approach

As this model fragment doesn’t have any circular constraints and references, there is no real need to use the ALTER TABLE approach – but just for completeness, I’ll show how I would structure the SQL to use this approach.

-- create the basic location table, columns and primary key

CREATE TABLE location(

locationcode LocCodes,

PRIMARY KEY (locationcode),

name LocNames,

area AreaDomain,

covered YesNo,

hardstand YesNo

);

-- create the basic feedingsession table, columns and primary key

CREATE TABLE feedingsession(

date DATE,

time TIME,

locationcode LocCodes,

PRIMARY KEY (date, time, locationcode),

duration TimePeriod,

weather WeatherDesc

);

-- using ALTER TABLE, add constraints representing relationships

-- The relationship OccursAt, mandatory participation

ALTER TABLE feedingsession

ADD CONSTRAINT occurs\_at

FOREIGN KEY (locationcode) REFERENCES location

;

-- using ALTER TABLE, add any general constraints

-- feeding sessions are historic and after 31 May 2002

ALTER TABLE feedingsession

ADD CONSTRAINT session\_date\_is\_historic

CHECK ((date <= CURRENT\_DATE) AND (date > '2002-05-31'))

;

-- location names are unique

ALTER TABLE location ADD UNIQUE(name)

;

Remember that once the SQL has evaluated all these, the table definitions will be the same; these alternative constructs are entirely for the benefit of the code developer and future code readers.

### 2.3.3 Feeds and Classifies implemented in SQL

Figure 4 shows a larger fragment of the relational diagram that includes the Feeds, Supervises and Classifies relationships. Classifies and Supervises are straightforward SQL translations, while Feeds is a more interesting relationship to realise in SQL.

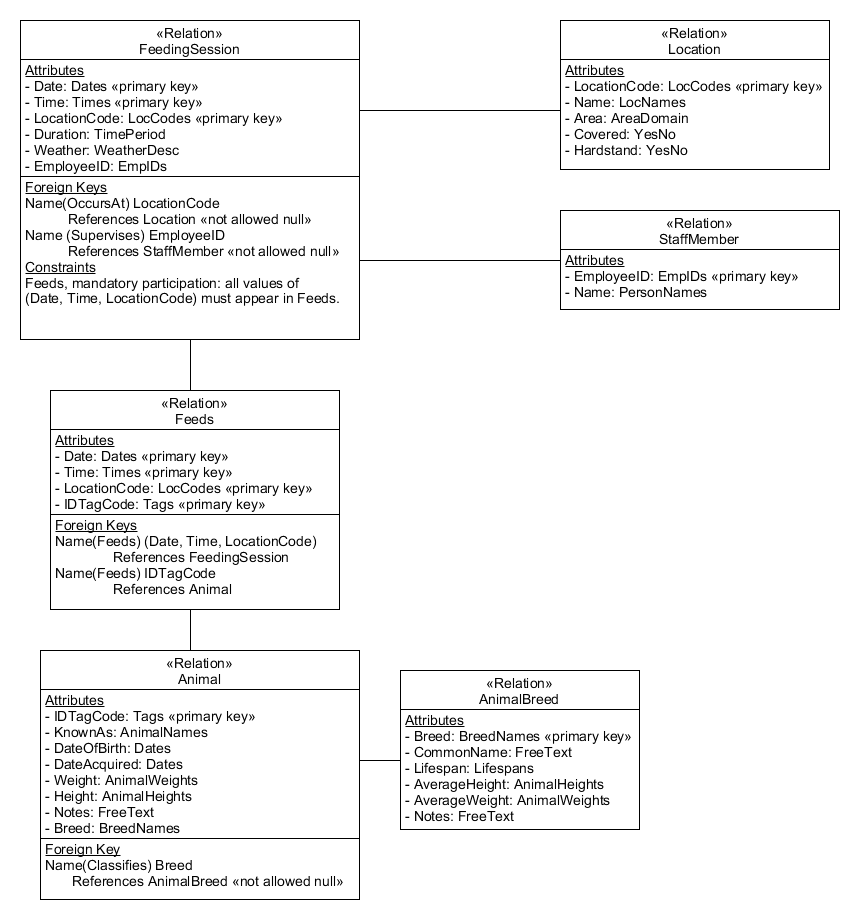


Figure 4  Feeding Time relational model fragment

The Feeds and FeedingSession relations have circular constraints between them. FeedingSession contains a constraint to enforce the mandatory participation and refers to Feeds, while Feeds contains a foreign key that references FeedingSession. As we saw in Part 6, this requires the partial definition of one of the tables without one of the problematic constraints to break the circle of constraints, then the use of ALTER TABLE to add that constraint in order to fully define both tables.

