Learning The World

Pete McBreen

02-Sep-2019

Contents

Chapter 1

Database Archaeology

Of all the tasks facing a business analyst in understanding what a system does, diving into the various legacy application databases to understand the information that is hidden in the database is probably the most daunting. In part this is because many Business Analysts do not have a software development background, and as such are not deeply familiar with relational databases and SQL. Unfortunately, practically all books on learning relational databases and SQL assume that the reader has a background in programming.

This book makes a different set of assumptions. It assumes that the reader is not a programmer, and is not interested in becoming a programmer. Instead it assumes that the reader is a business analyst faced with a set of legacy databases with inadequate documentation. As usual the task facing the business analyst is that of understanding what is stored in the database and how the information stored in the database can be extracted to add value to the business (or improve the day job of the users of the system).

I call this approach database archaeology because it is necessary to dig deep into the database to discover how the data got to be the way it is, and then to use that information to understand the data that drives the business. After the deep dive into the database it will be easier to undertake the task of *learning the world* that is the business.

1.1 Learning The World

Currently Business Analysis is an under appreciated skill in many businesses. In part that is due to the complexity of the systems that support the businesses, which makes it very hard to predict the consequences of any changes. Rather than take the time to understand the business, many managers try to put new systems in place believing that the new system will improve things. Unfortunately, just putting in a new system complicates things even more, because there are not even more points of interaction and more systems to learn (and potentially make mistakes in).

Another large part of the problem is that historically, business analysts generated a lot of paper documentation about the business processes that were to be automated, without providing much value in terms of ideas as to how to improve and automate those processes. The end result of this was the move towards the Agile approaches to software development that minimized the role of analysts, instead trying to get the business stakeholders to talk directly to the developers. This has only worked well in those businesses where the stakeholders have a really deep understanding of their business AND the developers have a deep appreciation for the needs of the business users. In most cases the result has been partial systems that do not integrate well with anything, and user hostile systems that were easy for the developers to write (but practically nearly impossible for the users to use effectively).

This book is a call for an alternate path forward out of the mess we are in. I call the approach **Learning The World** because it asks business analysts to approach an business with fresh eyes to really get an appreciation of what would be needed from the processes and systems to improve the day job of the users of the systems.

1.2 The role of the Business Analyst

To oversimplify, business analysis ultimately comes down to two related things,

- Understanding who does what to whom and when
- Understanding what information the business needs to remember about those activities

Everything else is just an elaboration on those simple sounding statements.

> Note. Finding out the *why* of the *who does what to whom and when*, although it can be interesting to discover, it is not a part of the business analysis task. It might help with understanding the overall context, and possibly help in finding out functionality that is no longer needed, but the *why* could easily sidetrack the effort of investingating what the system does.

This book focuses on a small part of that, understanding the data that the business currently has in the existing systems, specifically the data that is stored in relational databases. From this a business analyst can discover

- Business concepts that are named in the database
- Data attributes that are associated with the business concepts
- Relationships between the concepts
- Cardinality of relationships between the concepts
- Business rules related to the concepts and relationships
- Data volumes and transaction rates
- Data access paths that are supported
- How well the system deals with change over time
- What user activities are not supported
- Business concepts that are missing or only partially implemented

1.3 Business Analysis is Important

The Cynefin framework[1] identifies multiple problem domains, only one if which, the *simple/obvious* problem domain, where the users and managers know exactly what the best practices are, does not have much need for business analysis. An example of an obvious domain is Accounting, all accountants agree that *double entry bookkeeping* is the best practice, so not much business analysis is required in this domain. Accountants can explain it clearly and developers know how to implement it.

In all other problem domains, *complicated*, *complex*, *chaotic* and *disordered*, domain experts need the assistance of business analysts to identify and articulate the issues they face and what they need their software to be able to do. The problem is that in these domains, the domain experts have unconscious competence, so although they know what to do, there are few agreed ways to talk about the domain, and no best practices.

In these domains, there is a continuing problem with software development, that of *requirements change*, but Jesse Watson has suggested an alternative formulation:

> The nature of the beast is that software requirements rarely change; what changes is our awareness of them, and our grasp of their implications.[2]

Business analysts can help the domain experts capture and express their deep domain knowledge in a way that makes it available to the software developers.

1.4 Using SQL in Business Analysis

Most organizations are currently using relational databases, so when doing business analysis for the replacement of a system, or a new system to integrate with existing systems, it is useful to be able to look at the data in those existing systems. This is an archeology task since databases are rarely well documented, and with legacy applications there may be few people in the organization who really understand what the application does and how it works.

While it can be useful to run the applications and look at how they interact with their users, being able to also look in detail at the data in the databases is a massive help in understanding the systems and the overall problem domain. For the effort of learning to query a relational database using SQL you can get a much better idea of the business concepts it supports and the limitations it imposes on the applications that use the database.

When doing business analysis for the replacement of a legacy system, being able to query the database of the legacy system and the other systems it interfaces with allows you to identify overlaps in functionality, places where data isduplicated between the systems, and any mismatch in concepts between the systems.

[1] https://en.wikipedia.org/wiki/Cynefin_framework
[2] https://www.linkedin.com/pulse/hard-thing-software-development-jesse-watson

Chapter 2

A Sample Database

Commercially there are only a few relational database technologies that have any significant market presence:

- DB/2 (IBM)
- MySQL (Oracle)
- Oracle
- SqlServer (Microsoft)
- PostgreSQL

Fortunately all of these more or less follow the SQL standard, and to a large degree, as a business analyst, what you learn about querying the data in one of these databases will translate to any of the other databases. The technology underlying these databases is however completely different so you will find that the various databases are used in completely different businesses. Organizations that have historically used IBM mainframes tend to use either DB/2 or Oracle. MySQL is normally found in startups and many web development shops, mainly because it is easy to get started with MySQL and the licensing is relatively lenient. Organizations that have adopted the Microsoft develoment stack will use SQLServer. PostgreSQL is a practical, open source alternative for organizations that use DB/2, Oracle or SQLServer.

A legacy application might use a different relational database, but in most cases the simple queries you will need for *database archeology* will work with most databases. In practice however, since the above five databases have been in the market for well over 10 years, most legacy systems that need to be replaced are usign one of the above databases.

2.1 Sample Database is in PostgreSQL

All the examples and exercises in this book were written and tested against the PostgreSQL database. PostgreSQL closely follows the *SQL Standard*, so queries written for PostgreSQL will be close to the SQL syntax required by the other databases.

Another part of the rationale for using PostgreSQL in this book is that the adoption of PostgreSQL is increasing over time. While it still has a smaller installed base than Oracle, SQL Server or MySQL, it is now seen as a viable replacement for any of those databases. In part this adoption is due to the availablity of low cost implementations on the various cloud provider platforms. This is aided by the ease of creating local installations without the need to get puchasing and licensing approval.

PostgreSQL can be installed for free on practically any laptop, regardless of whether they are running Windows, OSX or Linux. This means that it is feasible (and expected) that you the reader install PostgreSQL on yor laptop and follow along with the exercises. While you can get an appreciation for the syntax of SQL by reading someone else's SQL statements, it is only by writing SQL that you will really begin to understand what you can achieve with SQL.

2.2 Installation Instructions

Rather than include quickly outdated installation instructions the simplest way to get PostgreSQL installed is to follow the instructions at http://www.postgresqltutorial.com/install-postgresql/. After installing PostgreSQL you can also download the sample database that is on that site, as it is used in some of the examples in this book. Sample database is available from http://www.postgresqltutorial.com/postgresql-sample-database/.

> Note. Looking at the history of the DVDRental database, it has been migrated from other systems, so should be OK to use it. Also it is amusing to use from the point of view of this book since DVD Rental business has been superceded these days, so it is a good example of organizational data that is obsolete, yet contains useful information. Sent email to postgresql tutorial maintainers early March 2019, no response yet.

Will need to host the version of the DVDRental sample database on a URL related to the book likely on ImprovingWetware.com website.

2.3 Post-Installation Test

To test that the installation of PostgreSQL succeeded and that the sample database is available, please launch the pgAdmin tool and under the Tools menu, select the Query Tool and enter the following SQL query. After you have completed typing the query, click the lightning bolt icon to have the query run.

```
select * from category
order by name
```

You should see output from this query similar to that shown in Table 2.1, but with a few more rows. If however you see an error message instead, first check for obvious

Table 2.1: First five rows from category table

category_id	name	last_update
1	Action	2006-02-15 09:46:27
2	Animation	2006-02-15 09:46:27
3	Children	2006-02-15 09:46:27
4	Classics	2006-02-15 09:46:27
5	Comedy	2006-02-15 09:46:27

typos, and then read the error message to see if it gives you any useful hint as to the problem. In most cases it will try to show the line and point to the first character of the word where it could no longer understand the query syntax.

2.4 Sample Database is now available

At this point you are ready to start looking inside the database server to see what data is hidden there.

> Note. If your organization uses PostgreSQL then all you will need to do different to query data from one of your test or production databases is to get the connection information and a username/password from your database team. If your organization uses a different database technology, then you will need to download an appropriate client application and connect with the relevant connection information and username/password.

When starting out with your exploration of a database, it is always best if possible to connect to a test database server rather than the production server. This avoids any problems of putting extra load onto a possibly heavily loaded production server.

Chapter 3

Database Concepts

This chapter covers the database concepts and terminology that you need to know in order to be able to use SQL to extract data from databases. Most of it is only useful to people who need to manage the databases, but a certain amount of jargon is necessary to talk to the other users of the database. For the most part the terminology is intended to clarify meaning, but it can be confusing the first time you come across it.

Rather that going too far into defining the terminology, it is simpler to just use it and allow you to understand the meaning from the context of the discussion. Precision is necessary sometimes, but most of the time when we use a word like database, from the context we can determine whether we mean

- the brand of database software we are using
- the database server we are connecting to
- the database instance or schema we are interacting with

Although we can interact with most databases from various front ends, in practice it is often easiest to use the client software that comes with the database server to connect to the database. With PostgreSQL that means that we either use the commandline `psql` or the graphical `pgAdmin` to connect to our database server. To connect to the server you typically need to know which machine it is running on, the network port to use, a username and password and the name of the database schema that we want to interact with. (Just for confusion, some database brands use the term instance where postgreSQL uses schema).

3.1 Database Schemas

Most database servers have multiple databases running inside. Any useful postgreSQL server will have a minimum of two databases, `postgres` which holds system related information and the application specific database, `dvdrental` if you have installed the sample from the previous chapter.

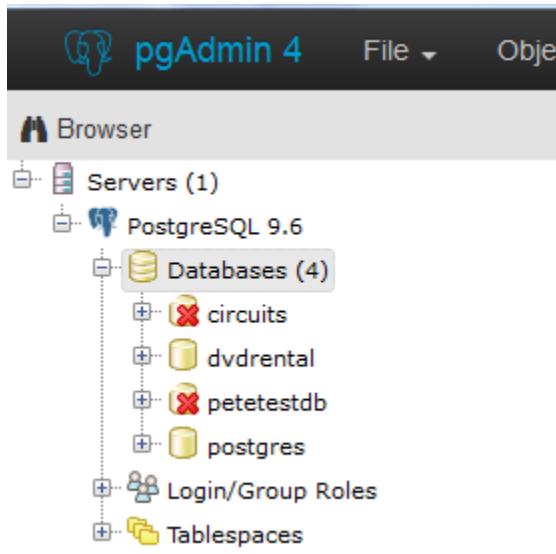

Figure 3.1: databases

As a practicing Business Analyst, the first thing you want to know about every database server is which application databases are running on that server, as well as some valid credentials that will let you explore those databases. The server browser in `pgAdmin` makes this trivially easy, because it shows the databases that are running in the database server.

In this example there are 4 databases, two of which are not connected, plus dvdrental and postgres.

In a corporate context, a Business Analyst will only be granted access to the databases that are needed for the job at hand, so get used to the idea of seeing the names of a lot of other databases that you do not have access to.

If the application designers did a reasonable job, an application will use one or more databases to store the data related to that application. There may be multiple databases if the application is split up into fairly distinct modules, or it might all be contained within a single database. Older applications will tend to have a single database, but the idea of using multiple databases to separate out differenct subsets of data is becoming a common idea. Having multiple databases also makes it easier to control access to the data by different users. Access to specific tables can be granted to an individual user, but it is much simpler for the database administrators to just grant complete access to a database while restricting access to another.

3.2 Database Tables

A relational database stores the data in tables, and any reasonably complex application will have many different tables, typically hundreds of tables, which makes the `dvdrental` database look like the toy it is. Each table within a database has to have a unique name, and hopefully that name has some meaning with in the application domain. Sometimes we are lucky and this is relatively true, other times the table names seem to have been purposefully made either cryptic or generic. Hopefully however by reading over the table names you can get a sense of the information that the application is remembering for the business.

The simplest way to think of a table is as a spreadsheet with a twist. Each column in the spreadsheet has a unique name (that hopefully is meaningful in the application domain), AND a specific datatype (typicaly numbers, date/time or text) potentially with interesting constraints (e.g. cannot be `null`, range or size limits, unique values). Unlike spreadsheets, tables can have an unlimited number of rows, constrained only by the speed of access and the usefulness of massive volumes of data. Within a table, one column is normally designated as the *primary key*, and that column is constrained to have guaranteed unique values – the database server will prevent anyone from creating a row in that table with the same value for the primary key.

These primary keys are what make a relational database interesting for applicaton development. As part of defining the columns in a table, you can define a *foreign key* relationship with another table, whereby the column is constrained to only contain values that are in the *primary key* column of another table. So a tables of `Cities` could have a `Country_id` column that is contstrained to only contain values from the primary key column of the `Countries` table which would be called either `id` or `country_id`.

> Relational databases have been around long enough that we could reasonably expect that there would be an agreed convention for table and column names, but of course there is not. Some database designers make their table names plural nouns, others make them singular (as in the `dvdrental` example). Some make the primary key column name be just `id`, others use the singular version of the table name with a suffix of `_id`.

3.3 Relationships between tables

The defined relationships between tables are what make relational databases interesting. Each distinct type of information that a business needs to remember is stored in a specific table, and if the database designer has thought to enforce referential integrity by foreign keys, we can guarantee that if we see a `film_id` in the `inventory` table, then the associated information about the film exists in the `film` table.

Some database designers choose not to enforce this referential integrity with foreign keys, in which case the film might not actually exist. True the programmers are

meant to ensure that the appropriate records are created, but as everyone knows, applications are not always correct, so bad stuff can happen. In the old days, not enforcing the referential integrity with foreign keys meant that you could get slightly better insert and update performance in the database, but that is irrelevant with current computers. Yes, it might still help in the margin, but even on a cheap server it is trivial to insert over 1,000 rows per second. Few commercial systems need that sort of performance, especially when we consider that the tradeoff is incorrect data in the database.

3.4 Third Normal Form Database Tables

Relational databases are not supposed to have repeating groups in a table. Admittedly some database designers will try to cheat this by having column names in a `Orders` table like `item1`, `item2`, `item3` but this makes it really awkward if you end up needing `item4`.

To work to the strengths of relational databases, what is done instead is to have two tables, an `Orders` table and an `OrderItems` table with a foreign key relationship back to the `Orders` table. The database can then store an unlimited number of items for an order.

Overall the main design principle for relational databases is that as a minimum the database tables should be in *Third Normal Form*, where each column in a table depends on the primary key, the whole key and nothing but the key. There are higher normal forms, where it can seem that all we can find in any row of a table is a single value and a foreign key, but as far as querying the database, the queries are the same, we just will have to select data from more tables when the tables are higher than *Third Normal Form*.

Might need more details here or a reference

Chapter 4

Basic SQL

Structured Query Language (SQL) is the main way of accessing a relational database. For our purposes, the only verb that a Business Analyst needs for doing database archeology work is the `select` verb of `SQL`. If we were going to also consider modifying the contents of a database, we would also need to be concerned about the `insert`, `update` and `delete` verbs, but for the initial business analysis tasks, we do not need to know them yet.

At the most basic, a `SQL` query just selects a number of rows from a table, potentially with ordering applied to the result set as per this example shown in Table 4.1.

```
select * from category
order by name asc
offset 4
limit 3
```

This query also has a `limit` clause that limits the number of rows returned in the result set from the query. This is very important when first exploring a database, because we might be executing a query on a large table, and returning a few million rows when we are just trying to see what kind of data is in the database is not very useful. *PostgreSQL* has basic windowing built into the SQL queries using the `limit` clause, but means of the `offset` keyword. Specifying `offset 4` means that the returned result set will skip over the first 4 rows and start showing the data from row 5 onwards. The database server still has to process the query in detail to get to the 5th row, so there are better ways to limit the result set, but as a quick and dirty option this works well.

The `select * from category` part of the query requests every column from the `category` table. Normally you would explicitly name the columns that you are interested in as a comma separated list, but specifying every column is a convenient starting point when you are just trying to get a sense of what is in the table.

The `order by` clause can name one or more columns, and for each column the ordering can be `asc`endin or `desc`ending, the default is `asc`. When specifying multiple columns for the ordering, the column names must be separated by a comma, `order`

Table 4.1: 3 records

category_id	name	last_update
5	Comedy	2006-02-15 09:46:27
6	Documentary	2006-02-15 09:46:27
7	Drama	2006-02-15 09:46:27

`by name asc, last_update desc`.

For a large result set the sorting necessary to fulfil the ordering can be a slow operation. Normally a query just returns the first rows it finds, so a simple query on a million row table is fast when we limit the result set to just three rows. When there is an order by clause, the query can still be fast if there is an appropriate index that the database server can use to traverse the data in the required sequence. Without a suitable index, the database server has to read the entire million rows and then sort the data by the columns specified in the `order by` clause. The result set may still come back in reasonable time, but when first investigating a table, omit the `order by` clause unless you know that the result set is going to be small or there is an appropriate index.

From the viewpoint of *Database Archeology* we will not be creating any new indexes to support any slow running queries. Any new index in a database has to be approved by the appropriate *Database Administrator*, because in a corporate context, you have to know who is making changes to the structure of the database. For our purposes the thing to remember is that when we have two seemingly similar queries that take completely different times to return the result set, the faster query is likely to be supported by the existing indexes, or is on a very small table which the database server can load into memory.

> Note. To a large extent, the available indexes determine the *Data Access Paths* that are supported by the application. For small tables, say less than 1 million rows, a complete scan of the table might complete in reasonable time. For larger tables, a complete scan looking for relevant records is likely to be too slow for most users, effectively making that *data acccess path* unusable.

4.1 Selecting specific columns

In the previous query, the `*` shows all the columns in the table `category`, but typically we only need that option when we are trying to discover what data is in a table or with a smaller table when we are feeling lazy. In most queries we will be specifying the specific columns we want to see the data for as in Table 4.2. The sequence of the columns that are selected does not have to match the sequence in the table, so in this example the `name` is the first column selected.

By default the selected column name is used as the label in the result set, but `SQL` allows *column labels* to be applied to the result set. These column labels look like `category_id as id`, which applies the label `id` to the `category_id` column

Table 4.2: 3 records

name	id
Sci-Fi	14
New	13
Music	12

name. These labels are useful for replacing the original column name with a more descriptive name

```
select name, category_id as id from category
order by id desc
limit 3 offset 2
```

4.2 Choosing the data using the `where` clause

The `where` clause in a `select` statement allows us to specify the data that we are interested in from the table. At the most basic level we just specify the column values we are interested in the `where` clause. For number and date columns, we can test for equality `=`, less than a value `<`, and more than a value `>`.

```
select name, category_id as id from category
where category_id = 3;

select name, category_id as id from category
where category_id < 3;

select name, category_id as id from category
where category_id > 3;
```

Sometimes we will end up wanting to select records matching more than one value, in that case the simplest way to do this is to use the keword `or` to link the clauses. We can also use the keyword `and` to link the clauses if the values are in different columns.

```
select name, category_id as id from category
where category_id = 3
  or category_id = 4;

select name, category_id as id from category
where category_id = 3
  and name = 'Children';
```

> Note. It is safest to always bracket the conditions when you have a mixture of `and` and `or`, because there is a lot of difference in the results between `(A and B) or C` and `A and (B or C)`. Without the brackets it is really easy to waste a lot of time trying to figure out why the result set is not quite what we expect.

Another way of selecting multiple records from the same column is to use the `in` keyword. This is functionally equivalent to a set of equality tests linked by the `or` keyword, but much easier to read and understand. Later on we will see more ways to use and abuse the `in` keyword, but for now we will restrict our usage to literal values.

```
select name, category_id as id from category
where category_id in (3,4);

select name, category_id as id from category
where name in ( 'Children', 'Action');
```

4.2.1 Excluding specific values

`SQL` has multiple ways of specifying not equal to, but if we see anything other than the first two forms below it is a sign that the person who wrote the query needs to be taken away from their keyboard. The normal recommendation, when faced with two bits of syntax that have exactly the same meaning such as `!=` and `<>`, is to pick one and **never ever** use the other. Using a mixture of the two is more or less a guaranteed path to confusion.

```
select name, category_id as id from category
where category_id != 3;

select name, category_id as id from category
where category_id <> 3;

select name, category_id as id from category
where category_id not in (3,4,5);

select name, category_id as id from category
where not category_id = 3;
```

4.3 Partial matching on text columns

When working with text columns, we will often need to do a partial match and *PostgreSQL* provides plenty of options for this

- `like`
- `similar to` regular expressions
- *POSIX* regular expressions

Most of the time we will only ever need to use `like`, but there is occasionally a need for `similar to` as will be seen in a later chapter. Business Analysts should stay a long way away from *POSIX* regular expressions unless they aspire to becoming a *UNIX* programmer.

Table 4.3: 2 records

name	id
Action	1
Animation	2

The `like` expression does a pattern match based on the idea that `%` matches any sequence of zero or more characters, and `_` matches any single character. So the queries below find all rows where the name begins with the uppercase letter 'A', all rows that have a name containing a lowercase letter 'a', and the last one finds all rows where the name has a lowercase letter 'a' in the second position.

```
select name, category_id as id from category
where name like 'A%';

select name, category_id as id from category
where name like '%a%';

select name, category_id as id from category
where name like '_a%';
```

The difference between the last two queries is that `_` matches a single character, meaning that it is checking for an 'a' in the second position of the string, while `%` matches any number of characters, so it will match a string with 'a' in the first, second, third, fourth, etc positions. It will not match against the name 'Action' however, since strings in *PostgreSQL* are case sensitive, but it will find 'Animation' since there is an 'a' further down the string.

Although , the matching can be made case insensitive using the keyword `ilike`, so this query finds any row where the name starts with a letters 'A' or 'a'. Table 4.1

```
select name, category_id as id from category
where name ilike 'a%';
```

There is of course a `not` like option, the first one is the more conventional way of writing it, but the negation form shown second gives an equivalent result. The third query with the criteria in brackets, `(name ilike '%a%')`, makes it clearer what is happening.

```
select name, category_id as id from category
where name not ilike '%a%';

select name, category_id as id from category
where not name ilike '%a%';

select name, category_id as id from category
where not (name ilike '%a%');
```

4.4 Indexes on Tables

Although relational databases do not require any table indexes, to get realistic performance, indexes are required to speed up access. For each criteria in a query, the database has to scan through the table checking which values in the table rows match the criteria. This works if the table is small enough to be held in memory, but when the tables are larger, a full scan takes O(n) time. This effectively means that if a scan though a 1 million row table takes 12 seconds, then scanning through a 10 million row table would take about 2 minutes.

When the table has an index on a column, the index is effectively a shortcut way to access individual values, giving an access time of the order of O(ln(n)). This means that if searching for a row in a 1 million row table takes 1 second, searching a 10 million row would maybe take 1.2 seconds.

Typically you will find an index on all columns that are named in a `where` clause of a query, so the presence of an index highlights the existence of a common *data access path*.

Chapter 5

Basic Database Archeology

The most basic question that you will first need answering about a database is the *Business Concepts* that it supports. To get the anser to this, you need to be able to get a list of all of the table names that are in the database. Unfortunately this is implemented in a database vendor specific manner, so you will need to look up in the documentation how to list out the table names.

> Note. The `pgAdmin` GUI shows the list of tables in the `DVDrental` database, but sometimes you will want to extract the names of the tables to use elsewhere.

For the example PostgreSQL database, the SQL is relatively simple to list out the catalog (`dvdrental`) for the `public` schema results are in Table 5.1.

```
select table_catalog, table_schema, table_name
from information_schema.tables
where table_schema = 'public'
  and table_type= 'BASE TABLE'
order by table_name
limit 6
```

> Note. Obviously when you run this query, you will not want to limit the result set to just six rows, but constraining the result set makes it easier to show in this book. Real applications are likely to have anything from 20 to 2000 tables, but for your sake I hope that you never have to deal with a 2000+ table database.

5.1 Business Concepts

Looking at the full list of table names returned by this query, most of the concepts you would expect to see are there `customer`, `rental`, `store` and the missing `dvd` concept is likely split between `film` and `inventory`.

Table 5.1: Tables in DVDrental

table_catalog	table_schema	table_name
dvdrental	public	actor
dvdrental	public	address
dvdrental	public	category
dvdrental	public	city
dvdrental	public	country
dvdrental	public	customer

The two strange table names `film_actor` and `film_category` represent the way that a relational database needs an extra table to represent a many to many relationship when constrained to be in *Third Normal Form*. So rather than having a list of `actor`s inside a `film` and a list of `film`s in every `actor`, there is a separate table that lists the pairings of (`film`, `actor`) and (`film`, `category`).

As noted in Chapter 3, in the section *Database Tables*, the table names in your database may be relatively clear, cryptic or generic, so you are likely to have to make notes about some table names to explain the udnerlying business concept. If you come across some table names that look pseudo-random (`xrq189576786`) or have what could be a date in the name (`actor_20061203`) you will likely need to have a conversation at a later date with your DBA or one of the developers to understand how that table is used. For now though just note the strange name and leave a question mark beside it.

Table	Business Concept
Customer	Person who rents a DVD from one of our stores
Film	Title of a DVD, will have zero or more copies of a film in the inventory of a store
Inventory	A physical DVD that can be rented from a store, details about the film are stored in the film table
film_actor	?
film_category	?