So, to realise the fragment of the full relational model shown in Figure 4 (assuming that domains have been defined as before), we need first to consider the order in which the tables can be created (or the constraints added to the basic tables if using the second approach to developing the SQL).

In this fragment, FeedingSession refers to Location, so Location must be defined before FeedingSession. Animal refers to AnimalBreed, so AnimalBreed must be defined before Animal. Animal must be defined before Feeds, as Feeds refers to Animal. Finally, we have the circular constraints between Feeds and FeedingSession – so we can choose one to implement first (Feeds) without the problem constraint, define the other table (FeedingSession) and then add the missing constraint.

From the above dependencies, I chose the implementation order as Location, AnimalBreed, Animal, Feeds, FeedingSession, and the code looks like:

CREATE TABLE location(

locationcode LocCodes,

PRIMARY KEY (locationcode),

name LocNames,

area AreaDomain,

covered YesNo,

hardstand YesNo,

UNIQUE(name) -- location.name has a general constraint

);

CREATE TABLE staffmember (

employeeID EmpIDs,

PRIMARY KEY (employeeID),

name PersonNames

);

CREATE TABLE animalbreed(

breed BreedNames,

PRIMARY KEY (breed),

commonname FreeText,

lifespan Lifespans,

averageheight AnimalHeights,

averageweight AnimalWeights,

notes FreeText

);

CREATE TABLE animal(

IDtagcode Tags,

PRIMARY KEY (IDtagcode),

knownas AnimalNames,

dateofbirth DATE,

dateacquired DATE,

weight AnimalWeights,

height AnimalHeights,

notes FreeText,

breed BreedNames NOT NULL,

-- relationship Classifies, mandatory participation

CONSTRAINT classifies FOREIGN KEY (breed) REFERENCES animalbreed

);

CREATE TABLE feeds(

date DATE,

time TIME,

locationcode LocCodes,

IDtagcode Tags,

PRIMARY KEY (date, time, locationcode, IDtagcode),

-- relationship Feeds, Animal

CONSTRAINT feeds1 FOREIGN KEY (IDtagcode) REFERENCES animal

);

-- note foreign key to FeedingSession not declared (yet)

CREATE TABLE feedingsession(

date DATE,

time TIME,

locationcode LocCodes,

PRIMARY KEY (date, time, locationcode),

duration TimePeriod,

weather WeatherDesc,  
 employeeID EmpIDs NOT NULL,

-- relationship OccursAt, mandatory participation

CONSTRAINT occurs\_at FOREIGN KEY (locationcode) REFERENCES location,

-- relationship supervises, mandatory participation

-- employeeID is not null in feedingsession

CONSTRAINT supervises FOREIGN KEY (employeeID) REFERENCES

staffmember(employeeID),

-- relationship Feeds, mandatory participation

-- WARNING - the Feeds mandatory participation

-- constraint has not been implemented.

-- This omission permits the entry of a row that

-- does not match a related Feeds record.

-- CONSTRAINT feeds\_mand CHECK ((date, time, locationcode)

-- IN (SELECT date, time, locationcode FROM feeds)),

-- feeding session is a historic record and after 31 May 2002

CONSTRAINT feeding\_session\_date\_is\_historic

CHECK ((date <= CURRENT\_DATE) AND (date > '2002-05-31'))

);

-- relationship Feeds, foreign key to FeedingSession

-- note that using ALTER TABLE like this has separated the

-- full relation-for-relationship pair of foreign keys,

-- so makes interpretation harder

ALTER TABLE feeds

ADD CONSTRAINT feeds2 FOREIGN KEY (date, time, locationcode)

REFERENCES feedingsession

;

This is a case where separating out all the relationship constraints by using the basic table definition followed by a series of ALTER TABLE statements does help readability, as all the relationship-related code can be kept together rather than distributed over the two CREATE TABLE statements for the two tables involved in the relationship.