5.2 Data Attributes

Once you have identified the main business concepts, the next thing you need to do is look at the data attributes associated with each of the concepts. As with getting a list of the tables, the list of columns and their data types is database vendor specific.

```
SELECT table_name, column_name, is_nullable as nullable,
  udt_name, character_maximum_length as size
FROM information_schema.COLUMNS
WHERE TABLE_NAME = 'customer'
```

In Table 5.3 you can see that this database uses the `customer_id` approach for the

Table 5.3: Customer Table Columns

table_name	column_name	nullable	udt_name	size
customer	customer_id	NO	int4	[null]
customer	store_id	NO	int2	[null]
customer	first_name	NO	varchar	45
customer	last_name	NO	varchar	45
customer	email	YES	varchar	50
customer	address_id	NO	int2	[null]
customer	activebool	NO	bool	[null]
customer	create_date	NO	date	[null]
customer	last_update	YES	timestamp	[null]
customer	active	YES	int4	[null]

Table 5.4: Customer Table Column Defaults

table_name	column_name	column_default
customer	customer_id	nextval('customer_customer_id_seq'::regclass)
customer	activebool	true
customer	create_date	('now'::text)::date
customer	last_update	now()

You will need to build you a data dictionary for the set of tables that you are interested in. The output from the query for each table can be used as the starting point, with extra details added as you discover more information about the way that the column is used. Some columns may not need any more information that just the column name, others should be left with a `?` if the reason for that column is unknown as a hint for future conversations with the development team.

5.3 Default Values

Another bit of information that can be extracted from the database schema is information about the default values (if any) for the various columns. Table 5.4 shows that the `customer_id` is populated automatically using the next value from a database sequence `customer_customer_id_seq`[1]. This guarantees that the customers are numbered uniquely, and there does not need to be any application logic to ensure that the customer id numbers are unique. The date columns are also set to the current time (`now`), the different formats for the default being because one is a date field and the other is a timestamp (which includes the date and time).

```
SELECT table_name, column_name, column_default
FROM information_schema.COLUMNS
WHERE TABLE_NAME = 'customer'
  and column_default is not null
```

[1] `nextval` advances the sequence and returns the new value https://www.postgresql.org/docs/9.6/functions-sequence.html

5.4 Looking at the Data

At this stage, with a complete list of the tables in your database, it can be useful to look at the first few rows in each table, just to get a sense of what is in each table.

```
select * from customer
limit 10;
```

Just running a query like this on each table in the domain gives you a feeling for the overall domain. The query result will give you a feel for the difference between the `date` column `create_date` ("2006-02-14") and the `timestamp` column `last_update` ("2013-05-26 14:49:45.738") in the `customer` table.

If you learn anything interesting about the data in the tables, then it can be good to go back to the data dictionary you are building for the tables and add that information into your data dictionary.

Chapter 6

Pulling Data From Multiple Tables

The real power of SQL comes from the ability to select data from multiple tables and combine it in interesting ways. This is done in a query by specifying a `join` condition that specifies which rows are wanted from the related table. We specify the rows we want by specifying which columns are expected to have what values, just like the `where` clause, but when relating tables we use a `join` clause.

The simplest form of a join is an *Inner Join* where rows from the joined table are only included if they meet the criteria specified. In the example below, a `city` is only shown if there is a match on the `country_id` column with the current `country` row value, as shown in Table 6.1. The `inner` keyword is optional and is normally omitted, so the query just has the `join` keyword rather than the more verbose `inner join`.

```
select country.country,
  city.city
from country
join city on city.country_id = country.country_id
limit 5;
```

> Note. It is important at this point to always prefix any column name with the name of the table that the column belongs to. In the above example, it is not stictly required by *PostgreSQL* that you prefix the column names in the select list, but if we omit the table names in the `join` clause there will be an error of the form `column reference "country_id" is ambiguous`. But we should prefix the column names even when *PostgreSQL* does not require it, so that our future selves, when looking at the query a few days later, will know which table the column is being selected from. In this simple toy query the table names are obvious, but we need to cultivate the habit now so that we do not run into problems later with more complex queries.

Table 6.1: Displaying records 1 - 5

country	city
Afghanistan	Kabul
Algeria	Batna
Algeria	Bchar
Algeria	Skikda
American Samoa	Tafuna

Table 6.2: Inner Join results

country	city
Canada	London
United Kingdom	London

With an *Inner Join* the result set only includes rows where the join condition is satisfied. So a country without any cities would not appear in the result set. More importantly it excludes the Countries when the join condition is more complex, as in this query that finds just those countries where there is a city named 'London' as in Table 6.2.

```
select country.country,
  city.city
from country
join city on city.country_id = country.country_id
  and city.city = 'London'
limit 5;
```

All of the `where` clause constructs from the *Basic SQL* chapter are valid for use in a `join` clause, so we can filter the result set based on any column in the `join`ed table.

6.1 Other types of `joins`

There are several, but for now we will only look at the *Inner Left Join*, which includes the rows that are missing from the *Inner Join* because there are no matching rows in the joined table. For a `left join` the only difference from the previous query is the addition of the work `left` which requests that the query return rows from the main table even when the joined table does not have any matching rows. The results in Table 6.3, where we only return the values for the `city` table when the city is called 'London', otherwise we return `null` which is the *SQL* way of stating that there is nothing there.

```
select country.country,
  city.city
from country
left join city on city.country_id = country.country_id
```

Table 6.3: Left Join results

country	city
Afghanistan	[null]
Algeria	[null]
American Samoa	[null]
Angola	[null]
Anguilla	[null]

```
  and city.city = 'London'
limit 5;
```

> Warning. `null` is weird. It denotes the absence of data, so it has some weird properties. We cannot compare a value to `null` like we can with any other value a where clause of `where city = null` will never return any rows, neither will `where city <> null`. The only way we can test for the presence of `null` is to use the special `is` operator, as in `where city is null` or `where city is not null`. There is other weirdness associated with `null`, but that will be covered later. The null columns are displaying as 'NA' in the book, seems to be an artifact of how R treats the null value, to resolve before sending out copies edit the ltw.tex file and compile from the top level directory so that it can see the images etc. This generates a correct pdf with null displayed (just have to replace all the NA with null and remove this part of the paragraph from the .tex file.

6.2 Joining multiple tables

The beauty of relational databases is that *SQL* allows us to follow the relationships between the tables to pull out the information we are interested in. To do this we just add in another `join` clause specifying the table to be joined into the query and the condition that we want to include the data with. The sample below finds the street addresses in all cities called 'London' in all countries. The results in Table 6.4.

```
select country.country,
  city.city,
  address.address,
  address.district
from country
join city on city.country_id = country.country_id
  and city.city = 'London'
left join address on address.city_id = city.city_id;
```

We can keep on adding extra table to our query by just adding an extra `join` clause for each table. As we add each extra table the database server has to do more work to find the rows we are interested in, so the performance will degrade somewhat,

Table 6.4: 3 records

country	city	address	district
United Kingdom	London	1497 Yuzhou Drive	England
United Kingdom	London	548 Uruapan Street	Ontario
Canada	London	NA	NA

but the worst case is that we have to wait a while for the data to be returned by the query. The upper limit tends to be limited by what makes sense as far as the application domain is concerned, and our understanding of the relationships between the various tables.

6.3 Table aliases

When reading complex queries, we often come across table aliases. A table alias has the same effect as a column label, in that it introduces a new name for the table. Although providing an alias for a table can save typing when the table names are long, aliases can also add confusion when reading a *SQL* query. All the alias does is introduce another name that has to be remembered and referenced back to the original table so that we know what it represents. The query below is functionally the same as the one above, but it is harder to comprehend because of the extra names that are introduced. The results are the same as those in Table 6.4.

```
select cntry.country,
  cty.city,
  addr.address,
  addr.district
from country cntry
join city cty on cty.country_id = cntry.country_id
  and cty.city = 'London'
left join address addr on addr.city_id = cty.city_id;
```

My personal preference is to initially use the complete table name without any aliases, and only when I am really familiar with the database design to adopt a few abbreviations for the most commonly queried tables. Single letter abbreviations, although we see them in programmer queries all the time are a mistake – does 'c' refers to `city`, `country`, `customer` or `category`.

When the selected columns make the first line of a query too long to read easily, split it up over multiple lines with each selected columm on a separate line, as has been done since we left Chapter 4 *Basic SQL*.

Chapter 7

Database Archeology - Relationships

With Relational Databases in the *Third Normal Form*, there is a lot of useful information that can be gained by looking at the logical connections between the various tables. These connections are made between values in columns of a table and the value in a *Primary Key* of another table. These logical connections can be enforced by the database as *Foreign Keys*, or they can just be a connection that is created by a `join` clause in a query.

The *cardinality* of the relationship between tables reveals useful information about some *Business Rules* encoded in the application. Optional relationships are supported in databases by allowing the relevant column to be *nullable*. Databases typically do not support upper limits to the cardinality of a relationship, but an upper limit may be encoded in the application code.

Cardinality	Example Relationship
1:1	Each store must have one and only one manager
1:0..1	If a film specifies a language, it can only specify one language
1:1..n	Each store must have at least one staff working there
1:0..n	For any film, there may be zero or more copies in inventory

As far as the *Business Rules* are concerned, it is important to know whether a relationship is optional, and whether the upper limit for the cardinality is one or many. You should note however that many relationships that initially look to have an upper bound of one turn out over time to have more than one associated records. So yes, a store must have one and only one manager, but over time many different people could manage a different store.

Many older systems have this type of cardinality problem, whereby the current state of the system is supported by the database, but information about past states of the system cannot be recorded. Similarly future planned changes to the state of the

Table 7.2: Primary Keys

table_name	primary_key_cols
actor	actor_id
address	address_id
category	category_id
city	city_id
country	country_id
customer	customer_id
film	film_id
film_actor	actor_id, film_id

system are not supported. For example, can we have a `film` in our database that is not released yet, and how would we know that it is not released?

7.1 Primary Keys

The one side of relationships in relational databases is typically to the primary key of a table. This primary key is normally a unique value, often the primary key is on just a single column, but it can be across multiple columns. Depending on when youir database was designed, the keys will either be artificial identifiers (an integer or a *GUID*), or meaningful business values. In the `DVDrental` sample database, all primary keys are made up of the artificial identifiers (with an interesting mix of 2 and 4 byte integers).

The query to get the primary keys contains a nested sub-query as part of the list of queried parameters, and this sub-query uses `string_agg` to concatenate the columns used in the primary key index. string_agg

```
select table_name,
  (SELECT string_agg(kcu.column_name, ', ' order by ordinal_position)
  FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE kcu
  where kcu.constraint_catalog = tc.constraint_catalog and
    kcu.table_name = tc.table_name and
    kcu.constraint_name = tc.constraint_name) primary_key_cols
from INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
where tc.table_schema = 'public'
and tc.constraint_type = 'PRIMARY KEY'
order by table_name
limit 8
```

The results of the Primary Keys query are in Table 7.2. For the most part the Primary Key column is named after the table name with a suffix of `_id`, and as we saw in Chapter 5 in the section on Default Values, these are all populated using a `nextval` from a database sequence. Another common idiom for maning the primary key is just to call the primary key column `id`.

The `film_actor` table has a composite key over the `actor_id, film_id` columns

to resolve the business problem that an actor can be in multiple films, and many actors will appear in a single film. The `film_actor` table resolves the problem of implementing many:many relationship between `actor` and `film` in a relational database. The composite key ensures that an actor can only appear once in a film.

7.2 Foreign Keys

Databases normally use the term *foreign key constraints* to describe *Foreign Keys*, and these constraints basically enforce that the `address_id` value that exists in the `customer` table also exists in the `address_id` column in a row in the `address` table.

The query to identify the Foreign Keys in the `DVDrental` database is more complex than the primary key query, as it needs to pull in two different sets of column names that define the relationship between the tables. The results in Table 7.3 identify the Primary Key table and the associated Foreign Key table.

```
select ut.table_name as PK_Table,
  (SELECT string_agg(kcu.column_name, ', ' order by ordinal_position)
   FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE kcu
   where  kcu.constraint_catalog = ut.constraint_catalog and
     kcu.table_name = ut.table_name and
     kcu.constraint_name = ut.constraint_name) Primary_Key,
    tc.TABLE_NAME as FK_Table,
 (SELECT string_agg(kcu.column_name, ', ' order by ordinal_position)
  FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE kcu
  where kcu.constraint_catalog = tc.constraint_catalog and
    kcu.table_name = tc.table_name and
    kcu.constraint_name = tc.constraint_name) Foreign_Key
from INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS rc
join INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
  on tc.CONSTRAINT_NAME = rc.CONSTRAINT_NAME
join INFORMATION_SCHEMA.TABLE_CONSTRAINTS ut
  on ut.CONSTRAINT_NAME = rc.UNIQUE_CONSTRAINT_NAME
where tc.table_schema = 'public'
order by ut.table_name, Primary_Key
limit 10
```

There are 18 foreign key relationships defined between the 15 tables in the `DVDrental` database, so keeping all these relationships in your head at the same time can be difficult. The way to work around this is to focus on a specific path through the data that you are interested in. For example, both a `customer` and a `store` both have an `address`, which has a relationship to `city`, so it should be possible to write a query that finds out all customers who do not live in the same city as their store.

> Note. From the listing of the Customer Columns in Table 5.3 you can see another possible relationship defined for a `customer` as it has a `store_id` column. This relationship is not defined by a foreign key, as it does not

Table 7.3: Foreign Key Columns

pk_table	primary_key	fk_table	foreign_key
actor	actor_id	film_actor	actor_id
address	address_id	staff	address_id
address	address_id	store	address_id
address	address_id	customer	address_id
category	category_id	film_category	category_id
city	city_id	address	city_id
country	country_id	city	country_id
customer	customer_id	payment	customer_id
customer	customer_id	rental	customer_id
film	film_id	film_category	film_id

appear in the results in Table 7.3. Since the `store_id` column is not nullable, so it must have a value, you can write a query between `customer` and `store` joining on the `store_id` to test if this logical relationship really exists. Further conversations would be needed to discover if the foreign key relationship is missing or whether the developers thought it was not needed for the purposes of the application.

7.3 Table Indexes

Looking at the table indexes is a vital part of database archeology, as the indexes identify the data access paths that have been optimized for the application. Each index identifies a path through the database that is used by the application and needs to be fast.

Primary Keys are automatically unique indexes in most databases, so as long as you know the value of the primary key (`film_id`) for a `film`, you can look up the details of that row quickly. But users do not know the internal identifiers for the table rows, so alternate indexes are needed. These indexes will typically be on *business identifiers* or *business keys*.

The query to access the indexes in a *PostgreSQL* database needs to access the `pg_catalog` catalog, since this information is not available from the `INFORMATION_SCHEMA` catalog. The query reports the columns covered by the index and whether the index enforces *unique* values. The sequence of columns is an important factor in a multi-column index because a query against a multi-column index will not be used if the query does not specify a value for the first column from the multi-column index. Some indexes are in Table 7.4

So with the `rental` specifying a `rental_date, inventory_id, customer_id` index, so the index can be used when searching for a particular date, but if the query specifies only the `customer_id` then this index would not be useful, as the database would have to scan the entire index to check the third field for the value of the `customer_id`.

Table 7.4: Table Indexes

table_name	idx_cols	unique
customer	address_id	FALSE
customer	last_name	FALSE
customer	store_id	FALSE
film	fulltext	FALSE
film	language_id	FALSE
film	title	FALSE
rental	inventory_id	FALSE
rental	rental_date, inventory_id, customer_id	TRUE
store	manager_staff_id	TRUE

```
With idxcols as (
 SELECT tab.oid as tab_oid,
  tab.relname as table_name,
  idxnam.relname as index_name,
  idx.indisunique, UNNEST(idx.indkey) as indatt
 FROM pg_catalog.pg_namespace ns
 join pg_catalog.pg_class tab on tab.relnamespace = ns.oid
    and tab.relkind = 'r'
 join pg_catalog.pg_index idx on idx.indrelid = tab.oid
    and idx.indisprimary = false
 join pg_catalog.pg_class idxnam on idxnam.oid = idx.indexrelid
 where ns.nspname = 'public'),
cols as (
 select tab_oid, table_name, index_name, indisunique, indatt,
  row_number() over() as idx_order
 from idxcols)
select
  cols.table_name,
  string_agg(attr.attname, ', ' order by idx_order) as idx_cols,
  cols.indisunique as unique
from cols
join pg_catalog.pg_attribute attr on attr.attrelid = cols.tab_oid
    and attr.attnum = cols.indatt
where table_name in ('customer', 'film', 'rental', 'store')
group by table_name, index_name, indisunique
order by table_name, idx_cols, indisunique
```

Note. This query has two new bits of SQL that are needed to deal with the `pg_catalog.pg_index` table[1], `UNNEST` and `row_number() over()`. These are needed due to the datatype of the `pg_index.indkey` column that is an `int2vector`, which basically encodes an array of short integers into one column (basically violating the *Third Normal Form*). `UNNEST` generates the extra rows needed to split out the values from the array,

[1] https://www.postgresql.org/docs/9.6/catalog-pg-index.html

Table 7.5: English Language Films

name	title
English	Academy Dinosaur
English	Ace Goldfinger
English	Adaptation Holes
English	Affair Prejudice
English	African Egg

and the `row_number() over()` generates a row number in sequence, so that the order of the index columns is preserved when the query uses `string_agg`.

The query results show that there is an access path that covers the other end of the *Foreign Keys*. That is why the `film` table has a `language_id` index, this will allow the set of English language films to be found by this index as shown in Table 7.5.

```
select language.name,
  film.title
from language
join film on film.language_id = language.language_id
where language.name = 'English'
order by film.title
limit 5;
```

The *Unique* indexes identify business rules. The `store` index on `manager_staff_id` enforces the rule that a manager can only manage one store, since that staff_id value can only appear once in the `store` table.

The indexes on the `rental` table reveals the queries that are possible, the `inventory_id` index means that it is possible to write a query to discover all the times that DVD was rented. There is no equivalent index on `customer_id`, so querying for all the DVDs that a customer has rented will require a table scan.

Chapter 8

Entity Relationship Diagrams (ERD)

A key challenge in understanding a database is figuring out the relationships between the various tables. From the list of relationships in Table 7.3, it is obvious `city.country_id = country.country_id` is a possible join criteria.

```
select country.country,
  city.city
from country
join city on city.country_id = country.country_id
limit 5;
```

But a picture makes it much more obvious. In this style of diagram, an *Entity-Relationship Diagram* (ERD) conventionally the table name occurs in the top box of each table symbol, and the column names occur below the table name. In this variant of an ERD the *Primary Key* of the table is in the box below the table name, and any *Foreign Keys* are listed below the primary key (and the crowsfoot symbol appears against the line). Then below that the rest of the table columns are shown.

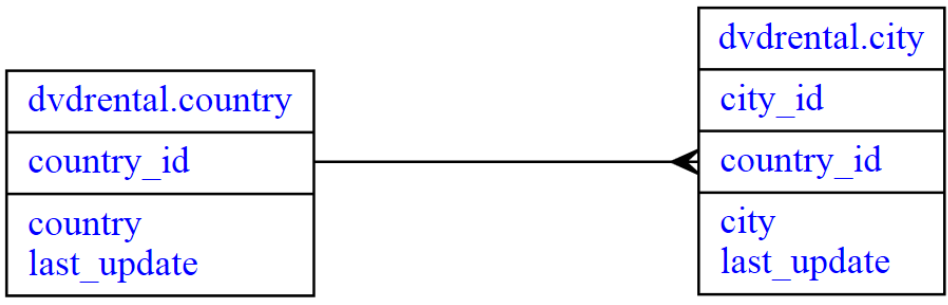

Figure 8.1: Country City Relationship

The relationshps between the tables are shown as lines between the tables, the lines link a primary key in one table to the foreign key in another table. The lines are written with a *crows foot* on the foreign key side of the relationship to suggest that there may be more than one row on that side of the relationship. So in the example below, for each row in the `country` table, there can be many rows in the `city` table.

The presence of the foreign key relationship implies that every `city` must exist within a `country`, but it is possible for a `country` not to be associated with any rows in the `city` table. So on the crows foot end of the relationship there are zero or more `city` rows for every `country` row. Sometimes the relationship will only make sense if crows foot end of the relationship has one or more rows, for example a Sales Order is incomplete if there are no Ordered Items associated with the Sales Order.

The practical difference between these two types of foreign key relationships when we are writing queries is that with *One or More* relationships we can use a plain `join`, but with *Zero or More* relationships we might need to use a `left join`. We would use a `left join` when we are primarily interested in reporting on the `country` rows, but if the focus of the report is on the `city` rows, then `join` works best since every `city` row is associated with a row in the `country` table.

A `left join` will always return at least as many rows as a `join`, but will return more rows when the equivalent row in the `city` table is not in the database. The differences between these two are shown in the results in Table 6.2. and Table 6.3.

8.1 Full Size Entity Relationship Diagrams

Any interesting database is going to have a LOT of tables, small applications can have over 50 tables, larger applications can easily have hundreds of tables, so it is important to have a means of visualizing the relatonships between these tables. Even with a small database like the *DVDRental*, it is easy to lose your place when following the relationships between the tables. The solution to this is to have a diagram that shows the ALL links between ALL the tables, as in https://improvingwetware.com/files/DVDRental.html. The picture in Figure 8.2 is a static version of the interactive diagram. Note to selves, @pageref(fig:DVDERD) does not work as a second type of link

The interaction that the web version of the diagram allows is that you can select the links and the tables (click on a relationship line to turn the line red, click again to remove the highlight.)

A printed version of the diagram does not really work for anything except wall artwork. As the printed *DVD Rental* ERD shows, if it is small enough to fit on one page, the print size borders on unreadable, and following a chain of relationships to visualize a query with multiple joins is difficult.

Workarounds that have been tried include:

- using large format printing to print the diagram big enough so that it is readable,

Figure 8.2: Sample DVDRental Data Model

- just showing the table names, leaving the rest of the detail off the diagram,
- drawing multiple diagrams, each centered on a single key table and the relationshps to that table.

None of these work very well, a large format diagram means we need a suitable wall near to the team working on the database, just showing the table names omits the key information as to what column names are involved in each join, and multiple diagrams means that we lose the big picture.

8.2 CASE Tools

One option that works in some circumstances is to use a *CASE tool* that has good *Data Modelling* capabilities. Since the early 1990's CASE tool vendors have been claiming that their CASE tool can connect to a database and *Reverse Engineer* the data model to produce an ERD. To some extent these claims were true, the definition of the tables could be imported, we could examine the definition of a table and regenerate the database on a different server or even on a different vendors database.

What they could not do however was produce a decently readable diagram for anything but small, toy databases. One early tool could produce a reasonable diagram for a sample, ten table database, which was great for demonstrating the capabilities of the tool. When it was connected to a 200 table database however, the resulting ERD was a black mess of overlapping lines and boxes – completely unreadable.

Modern CASE tools are probably better than the older ones for generating a readable diagram, but many have sidestepped the problem by just showing the relationships from a single table, making it harder to see the big picture.

Unfortunately many organizations have also chosen not to invest in CASE tools, so although using a modern CASE tool may be a feasible way of visualizing a database design, it is not always a viable path.

8.3 An ERD is essential for understanding a large database

An *ERD* is more or less a prerequisite for understanding a database design. Unfortunately, most projects either did not create one intitially or have not kept the ERD up to date with any changes that have been made to the database. So as a Business Analyst brought in to look at a project, the simplest thing to do is to generate our own ERD.

This is possible because all databases contain details about the tables and their relationships, otherwise the database software would not be able to work. The *DVD Rental* ERD was produced by running a set of queries against the database to identify the tables, columns and the relationships between the tables. Then using

the Graphviz [1] open source graph visualization software toolset, specifically the *dot* tool, an *SVG* image of the design was generated. This *SVG* image was then placed into a HTML file and made interactive usign some simple Javascript and the D3.js[2] javascript library.

The overall procedure is explained in detail in the appendix, with explanations of the SQL queries used to get the table and relationship information out of the database. Online there are current versions of the queries for PostgreSQL[3], Oracle[4] and SQLServer[5] databases.

These scripts have been used to generate diagrams for up to 500 table databases, and while viewing there ERD in a browser is not ideal, it is trivially easy to find a table or column name and from there follow the relationships to other tables. As long as we remember to highlight the tables and relationships we can visualize the complete path. On one project with over 2,000 tables and over 4000 relationships, generating the ERD took a significant time, but the diagram was usable in the browser. The only problem with the very large diagram was that some relationship lines turned out to be very long.

[1] http://graphviz.org/
[2] https://d3js.org/
[3] http://improvingwetware.com/files/PostgreSQLERDGeneration.sql
[4] http://improvingwetware.com/files/ErdCreation-specific.sql
[5] http://improvingwetware.com/files/ERDCreationSqlServer-2012-populated.sql

Chapter 9

Learning The Domain

Starting with an ERD gives you a good starting point for your *Database Archeology* exploration of the concepts used within the domain. In order to really learn about the domain, you need to look at the actual data in the database.

9.1 How large are the tables?

A good place to start is to find out how large are the various tables. To do this you need to use some of the various functions [1] that are available in PostgreSQL. The `count(*)` function is the simplest of these, it just returns the number of rows returned by that query. For this query rather than have the column named after the function name, the construct `as number_of_actors` is used to rename the column.

```
select count(*) as number_of_actors
from actor;
```

You can do this type of counting to discover the cardinality of relationships. Another relationship not shownon the ERD in Chapter 8 and not enforced by a *Foreign Key* between the `store` and `inventory` on `store_id`. You can use the `group by` option on the query to get the count of records per `store_id`, specificaly the inventory per store, and a count of the unique (`distinct`) films. You can also get the average number of copies of a film in each store. The `* 1.0` is needed because the `count` result is an integer, and when *PostgreSQL* divides two integers the result is also an integer. Table 9.2 has the results of this query.

```
select store.store_id,
    count(inventory.store_id) as inventory_count ,
    count(distinct inventory.film_id) as num_films,
    round((count(inventory.store_id) * 1.0) /
      count(distinct inventory.film_id), 3) as avg_copies
from store
```

[1] https://www.postgresql.org/docs/9.6/functions.html

Table 9.1: Number of actors

number_of_actors
200

Table 9.2: Inventory and films by store

store_id	inventory_count	num_films	avg_copies
1	2270	759	2.991
2	2311	762	3.033

```
join inventory on inventory.store_id = store.store_id
group by store.store_id
```

9.2 Walking the database

A lot of useful information can be pulled from a database by doing a simple query that answers the question, *What are the X related to Y.* For example, which actors have appeared in English language films?

To get the answer to this query, you neeed to start with the `language` table, then join that to the `film` table, from there to the `film_actor` and finally to the `actor` table. One thing to consider when doing this type of query is that you can get very large result sets, so when initially exploring, it is useful to limit the number of rows returned by use of the `limit` keyword. Table 9.3 has the results of this query.

```
select language.name,
  film.title,
  actor.first_name,
  actor.last_name
from language
join film on film.language_id = language.language_id
join film_actor on film_actor.film_id = film.film_id
join actor on actor.actor_id = film_actor.actor_id
where language.name = 'English'
limit 5;
```

9.3 Some joins are incorrect

Sometimes you will find that there is a match between some columns in different tables, but that does not always mean that a join between these tables is a sensible idea. The example below shows a join between the `film` and `address` tables, where through an accident of the identifiers used for each table matching. Table 9.4 has the results of this query.