So, we could have declared the feeds and feedingsession tables as follows:

CREATE TABLE feeds(

date DATE,

time TIME,

locationcode LocCodes,

IDtagcode Tags,

PRIMARY KEY (date, time, locationcode, IDtagcode)

);

CREATE TABLE feedingsession(

date DATE,

time TIME,

locationcode LocCodes,

PRIMARY KEY (date, time, locationcode),

employeeID EmpIDs NOT NULL,

duration TimePeriod,

weather WeatherDesc,

-- feeding session is a historic record and after 31 May 2002

CONSTRAINT session\_date\_is\_historic CHECK ((date <= CURRENT\_DATE)

AND (date > '2002-05-31'))

);

-- relationship Feeds, foreign keys to FeedingSession and Animal,

-- mandatory participation of FeedingSession

-- Note that here, all the constraints related to the Feeds

-- relationship can be declared together, rather than in

-- different CREATE TABLE statements, making the Feeds relationship

-- easier to identify.

ALTER TABLE feeds

ADD CONSTRAINT feeds1 FOREIGN KEY (IDtagcode) REFERENCES animal,

ADD CONSTRAINT feeds2 FOREIGN KEY (date, time, locationcode)

REFERENCES feedingsession

;

-- relationship Feeds, mandatory participation of FeedingSession

-- WARNING - the Feeds mandatory participation

-- constraint has not been implemented.

-- This omission permits the entry of a row that does not match a

-- related Feeds record.

-- ADD CONSTRAINT feeds\_mand CHECK ((date, time, locationcode)

-- IN (SELECT date, time, locationcode FROM feeds))

-- ;

-- relationship OccursAt, mandatory participation

-- locationcode is not null in feedingsession

ALTER TABLE feedingsession

ADD CONSTRAINT occurs\_at FOREIGN KEY (locationcode)

REFERENCES location;

-- relationship supervises, mandatory participation

-- employeeID is not null in feedingsession

ALTER TABLE feedingsession  
ADD CONSTRAINT supervises FOREIGN KEY (employeeID)

REFERENCES staffmember(employeeID)

;

A note about developing SQL code

When I worked through the Feeding Time scenario, it took me several attempts before I produced the above code accurately. I often get SQL syntax incorrect, and have to check carefully the placing of bracketing, keywords, commas, semicolons and the naming of tables and columns. While this is very frustrating, it is also fairly typical of the nature of coding activity – the end product often hides a lot of bitty development. The key thing about the translation to SQL is that you can quickly see what statements are required and sketch the outline – but then the detail to realise that as running code can take a lot longer. It does get better with practice.

# 3 Now, complete the SQL code for the logical schema

Now complete the translation to SQL code of the full logical schema shown in Figure 2 (near the start of Section 2 of this activity). You can use the above table definitions and the file of domain declarations (Feeding\_Time\_DOMAINS\_script.txt) as a starting point for your work.

I suggest working in a text file to develop your script first; then, as you complete table definitions, you can copy and paste them into psql to try them out. You will probably go through several ‘edit, then try, then understand error message’ loops before you get a definition that will work, but you will eventually get there.

You should connect to the default database as user r2d2 to develop your database. The aim is to have an error-free script file (or files) that you can run using the \i command in psql to read and evaluate, which creates the appropriate tables.

Choosing one of the implementation approaches (everything in the table declaration, or separate constraints added using ALTER TABLE) and sticking with it will help you to avoid confusing yourself about syntax. Also, use comments in the script file to remind yourself what the table definition needs to contain – this avoids you getting so focused on a specific bug that you forget the overall shape of the intended table.

You may need to add additional constraints and attributes to the table declarations that appear above. When complete, you will have something to represent each line of text in each relation – either a column declaration, a foreign key or a constraint (or a commented-out CHECK constraint, if PostgreSQL cannot implement the constraint).

If you struggle with any of the translation then use the module forums to discuss and ask for help, but do try to use the PostgreSQL online documentation or a source such as the W3Schools tutorial pages (<https://www.w3schools.com/sql/>) before raising a problem.