Table 9.3: English Language Actors

name	title	first_name	last_name
English	Chamber Italian	Alec	Wayne
English	Chamber Italian	Henry	Berry
English	Chamber Italian	Rip	Winslet
English	Chamber Italian	Gina	Degeneres
English	Chamber Italian	Adam	Hopper

Table 9.4: An Incorrect Join film to address

film_id	title	address
1	Academy Dinosaur	47 MySakila Drive
2	Ace Goldfinger	28 MySQL Boulevard
3	Adaptation Holes	23 Workhaven Lane
4	Affair Prejudice	1411 Lillydale Drive
5	African Egg	1913 Hanoi Way

```
select
    film.film_id,
    film.title,
    address.address
from film
join address on address.address_id = film.film_id
order by film.film_id
limit 5;
```

You have to be careful when typing the SQL queries to make sure that you do not type an incorrect column name, otherwise you can get an unexpected result set. The problem arose because the `_id` columns are all of the same datatype, so if the numbers are in the same range, then you can accidentally get a match. An obvious fix for this is to use different number ranges for the identifier numbers, or to use a large enough identifier that the identifiers are guaranteed to be more or less unique.

9.4 Common Table Expressions

The idea behind a Common Table Expression[2] (or a `with` query) is to create a temporay named table that can be used in subsequent queries. The line `with film_inventory as` names the common table expression, so that it can be used later `select * from film_inventory`.

The query below shows the list of films that are in `inventory` at both stores in Table 9.5. The `film_inventory` query is a `left join` between the `film` and `inventory` tables, so a film will be in the result even if it is not in the `inventory` table. The names of the columns in the common table expression query are the names available

[2]https://www.postgresql.org/docs/9.6/queries-with.html

Table 9.5: Common Table Expression example

film_id	title	inventory_count	num_stores
1	Academy Dinosaur	8	2
4	Affair Prejudice	7	2
6	Agent Truman	6	2
7	Airplane Sierra	5	2
9	Alabama Devil	5	2

to the subsequent queries, so the name `num_stores` is available to use in the query rather than having to use the expression `count(distinct inventory.store_id)`.

Using a where clause of `where num_stores = 0` will find all films that are not in inventory at any store, and querying for `where num_stores = 1` will identify those films that are only available at one store.

```
with film_inventory as (
    select film.film_id,
        film.title,
        count(inventory.film_id) as inventory_count,
        count(distinct inventory.store_id) as num_stores
    from film
    left join inventory on inventory.film_id = film.film_id
    group by film.film_id, film.title
)
select * from film_inventory
where num_stores = 2
order by title
limit 5
```

These Common Table Expressions (CTE) are used extensively to make queries simpler and more readable, adn can be used for writing what would otherwise be a very complex query. A simple extension of the above query would be to add in the `category` of the film.

```
with film_inventory as (
    select film.film_id,
        film.title,
        count(inventory.film_id) as inventory_count,
        count(distinct inventory.store_id) as num_stores
    from film
    left join inventory on inventory.film_id = film.film_id
    group by film.film_id, film.title
)
select category.name,
    film_inventory.title,
    film_inventory.inventory_count
from film_inventory
join film_category on film_category.film_id = film_inventory.film_id
```

```
join category on category.category_id = film_category.category_id
order by name, title
limit 5
```

It is possible to use more than one Common Table Expression in a query, and it is possible for subsequent sub-queries to refer to previous CTE. In the example below, the `film_inventory` CTE is as before, and it is used by the `categories` CTE, adn then those two CTE are joined in the final query to product the list of categories, the number of films in those categories and the inventory of those films, with that list restricted to films that are stocked in both stores. Results are shown in Table 9.6.

```
with film_inventory as (
    select film.film_id,
        film.title,
        count(inventory.film_id) as inventory_count,
        count(distinct inventory.store_id) as num_stores
    from film
    left join inventory on inventory.film_id = film.film_id
    group by film.film_id, film.title
),
categories as (
    select category.name,
        film_inventory.film_id,
        film_inventory.title
    from film_inventory
    join film_category on film_category.film_id = film_inventory.film_id
    join category on category.category_id = film_category.category_id
)
select categories.name,
    count(categories.film_id) as films_in_category,
    sum(film_inventory.inventory_count) as inventory_count,
    round(avg(film_inventory.inventory_count),2) as average_inventory
from categories
join film_inventory on film_inventory.film_id = categories.film_id
    and film_inventory.num_stores = 2
group by categories.name
order by categories.name
limit 8
```

With an ERD and the ability to run queries against the database, it is possible to get a good idea of the overall domain. You do not need to know the full complexity of SQL to be able to explore a database, basically all that is needed is the ability to `join` tables, and `group` results to `count`, `sum` and `avg` (average) column values.

Table 9.6: Using two CTE in one query

name	films_in_category	inventory_count	average_inventory
Action	44	257	5.84
Animation	46	285	6.20
Children	31	185	5.97
Classics	33	209	6.33
Comedy	34	202	5.94
Documentary	34	208	6.12
Drama	37	224	6.05
Family	36	215	5.97

Table 9.7: Cardinality statistics

minimum	average	median	mode	maximum
0	4.58	5	6	8

9.5 Exploring Cardinality

These *aggregate* queries allow you to get a good handle on the cardinality of the relationships between the various business concepts. SQL supports `min` and `max` to allow you to discover the range of values. The syntax to calculate the `median` (middle) value and the `mode` (most common) value is a little awkward, but the benefit is that `PERCENTILE_CONT` allows you to calculate *quartiles* or any other fraction.

```
with film_inventory as (
    select film.film_id,
        film.title,
        count(inventory.film_id) as inventory_count,
        count(distinct inventory.store_id) as num_stores
    from film
    left join inventory on inventory.film_id = film.film_id
    group by film.film_id, film.title
)
select min(inventory_count) as minimum,
    round(avg(inventory_count),2) as average,
    PERCENTILE_CONT(0.50) WITHIN GROUP
        (ORDER BY inventory_count) as median,
    mode() WITHIN GROUP (ORDER BY inventory_count) as "mode",
    max(inventory_count) as maximum
from film_inventory
```

The results in Table 9.7 show that the maximum number of copies of a film in the database is 8, so if you were a customer you might have to wait for a long time to see a popular DVD.

Table 9.8: Full Text Query

film_id	title
1	Academy Dinosaur
231	Dinosaur Secretary

9.6 Full Text Indexes

There is one special index in the `DVDRental` database, the index on the `fulltext` column in the `film` table. This index is there to support *search engine style*, Full Text Search[3] across the title and description of the film.

A full text search in *PostgreSQL* is done using the `tsvector` and `tsquery` types. The `to_tsvector` converts the normal text of the title and description into a `tsvector`

- Original Text: Academy Dinosaur A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The Canadian Rockies
- `tsvector`: 'academi':1 'battl':15 'canadian':20 'dinosaur':2 'drama':5 'epic':4 'feminist':8 'mad':11 'must':14 'rocki':21 'scientist':12 'teacher':17

You would not normally look at the `tsvector` format, but it is an alphabetically sorted list of all of the words in the input text minus the stop words. The numbers next to the words corresponds to the position(s) of the word in the original stream. If `epic` occurred twice in the string, then it would show up as `'epic':4,22` in the `tsvector`.

Without the index on the `fulltext` column, you would need to write a query that did a full table scan to find the films that contain both `canadian` and `dinosaur` somewhere in the title or descripton. So the where clause needs to build the appropriate `tsvector` and then check that against the relevant words. See Table 9.8 for the results.

```
SELECT film_id, title FROM public.film
where to_tsvector(title || ' ' || description)  @@
  to_tsquery('canadian & dinosaur')
ORDER BY film_id ASC
```

With a *Full Text Index* on the `fulltext` column, the query is simpler to write, and will run faster since it can use the index and does not need to scan the entire table.

```
SELECT film_id, title FROM public.film
where fulltext @@ to_tsquery('canadian & dinosaur')
ORDER BY film_id ASC
```

> Note. The `film` table has a *database trigger* `film_fulltext_trigger` that fires on insert and update to populate the `fulltext` column. This trigger avoids the need for the developers to do anything to update the `fulltext` column.

[3] https://www.postgresql.org/docs/9.6/textsearch.html

From a database archeology viewpoint, *full text indexes* are often missing features in many existing applications. The applications are missing the functionality that allows a user to do a *search engine style* query against the data in the application. Many existing applications are missing *description* fields, since previously there were no easy way to search text fields, so the text fields were omitted from the applications.

Chapter 10

Database Views

As well as the tables, most databases also have *views* and *materialized views*, which are effectively just queries that have been named and stored in the database. With a materialized view the query is precomputed by the database and the results stored in the database so that queries against materialized views would normally run faster than queries against a normal view.

10.1 Views

In the DVD Rental database, there is a view called `sales_by_store` and it can be queried just like a table. When you include the view name in a query, the database has to run the underlying view query to get the results shown in table 10.1.

```
select sales_by_store.store,
  sales_by_store.manager,
  sales_by_store.total_sales
from sales_by_store;
```

The query that underlies the `sales_by_store` view is relatively complex and joins eight different tables to get the result. The extra text to define the view[1] is the line:

- `CREATE VIEW sales_by_store as`

All lines after that are just a normal SQL query that is using `||` to concatenate the `city` and `country` with a `,` separating the two values.

```
CREATE VIEW sales_by_store as
SELECT (c.city::text || ','::text) || cy.country::text AS store,
  (m.first_name::text || ' '::text) || m.last_name::text AS manager,
  sum(p.amount) AS total_sales
FROM payment p
JOIN rental r ON p.rental_id = r.rental_id
```

[1] https://www.postgresql.org/docs/9.6/sql-createview.html

Table 10.1: Sales by Store from View

store	manager	total_sales
Woodridge,Australia	Jon Stephens	30683.13
Lethbridge,Canada	Mike Hillyer	30628.91

```
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN store s ON i.store_id = s.store_id
JOIN address a ON s.address_id = a.address_id
JOIN city c ON a.city_id = c.city_id
JOIN country cy ON c.country_id = cy.country_id
JOIN staff m ON s.manager_staff_id = m.staff_id
GROUP BY cy.country, c.city, s.store_id, m.first_name, m.last_name
ORDER BY cy.country, c.city;
```

> Note. *The `::text` appended to the column names is to convert the column values to the text datatype, but it is not actually needed for this example.*

As a Business Analyst, you would not normally create a view in a database, but investigating the views that exist in the database gives a good idea of the complex queries that are often required by the application.

10.2 Materialized Views

The syntax for creating a *Materializsed View*[2] is more commplex since when you create a view you also have to specify information about where the data will be stored and when the view will be refreshed. Other than that though, you query a materialized view exactly the same as any other view or table.

The one gotcha that exists for Materialized Views is that the data in the view can be stale. This means that the data in one or mroe of the underlying tables has been updated, but the view has not yet been refreshed. The end result of this is that a query on a Materialized View may return old values for the data.

10.3 Listing the views

The query to get the list of views uses the INFORMATION_SCHEMA. Teh query below just pulls the name of the view and the tables involved in the view. If you want to see the query that underlies the view, include the column `views.view_definition` in your query.

[2] https://www.postgresql.org/docs/9.6/sql-creatematerializedview.html

Table 10.2: List of Views

view_name	tables
actor_info	actor, category, film, film_actor, film_category
customer_list	address, city, country, customer
film_list	actor, category, film, film_actor, film_category
nicer_but_slower_film_list	actor, category, film, film_actor, film_category

```
select views.table_name as view_name,
    string_agg(vtu.table_name, ', ') as tables
from information_schema.views
join information_schema.view_table_usage vtu
  on vtu.view_name = views.table_name
  and vtu.view_schema = views.table_schema
where views.table_schema = 'public'
group by views.table_name,
    views.view_definition
order by views.table_name,
    views.view_definition
limit 4
```

10.4 Understanding Views

As a business analyst, the value of the various database views is that they name common queries, and hence data access paths, that are used in the application. For all views you can look back at the ERD to see how the query underlying the view accesses the underlying tables.

Some applications will have a lot of views in the database, others will have very few. Often this is due to the use of toolsets that make it hard to write complex queries, so the queries are encoded as views in the database.

From a database archeology viewpoint, understanding how the various views are used will require conversations with the development team. You may find that like the `sales_by_store` view, that there are lots of views created for a single, specific purpose. That view gives the total sales by store over all time, but it cannot give weekly, monthly or quarterly sales figures.

Chapter 11

Business Analysis and Database Archeology

Once you have a good handle on the problem domain, you can start working on a deeper level of business analysis by looking deeper into the contents of the database. At this level of database archeology you are looking at the data to gain a deeper understanding of the business, from the data that has been recorded to date.

In a way this is using the ideas of *Big Data* to ask interesting questions of the existing data to discover information about the overall business.

> Note. The `DVDrental` database contains synthetic data that does not mimic a real world application, so the sample data returned by the queries will not be representative of what you would see in a real application.

11.1 Transaction Rates

It is possible to get insights about a business from the rate of transactions over time. Some businesses are very seasonal, or affected by promotions, in others the problem is more fine grained with the need to match resources to demand. An example is the number of active cashiers in a supermarket, when you have a lot of customers in the store, you need a lot of open checkouts, and at quiet periods you need less open.