This activity isn’t assessed, so there are no problems with using the forums for advice and support. It would also be good, if you’re not having problems with the SQL, to drop into the forums from time to time to see if you can help others. Good coding skills come from sharing practice and knowledge; it can’t all be developed from manuals and tutorials.

# 4 How did you do?

Remember that there are several ways in which the same outcome can be produced in SQL, so your code may not be identical to that produced by another coder. However, when working from a complete relational model it should be possible to get the same overall structure as others who use the same model – that is, the same tables, foreign keys, structural and other representations. So, two different sets of scripts should be comparable.

In the ‘Feeding Time preparation’ area of the ‘Database resources’ section of the module website, you will find a series of script files I produced that contain the SQL code I implemented from the relational model. The Feeding\_Time\_DOMAINS\_script.txt and Feeding\_Time\_CREATE\_TABLES\_script.txt files should be able to be read in the PostgreSQL DBMS, and will create the appropriate domains and tables if you haven’t already created tables with the same names. If running them in the DBMS, you need to use the order Feeding\_Time\_DOMAINS\_script.txt and then Feeding\_Time\_CREATE\_TABLES\_script.txt.

You should be able to read these files in a text editor. The content should be similar to the scripts you generated when you completed the above activity. If there are variations then explore them in the forums.

# 5 Preparing for Parts 7 and 8

In Part 7 we’re going to start looking at the SQL Data Manipulation Language (DML), which covers the INSERT, DELETE, UPDATE and SELECT statements.

To do this, I have developed scripts to implement two databases: study, which you saw when you worked through the *Databases Software Guide*, and bensfarm, which contains the tables developed for the above activity.

If you haven’t completed the *Databases Software Guide* installation of the economics schema then you will need to do this, as the content of Part 7 uses the economics schema for the worked examples. You can check that you have done this by starting psql and connecting to the study database as the user student with password TM254.

During Parts 7 and 8 there will be ongoing activities that allow you to produce SQL statements for queries over the Feeding Time scenario data. To do these, you will need to have installed the bensfarm database according to the following instructions.

In the section of the module website called ‘Feeding Time preparation’, locate the following files:

Feeding\_Time\_DROP\_script.txt

Feeding\_Time\_CREATE\_Users\_Database\_Schema\_script.txt

Feeding\_Time\_DOMAINS\_script.txt

Feeding\_Time\_CREATE\_TABLES\_script.txt

Feeding\_Time\_POPULATE\_small\_tables\_script.txt

Feeding\_Time\_POPULATE\_feeding\_sessions\_script.txt

Feeding\_Time\_POPULATE\_feeds\_script.txt

Feeding\_Time\_POPULATE\_serving\_script.txt

You can open and read these to see what they contain – they make most sense if you read them in the above order.

Now:

1 Connect to the default (postgres) database as the superuser (postgres).

2 Use the \i command to run the Feeding\_Time\_CREATE\_Users\_Database\_Schema\_script.txt file.

• This will create two users (farmadmin and reception, both with the password TM254), the bensfarm database and the feedingtime schema. It will also give access rights to the farmadmin user, including changing that user’s schema search\_path to put the feedingtime schema first as the default schema in the bensfarm database.

3 When the above file read completes, you must log out of the default database (\q).

4 Connect to the bensfarm database as farmadmin using the password TM254.

5 Use the \i command to run the following scripts, one a time in the order shown:

Feeding\_Time\_DOMAINS\_script.txt

Feeding\_Time\_CREATE\_TABLES\_script.txt

Feeding\_Time\_POPULATE\_small\_tables\_script.txt

Feeding\_Time\_POPULATE\_feeding\_sessions\_script.txt

Feeding\_Time\_POPULATE\_feeds\_script.txt

Feeding\_Time\_POPULATE\_serving\_script.txt

• These scripts build the required domains and tables, and then populate them with data.

6 To test that you have installed the scripts as intended, you can log out and connect to the bensfarm database as user farmadmin using the password TM254, then evaluate SELECT \* FROM location; to check that it all has worked.

• This should display a table that has 12 rows present.

We will be using this database (bensfarm), schema (feedingtime) and user (farmadmin) in the activities in Parts 7 and 8.