Since the `rental` table has a `rental_date` column, you can use that to identify busy parts of the day. This is done by using either `date_part('hour', rental_date)` or `extract(hour from rental_date)`, since for some reason `PostgreSQL` has two equivalent methods to extract part of a date[1]. Using either of these methods, the query will have the hours of the day extracted as a number in the range `0..23`, and then you can count the number of rentals in that hour. The results shown in table 11.1 show the artificial nature of this dataset.

[1] https://www.postgresql.org/docs/9.6/functions-datetime.html

Table 11.1: Rentals by hour of day

hours	rentals
0	694
1	649
2	630
3	684

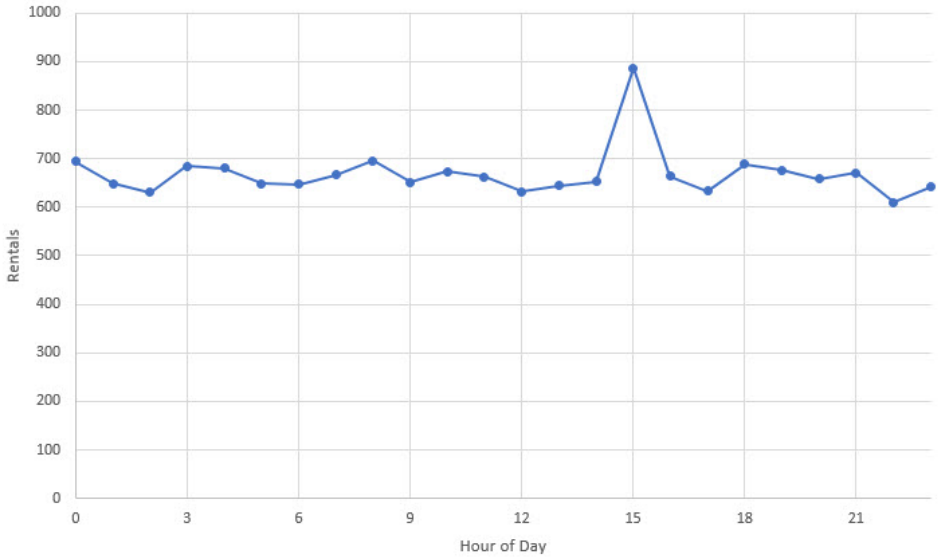

Figure 11.1: Transaction Rates

```
with rentals_by_hour as (
  select date_part('hour', rental_date) as hours,
    inventory_id
  from rental
)
select hours,
  count(inventory_id) as rentals
from rentals_by_hour
group by hours
order by hours
```

Importing the query results into *Excel* and then generating a graph shows that the rentals are effectively randomly distributed thoughout the complete day.

While this might be caused by merging the data from different stores which are in different time zones, rerunning the query for a single store gives essentially the same flat plot, just with smaller numbers. When looking at real transaction rate data you should see one or two clear peaks in the graph.

- If the peaks look clipped with roughly the same value continuing for a few hours, then the transaction rate is likely limited by some capacity constraint.
- Two peaks are a signal that you need to do some further investigation as to the nature of the business
- If there are no clear peaks then you might need to split the data out by day of week (`date_part('dow', rental_date)`) as well, to see if there is a pattern by day of week and hour. The query in the common table expression would then return hours in the range `0..167`. It may be that in you see a distinct pattern by day of week

```
  select date_part('hour', rental_date) +
      (date_part('dow', rental_date)* 24) as hour_of_week,
    inventory_id
  from rental
```

- Another cause of no clear peaks is having insufficient data over which to identify any trends or patterns within a day. If this is true then group your data by `day`, `week`, `month` or `quarter` to see which is appropriate to use.

Related to the transaction rates, you can do similar queries across the `payment` table to measure the income in a period

```
with payments_by_week as (
  select date_part('week', payment_date) as weeks, * from payment
)
select
    weeks,
    sum(amount) as payments
from payments_by_week
group by weeks
order by weeks
```

The analysis of these payments queries results is effectively the same as the transaction rates queries, just with different people interested in the results.

11.2 Investigating Durations

Looking at the `rental` table, there are two business events recorded, the `rental_date` and the `return_date`, so provided the `return_date` is not null, the duration of the rental can be calculated. Unlike other database systems, *PostgreSQL* provides a nicely formatted `interval` datatype to display the results in the format `d days hh:mm:ss` as in Table 11.2.

```
select customer_id, rental_date,
  (return_date - rental_date) as duration
 from rental
```

The `interval` datatype can be used with the grouping functions to find the `avg`, `min` and `max`, though when doing calculations on a set of `intervals`, you might need to

Table 11.2: Rental Durations

customer_id	rental_date	duration
459	2005-05-24 22:54:33	3 days 20:46:00
408	2005-05-24 23:03:39	7 days 23:09:00
333	2005-05-24 23:04:41	9 days 02:39:00
222	2005-05-24 23:05:21	8 days 05:28:00
549	2005-05-24 23:08:07	2 days 02:24:00

Table 11.3: Rental Duration Statistics by Customer

customer_id	average	minimum	maximum
1	4 days 11:00:00	1 day 01:57:00	9 days
2	5 days 12:00:00	19:13:00	9 days
3	5 days 21:00:00	19:33:00	8 days
4	3 days 22:00:00	23:38:00	8 days
5	5 days 01:00:00	20:10:00	9 days

apply the `justify_interval` function to force the interval to display correctly (this corrects an interval of `1 day 26:00:00` to show `2 days 02:00:00` – the interval still shows the correct duration, but it looks a bit strange.) The `date_trunc` function works the same way as the `date_part` function seen previously, but it truncates the date to remove all smaller units of time, as in Table 11.3.

```
select customer_id,
  date_trunc('hour',
    justify_interval(avg(return_date - rental_date))
  ) as average,
  min(return_date - rental_date) as minimum,
  date_trunc('day',max(return_date - rental_date)) as maximum
from rental
group by customer_id
order by customer_id
```

If you need to round the `interval` rather than truncate the `interval`, simply add half the unit used in `date_trunc` and the effect will be one of rounding. You will also need to use `justify_interval` before applying the `date_trunc` as shown in query below.

```
select customer_id,
  date_trunc('day',
   justify_interval(min(return_date-rental_date)+interval '12 hours')
  ) as minimum,
  date_trunc('day',max(return_date - rental_date)) as maximum
from rental
group by customer_id
order by customer_id
```

Note. *PostgreSQL* returns `null` when you try to create an interval when

Table 11.4: Rental Duration Statistics by Month

rental_month	average	minimum	maximum	rentals
2005-05-01	4 days 21:00:00	18:01:00	9 days	1156
2005-06-01	5 days 01:00:00	18:02:00	9 days	2311
2005-07-01	5 days	18:00:00	9 days	6709
2005-08-01	5 days 01:00:00	18:00:00	9 days	5686

either the start or end of the interval is `null`, so if the `rental` has not been completed and `return_date` is `null`, that record is skipped over by any grouping function like `min`.

From the Business Analysis viewpoint, being able to report on the duration of business events can give you a better understanding of the problem domain. Knowing how long a business activity takes is a key part of understanding business functions, especially in service organizations that need to know how long it takes from the original request until the customer is satisifed with the service received.

A simple adjustment to the query that produced Table 11.3, provides an average rental duration by month, Table 11.4.

```
select date_trunc('month', rental_date) as rental_month,
  date_trunc('hour',
    justify_interval(avg(return_date - rental_date))
  ) as average,
  min(return_date - rental_date) as minimum,
  date_trunc('day',max(return_date - rental_date)) as maximum,
  count(1) as rentals
from rental
group by rental_month
order by rental_month
```

In a service organization, it can be useful to split the duration of business activities into two different parts, the *queue time* and the *activity time*. For many organizations the queue time is longer than then activity time, so a project to optimize the activities may not have any impact on how long it takes for the customer to be satisfied.

The archeology task in many systems involves identifying the different business events, so the overall duration can be decomposed into the component steps. Many existing systems do not track all the low level steps involved in a workflow, so you cannot differentiate between time spent in queues and time spent actually working on a task.

11.3 Comparing Capacity and Demand

In the `DVDRental` sample, the *realized demand* for a title can be found by identifying overlapping rental periods. You can only get the realized demand because the

Table 11.5: Demand for a film

film_id	title	rental_date	demand	copies
951	Voyage Legally	2005-05-25 15:54:16	1	7
951	Voyage Legally	2005-05-26 06:14:06	2	7
951	Voyage Legally	2005-05-27 16:40:40	3	7
951	Voyage Legally	2005-06-15 05:55:40	1	7
951	Voyage Legally	2005-06-17 11:24:57	2	7

database does not record the customers who were not able to rent a title because it was not available in the store. In some domains this *latent demand* is an important concept as it is a signal to the organization, so a concert promoter may put on more dates if all the concerts sell out quickly.

To get the set of overlapping rentals, you first need to query to get a list of `film`s and the associated `rental_date`s, and then at the rental_date check how many rentals overlap that `rental_date`. You only need to check at the start of a rental, since the end time is when the demand reduces. Results are in Table 11.5.

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
       where tot.film_id = film.film_id) as copies
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Voyage Legally'
)
select film_id, title,
  rental_date,
  (select count(1) from rental
   join inventory on inventory.inventory_id = rental.inventory_id
   where inventory.film_id = rentals.film_id
    and rental.rental_date <= rentals.rental_date
    and (rental.return_date > rentals.rental_date
      or rental.return_date is null)
  ) demand,
  copies
from rentals
order by film_id, rental_date
```

> Note. The `rentals` CTE has clause `film.title = 'Voyage Legally'` to improve the speed of the query by restricting it to a single title. This is because every rental for a film is compared against all other rentals for that film, so this is an $O(n^2)$ problem, and when you query all titles, the query takes about a minute to execute.

Table 11.5 shows that the demand at each of the rental dates, together with the number of copies of that title at the rental date. In your real system you may need

Table 11.6: Stockouts

film_id	title	first_stockout	stockouts	copies
951	Voyage Legally	2005-08-02 11:42:23	2	7

to calculate the capacity at each rental date, if resources change over time or are unavailable for periodic maintenance.

As written, the query is only partially useful, a refinement is to just report the rental_dates when there are no more copies of the film available, and then count the number of times when the availability of the film drops to zero.

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
     where tot.film_id = film.film_id) as copies
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Voyage Legally'
),
out_of_stock as (
  select film_id, title,
    rental_date,
    case when copies <= (
      select count(1) from rental
      join inventory on inventory.inventory_id = rental.inventory_id
      where inventory.film_id = rentals.film_id
        and rental.rental_date <= rentals.rental_date
        and (rental.return_date > rentals.rental_date
          or rental.return_date is null)
    ) then true end as no_copies,
    copies
  from rentals
)
select film_id, title, min(rental_date) as first_stockout,
  count(1) as stockouts, copies
from out_of_stock
where out_of_stock.no_copies = true
group by film_id, title, copies
order by stockouts desc, film_id, title, copies
```

Ordering the results of this query by the count of stockouts, identifies the titles which went out of stock the most. This query reveals the capability to determine the titles that need more copies to prevent future stockouts.

You will find that on real databases, this type of query may take minutes or hours to run. This is acceptable as part of database archeology, because you are interested in identifying business concepts and capabilities. If it turns out that this capability

is needed in the application, the development team can build in features to enable this type of capacity vs. demand monitoring.

> Note. In the section on *Transaction Rates* you will need to divide the results in table 11.1 by the number of days covered by your query if you want the actual transaction rate for that hour of the day. Sometimes all you will need is the relative rates at different times to get the information you need, othertimes you might need to calculate the actual rates. A simple query like the one shown below will give you the number of days in the dataset, and the number of rentals on each of the days.

```
select date_trunc('day', rental_date) as days, count(1)
from rental
group by days
order by days
```

Chapter 12

Learning To Write SQL Queries

The last query in the previous chapter that generated Table 11.5 needed two *CTE* to generate the final result. Although when following through these ideas in the book you can just retype the query from start to finish, when building your own queries, it is much easier to build up the queries in a series of steps, executing the query as you go to ensure that it makes sense

12.1 Plan Your Query

Your first step in planning a query is to look at your ERD to find an appropriate path through the tables. Since the query is trying to detect stockouts, one place to start is with the `rental` table, giving this possible plan for the query:

- `rental` access via `rental_date` to get `inventory_id`
- Join to `inventory` on `inventory_id`
- Join to `film` to get `film_id` and `title`
- From `film` go back to `inventory` to count copies
- On each `rental_date` associated with a `film`, count the active rentals there are (rented before this date and not yet returned at this date)
- Compare the count of copies to the count of active rentals and record a stockout for whenever the availability goes to zero
- For each `film`, count the number of stockouts and get the date of the first stockout

  Note. Initially you might need to write down a detailed plan for the overall query. Later on as you become more familiar with the database you might be able to get away with just minimal notes on yor planned path through the tables.

12.2 Start by Querying a Single Table

With every query, it is easiest to start querying a single table, that way you know the dataset you are getting back from your query. If it might be a large dataset, then you might want to put a `limit` on the size of the result set you get back, but with only 16k rows, the `rental` table is small enough that this is not an issue.

```
select rental.inventory_id, rental.rental_date
from rental
where rental_date between '2005-01-01' and '2005-12-31'
```

For a real query you would likely want to limit the resulting dataset to a specific date range as shown above, usign whatever date format is set for your system (`yyyy-mm-dd` in the above case).

12.3 Add Joins One at a Time

Although it is tempting to wrtie out the complete set of joins in one go, doing so may delay you if you make a mistake in the join condition, especially if the mistake results in more rows than you expect.

```
select inventory.inventory_id, rental.rental_date
from rental
join inventory on inventory.inventory_id = rental.inventory_id
```

You should get exactly the same number of rows for this query as you had for the first query. To confirm that the result datasets are identical, you can combine[1] the two queries using `except` to find the results that are in the first query and not in the second query.

```
select inventory.inventory_id, rental.rental_date
from rental
join inventory on inventory.inventory_id = rental.inventory_id
except
select rental.inventory_id, rental.rental_date
from rental
```

> Note. *PostgreSQL* uses `union`, `intersect` and `except` for combining queries, *Oracle* uses `minus` instead of `except`.

Once you have verified that the `join` to the `inventory` table is correct, then you can add in the `join` to the `film` table.

```
select film.film_id, film.title, rental.rental_date
from rental
join inventory on inventory.inventory_id = rental.inventory_id
join film on film.film_id = inventory.film_id
order by film_id, rental_date
```

[1] https://www.postgresql.org/docs/9.6/queries-union.html

Table 12.1: Rentals by Title and Date

film_id	title	rental_date
1	Academy Dinosaur	2005-05-27 07:03:28
1	Academy Dinosaur	2005-05-30 20:21:07
1	Academy Dinosaur	2005-06-15 02:57:51
1	Academy Dinosaur	2005-06-17 20:24:00
1	Academy Dinosaur	2005-06-21 00:30:26

Table 12.2: Number of Copies

count
8

The result set in Table 12.1 is ordered by `film_id` and `rental_date` to make it easier to understand the dataset.

12.4 Add Subquery After Main Query is Correct

Using the results in Table 12.1, you can write the subquery to find the total number of copies as a separate query. Initially you can *hardcode* the `film_id` to make sure that the query is correct.

```
select count(1) from inventory tot
where tot.film_id = 1
```

Table 12.2 shows how many copies of *Academy Dinosaur* exist in the database. You can now add the subquery to the first query to get the copies at the rental date, replacing the hardcoded `1` with `film.film_id` so that the subquery executes for each row in the dataset.

```
select film.film_id, film.title, rental.rental_date,
  (select count(1) from inventory tot
  where tot.film_id = film.film_id) as copies
from rental
join inventory on inventory.inventory_id = rental.inventory_id
join film on film.film_id = inventory.film_id
```

This type of subquery, because it runs for each row in the dataset, takes much longer than the query without the subquery. Ideally you would avoid writing this type of subquery, but it cannot be avoided unless you know for definite that the availability information that you are quering for does not vary by date

For *DVDRental*, because the number of copies does not change over time, you could try to get the number of copies using the following query that has another `join` back to the `inventory` table and then uses `group by` to get the count.

Table 12.3: Simultaneous Rentals

film_id	rental_date	simult
462	2006-02-14 15:16:03	2
789	2006-02-14 15:16:03	2
189	2006-02-14 15:16:03	2
301	2006-02-14 15:16:03	2
892	2006-02-14 15:16:03	2

```
select film.film_id, film.title, rental.rental_date,
count(tot.inventory_id) as copies
from rental
join inventory on inventory.inventory_id = rental.inventory_id
join film on film.film_id = inventory.film_id
join inventory tot on tot.film_id = film.film_id
group by film.film_id, film.title, rental.rental_date
```

Interestingly this gives a few less rows than the original subquery version, as you can confirm by using `except` to identify the differences between the two queries. The problem is that there are some records in the `rental` table where the same `film` is rented twice at the same time (but under a different `inventory_id`).

```
select film_id, rental.rental_date, count(1) as simult
from rental
join inventory on inventory.inventory_id = rental.inventory_id
group by film_id, rental_date
having count(1) > 1
```

These simultaneous rentals shown in Table 12.3 cause the `group by` version of the query to double count the copies of the film (since the count is done once for each distinct `inventory_id` at the `rental_date`). The above query uses `having`[2] in place of the usual `where` clause to only return the rows where there are simultaneous rentals.

Assuming your resource availabilty does not vary by time, write another, separate CTE query that just gets the number of copies of each `film`. Since this does not count over the `rental` records, it correctly counts the number of copies.

```
select film.film_id, film.title,
count(tot.inventory_id) as copies
from film
join inventory tot on tot.film_id = film.film_id
group by film.film_id, film.title
order by film.film_id
```

Taking this approach would make the overall query run faster, but with the added complication of now needing three CTE before the main query.

[2] https://www.postgresql.org/docs/9.6/tutorial-agg.html

Table 12.4: Rentals CTE Query Results

film_id	title	rental_date	copies
1	Academy Dinosaur	2005-07-08 19:03:15	8
1	Academy Dinosaur	2005-08-02 20:13:10	8
1	Academy Dinosaur	2005-08-21 21:27:43	8
1	Academy Dinosaur	2005-05-30 20:21:07	8
1	Academy Dinosaur	2005-06-17 20:24:00	8

12.5 Initially Limit the Size of CTE Results

The decision to use a CTE depends on whether you can figure out a reasonable and readable way to get your desired results from a single query. For the stockout query, it was simpler to have a CTE that just had the `film`, `rental_date` and number of copies.

To make the next stage of the query writing easier, you can limit the size of the result set so that this first part of the query runs faster. In the DVDRental example, this can be done by limiting the result set to a single `film.title`, e.g. *Academy Dinosaur* as in Table 12.4.

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
     where tot.film_id = film.film_id) as copies
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Academy Dinosaur'
)
select film_id, title,
  rental_date, copies
from rentals
```

Using the results from the `rentals` CTE, you can now write the subquery to find the number of concurrent rentals at the `rental_date` for the relevant `film_id`. As before, when developing the subquery it is useful to hardcode relevant values from the `rentals` CTE results in Table 12.4.

```
select count(1) from rental
join inventory on inventory.inventory_id = rental.inventory_id
where inventory.film_id = 1
and rental.rental_date <= '2005-05-30 20:21:07'
and (rental.return_date > '2005-05-30 20:21:07'
  or rental.return_date is null)
```

> Note. This subquery has to take into account `null return_date` values, that mean the *DVD* has not been returned yet.

The next step is to add this subquery into the previous query, removing the hard-

Table 12.5: Concurrent Rentals at 2005-05-30 20:21:07

count
2

Table 12.6: Rentals with count of concurrent rentals

film_id	title	rental_date	rented	copies
1	Academy Dinosaur	2005-05-27 07:03:28	1	8
1	Academy Dinosaur	2005-05-30 20:21:07	2	8
1	Academy Dinosaur	2005-06-15 02:57:51	1	8
1	Academy Dinosaur	2005-06-17 20:24:00	2	8
1	Academy Dinosaur	2005-06-21 00:30:26	2	8

coding. The results are ordered by `rental_date` to make it easier to look up the result for `2005-05-30 20:21:07` to ensure that the query still gives the same result for that date.

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
     where tot.film_id = film.film_id) as copies
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Academy Dinosaur'
)
select film_id, title,
  rental_date,
  (select count(1) from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  where inventory.film_id = rentals.film_id
    and rental.rental_date <= rentals.rental_date
    and (rental.return_date > rentals.rental_date
      or rental.return_date is null)) as rented,
  copies
from rentals
order by rental_date
```

The last part of this second query is to use `case`[3] `when` to compare the number of copies `rented` with the total number of `copies`. This can be done by `case when copies <= rented then true end`, although in the query you cannot use `rented` and instead have to put in the entire subquery.

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
     where tot.film_id = film.film_id) as copies
```

[3] https://www.postgresql.org/docs/9.6/functions-conditional.html

Table 12.7: Rentals with no copies

film_id	title	rental_date	no_copies	copies
1	Academy Dinosaur	2005-05-27 07:03:28	null	8
1	Academy Dinosaur	2005-05-30 20:21:07	null	8
1	Academy Dinosaur	2005-06-15 02:57:51	null	8
1	Academy Dinosaur	2005-06-17 20:24:00	null	8
1	Academy Dinosaur	2005-06-21 00:30:26	null	8

```
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Academy Dinosaur'
),
out_of_stock as (
select film_id, title,
  rental_date,
  case when copies <= (select count(1) from rental
    join inventory on inventory.inventory_id = rental.inventory_id
    where inventory.film_id = rentals.film_id
      and rental.rental_date <= rentals.rental_date
      and (rental.return_date > rentals.rental_date
        or rental.return_date is null))
  then true end as no_copies,
  copies
from rentals
)
select film_id, title, rental_date,
  no_copies, copies
order by rental_date
```

12.6 Group and Count in Final Query

The lasy part of the query is to `count` the number of times there were `no_copies` available. *SQL* makes this simple because when you use the expression `count(no_copies)` it only counts the rows when the value of `no_copies` in a row is not `null`[4]. This property also allows the query to count the number of rentals by using `count(1)` to record the number of rows that meet the grouping criteria.

In combination these two counts give a sense of the relevance of any stockout.

- No Demand - few stockouts, few rentals
- Meeting Demand - few stockouts, lots of rentals
- Capacity Issues - many stockouts, lots of rentals

[4]https://www.postgresql.org/docs/9.6/functions-aggregate.html

Table 12.8: Rentals with stockouts and rentals

film_id	title	first_stockout	stockouts	rentals	copies
1	Academy Dinosaur	2005-05-27 07:03:28	0	23	8

- Possible problem - many stockouts, few rentals

```
with rentals as (
  select film.film_id, film.title, rental.rental_date,
    (select count(1) from inventory tot
     where tot.film_id = film.film_id) as copies
  from rental
  join inventory on inventory.inventory_id = rental.inventory_id
  join film on film.film_id = inventory.film_id
    and film.title = 'Academy Dinosaur'
),
out_of_stock as (
select film_id, title,
  rental_date,
  case when copies <= (select count(1) from rental
    join inventory on inventory.inventory_id = rental.inventory_id
    where inventory.film_id = rentals.film_id
      and rental.rental_date <= rentals.rental_date
      and (rental.return_date > rentals.rental_date
        or rental.return_date is null))
  then true end as no_copies,
  copies
from rentals
)
select film_id, title, min(rental_date) as first_stockout,
  count(no_copies) as stockouts,
  count(1) as rentals, copies
from out_of_stock
group by film_id, title, copies
order by film_id
```

> Note. The results in Table 12.8 indicate that *Academy Dinosaur* was not a good sample for the first CTE, as that title never had a stockout.

As you work though your query plan, you are likely to find that you do not follow it exactly. This is normal when doing *Database Archeology* because you are interested in finding out what you can discover about the data to help inform your *Business Analysis* work.

Chapter 13

Modifying Data

Although strictly not part of business analysis, sometimes as part of *database archeology* it can be useful to make some minor changes to the data in a test database to see how the application responds. *SQL* makes it easy to `update` column values in a row, `insert` new rows and `delete` existing rows.

13.1 Turn Off Autocommit

Databases have the concept of a transaction. The idea is that you can make a set of related changes to the database, and then decide whether to `commit` the changes, or `rollback` the changes. After youi have made the changes, any `seelct` statement you write will see the changes, but your changes will not be made available to any other users of the database until you `commit` your changes. This is great because it allows you to check the results of your changes before making them permanent.

Unfortunately, to make it easy for users, many tools like `pgAdmin` by default enable *Auto Commit* whereby after each statenment is executed against the database, it automatically commits the change. This means that you cannot use `rollback` to undo your changes, so you should turn off this option. In `pgAdmin` the option to turn off *Auto Commit* is next to the icon used to run a query.

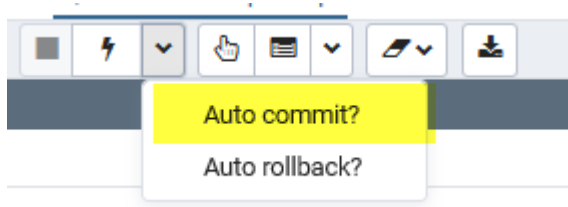

Figure 13.1: pgAdmin Auto Commit Option

Table 13.1: Original Value

film_id	title
1	Academy Dinosaur

Table 13.2: Modified Title

film_id	title
1	Hard Times

Once you have turned off *Auto Commit*, you will need to issue the command `commit* to save your changes, orrollback‘` to undo your changes.

13.2 SQL Works on Sets of Data

Before attempting any change to the data in a database, you need to understand that *SQL* operates on sets of data. Just as a `select` returns a dataset based on the `where` clause and any `join` conditions, an `update` or `delete` will affect every row that matches the `where` condition.

The safest option when developing an `update` or `delete` statement is to first create a `select` statement with the same `where` clause that you plan to use in the `update` or `delete` statment. That way you can see exactly the rows that will be affected by the statements.

```
select film_id, title
from film
where film_id = 1
```

When you execute this `select` statement, you should only see one row in the result dataset in table 13.1, showing the `film_id` of *1* and a `title` of *Academy Dinosaur*.

```
update film
set title = 'Hard Times'
where film_id = 1
```

The update will show a message stating how many rows were updated and how long it took to run the updates.

```
UPDATE 1

Query returned successfully in 108 msec.
```

Now when you re-run the original query, it should see the new `title` to the `film`, the old value of *Academy Dinosaur* has been replaced as shown in in table 13.2.

```
select film_id, title
from film
where film_id = 1
```

Table 13.3: Modified Title

film_id	title
1	Academy Dinosaur

> Note. Now that you have used `update` against that row in the `film` table, that new value is only visible by your current connection to the database. All other users are blocked from making any further changes to that row of the database until you either `rollback` or `commit` your change.

To demonstrate the reversible nature of database updates prior to issuing a `commit`, next issue a `rollback` command to the database.

```
rollback
```

Now, when you re-run the `select` statement will show the original *Academy Dinosaur* value again as shown in in table 13.3.

```
select film_id, title
from film
where film_id = 1
```

Alternatively, issuing a `commit` command would make your changes permanent.

> Warning. If you had omitted the line `where film_id = 1`, then **every record** in the `film` table would now have a `title` of *Hard Times*. The only warning you would have would be the message after the update that would state `UPDATE 1000`, so you should always check that the number of rows updated by your statement matches the number of rows you expected to update. If the number is different, always do a `rollback` to revert the data to the previous state.

13.3 Updating Existing data

The simple `update` statements in the previous section just updated a single column on a single row. *SQL* lets you update multiple columns in a table at the same time, all you have to do is list the column name and new value separated by a comma as per the example below updating the `title` and `description` columns.

```
Update film
set title = 'Hard Times',
  description = 'Chaplin Film'
where film_id = 1
```

Single table, but triggers (out of scope for this book) allow the update to be propagated to other tables.

calculation on existing value

Query to set value,

13.4 Inserting new data

13.5 Deleting rows from the database

After you have used SQL to explore a database, to understand an application, there is still more SQL to learn.

Chapter 14

Metadata in Databases

For systems that need to be configurable in the data they store about the key entities, the names of the tables in the database are often not reflective of the business concepts, as in this stylized patient record database.

Figure 14.1 has a `Patient` table in the *ERD*, but the rest of the table names do not reflect the *key business concepts*. The reason for this is that in hospital patient data systems, many different types of readings need to be recorded about the patient, but these readings will be different depending on the type of hospital and between different units. So these types of systems tend to be locally customized to add the appropriate readings that are needed by a particular unit and associated specialists.

> Note. This Example Patient Data Model uses `id` as the name of the primary key field in a table, and `tablename_id` as the name of the foreign key field in other tables. So where the *DVDRental* database used `film_id` in every table, in this database the `patient` table has an `id` field and the other tables refer to it using `patient_id` as the column name.

Another example of this type of systems are SCADA systems, used to display data from multiple sensors around an industrial plant and then send controls signals back. These systems have to be locally customized to match the exact equipment and monitors in the plant.

14.1 Trading Complexity for Extensibility

For these kind of systems, it is not effective to change the database tables every time a new parameter needs to be recorded, so instead the different parameters are stored as a row in a table. In the example patient data model, the `Attributes` table, which are grouped by `Categories` and each has an associated `Units` record.

> So if you need to define a *Manual Heart Rate* to record a patient reading that the staff have measured manually, then a new row needs to

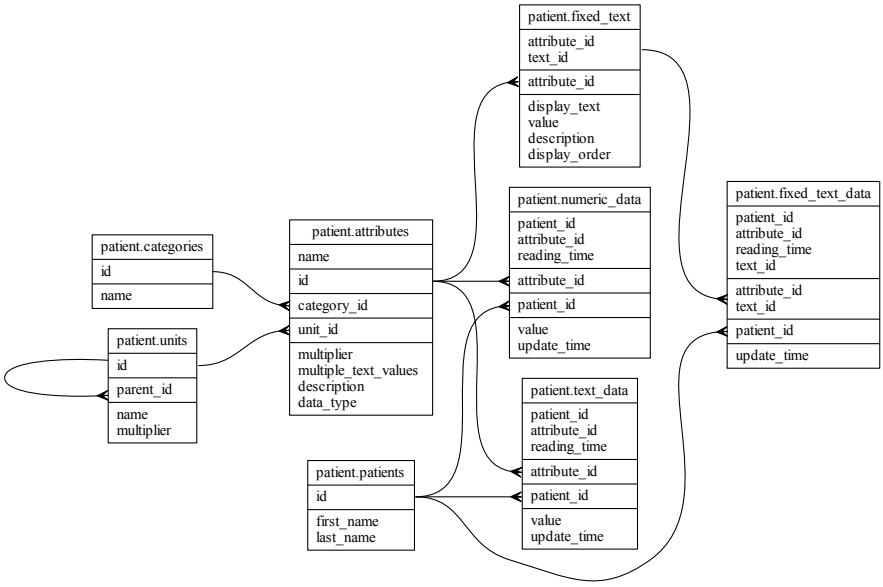

Figure 14.1: Example Patient Data Model

be inserted into the `Attributes` table with a name of `Manual Heart Rate` and an `data_type` of `Numeric`. This has to be associated with a `Units` record of `Heart Rate BPM` and then data can be entered into the `Numeric_Data` table for the appropriate new `attribute_id` and `patient_id` for the `reading_time` when the reading was taken.

This is more complex than just entering a value into a row in an existing table, but it is much simpler for the development team when there are multiple different heart rate monitors that can potentially be hooked up to a patient. When defining a new `Attribute` all that has to be decided is the `data_type`, `Numeric`, `Text` or `FixedText`, and then a name and unit assigned to the new `Attribute`.

From a development standpoint, there is extra complexity in that the developer needs to know the `data_type` of the `Attribute` in order to know which of the `Data` tables to query for the data. From a user customization viewpoint however, it is much simpler. All that is needed is to define a new `Attribute` and then place that attribute on a datasheet or data entry form and the customization is complete. The code simply needs to look up the `data_type` of the attribute in order to choose which *data* table to use.

Table 14.1: Attributes by Category

category	attribute	unit
Blood Pressure	Diastolic Blood Pressure	mmHg
Blood Pressure	Systolic Blood Pressure	mmHg

14.2 Database Archeology and Metadata

When confronted with this type of system, a simple measure of how complex the *database archeology* task will be to use `select count(1) from attributes`. A database with fewer than 100 `attributes` is easy to understand and deal with, but when you find a database with over 10,000 rows in `attributes`, then you have to hope that the `Attributes` are grouped into understandable `Categories`, as shown in Table 14.1.

```
select cat.name as category,
    attr.name as attribute,
    units.name as unit
from categories cat
join attributes attr on attr.category_id = cat.id
join units on units.id = attr.unit_id
order by category, attribute
```

This query also pulls in the `Units` as that often helps an outsider get a handle on what the name of the attribute means, it also puts it in context for conversations with the users.

What makes the business analysis task different is that, in these types of systems, the relationships between the different attributes are partially obscured. In a third normal form database, just looking at the column names in a table lets you know the important details about that business concept. In this case, you have to trust that the grouping of the `Attributes` into `Categories` makes logical sense without deep domain knowledge.

Sometimes the information as to which `Attributes` are displayed on a form or datasheet is only contained in the files that define the form or datasheet, but often that information is available in the database. The simplest form of this will have a `Forms` table with a relationship to a `Controls` table that has a relationship to the `Attributes` table. With this setup the positioning and layout of an attribute is handled by the `Controls` row, with the various `Controls` rows being organized by a row in the `Forms` table.

Discovering the key business concepts then resolves down to identifying the *attributes* that are collected on each data entry *form*, since logically users would expect to be entering data associated with business concept on one form. Relationships between the different concepts are harder to discover since the only real association present is that of the readings with the patient.

14.3 Business Analysis with Metadata

With metadata in the database, the business analysis task shifts from looking at the structure of the database, to looking at the configuration data in the database. In the context of the *Example Patient Data Model* this means focusing on the contents of the `Attributes`, `Categories` and `Units` tables.

Useful questions that can be asked include:

- Are all the defined `Attributes` used (count rows in the appropriate `data` table to discover this)
- Are all the `Attributes` uniquely named (sort list by attribute name and then category name)
- Do related `Attributes` have the appropriate `Units`
- Do the `Categories` and associated `Attributes` make sense to the domain users
- Is there any overlap between `Categories` that could cause confusion
- Are there subsets of `Attributes` that could be usefully and safely extracted into a different `Categories`

When dealing with metadata in the context of migrating from a legacy system, there are three main cases to consider

1. Legacy system and new system both use metadata, so the business analysis task lies mapping from the `Categories` and `Attributes` in the old system to the `Categories` and `Attributes` in the new system.
2. Legacy system used a relational database model but the new system uses metadata. Here the business analysis task lies in mapping from the old relational columns to an appropriate set of `Categories`, `Attributes` and `Units`.
3. Legacy system used metadata, but the new system uses a relational database model. Here the business analysis challenge lies in creating a mapping from the `Categories` and `Attributes` into the appropriate columns in the tables in the new system. This can be more of a challenge because you might need to map multiple attributes from the legacy system into a single column in the new database (e.g. the old system might have `Admission Weight`, `Daily Weight` and `Discharge Weight`, while the new database has a single column for `Patient Weight`).

Metadata in databases is relatively rare outside of specific domains, so you need to be extra careful when you encounter your first example. These systems are often customized by the end users, so there can be a lot of duplication and overlap in naming of attributes and grouping into categories.

Chapter 15

Object Models in Databases

One step up in complexity beyond the metadata approach is to define everything in the database, so that anything can be added to the application without needing to change the structure of the database. This takes the example of the *Metadata in Databases* and extends it to make everything, not just the `Attributes` configurable via rows in various tables.

What follows is a complete reimplementation of the *DVDRental* database, but using a configurable database design.

For systems that need to be configurable in the data they store about the key entities, the names of the tables in the database are often not reflective of the business concepts, as in this stylized patient record database.

Figure 15.1 has a `Patient` table in the *ERD*, but the rest of the table names do not reflect the *key business concepts*.

> Note. Object Models in Databases are a way of avoiding having to change the schema of the database. An alternative strategy is to use a microservices architecture where each microservice has a small database schema that matches the purpose of the microservice. This avoids the need to change the database schema, since microservices are very narrow and hence do not often need to evolve the schema.

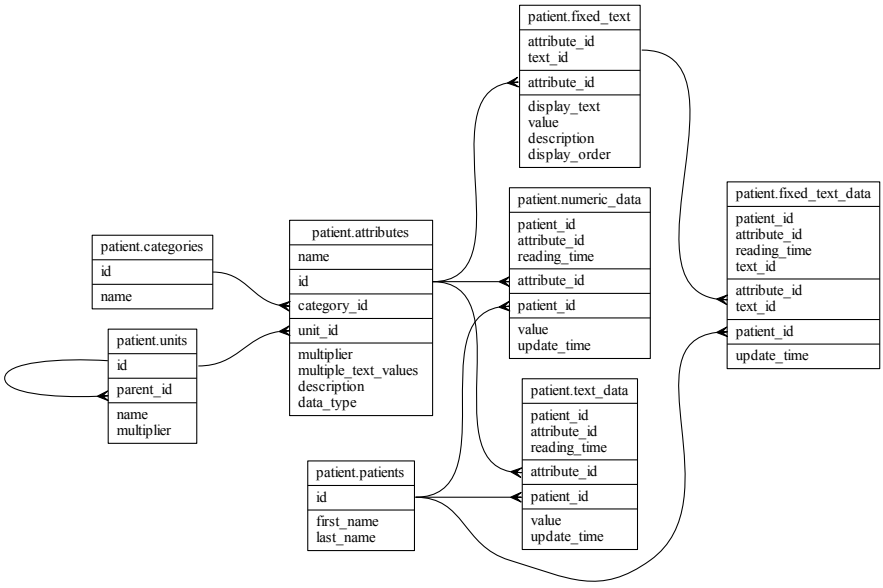

Figure 15.1: Example Patient Data Model

Chapter 16

Hierarchical Data

In some databases, there is a hierarchical relationship between rows in tables. The most common types seen are that of manufacturing systems, where there is the relationship between components, parts and sub assemblies, and in a personnel systems with the relatinship between employees and managers as shown below.

For simple cases where the number of levels of the hierarchy is known, then this case can be simply dealt with by a join back to the original table looking for the relevant paretn record (or child records if your query needs to look down the hierarchy). Unfortunately for the general case where the depth of the hierarchy is not known, then you need to use a *CTE* with an extra option.

16.1 Parts and Sub-Parts

Instead of Oracle's `start with … connect by` syntax, PostgreSQL uses

https://www.postgresql.org/docs/9.6/queries-with.html

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
  SELECT sub_part, part, quantity
  FROM parts
  WHERE part = 'our_product'
  UNION ALL
  SELECT p.sub_part, p.part, p.quantity
  FROM included_parts pr, parts p
  WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

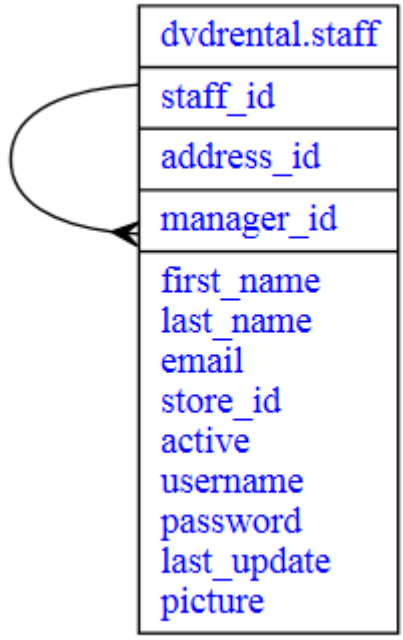

Figure 16.1: Staff Manager Relationship