Table of Contents

1
Database Access

Introduction

In Chapter 6, *Working with Data*, we looked at working with data that is stored incdisk files: we looked at plain text files and also datasets that take the form of R datafiles, CSV, HDF5 etc.

In this chapter, we will consider data stored in databases but will exclude the "big data" data repository as networking and remote working is the subject of the chapter 11.

There are a number of databases of various hues, and as it will not be possible to deal with all that Julia currently embraces, we will to pick one specific example for each category as they arise.

It is not the intention of this chapter to delve into the working of the various databases or to set up any complex queries, nor to go in depth into any of the available packages in Julia. All we can achieve in the space is to look at the connection and interfacing techniques of a few common examples.

A basic look at databases

In this section, we are going to cover a little bit of groundwork to work with databases. It's a large topic, and many texts are devoted to specific databases and also to accessing databases in general.

It is assumed that the reader has knowledge of these, but here we will briefly introduce some of the concepts that are necessary for the topics in the remaining sections.

The RED pill or the BLUE pill?

A few years ago discussing databases was simple. One talked of relational databases such as Oracle, MySQL, and SQL Server and that was pretty much it. There were some exceptions such as databases involved in Lightweight Directory Access Protocol (LDAP) and some configuration databases based on XML, when the whole world was relational.

Even products that were not relational, such as Microsoft Access, tried to present a relational face to the user.

We usually use the term Relational and SQL database interchangeably, although the latter more correctly refers to the method of querying the database than to its architecture.

What changed was the explosion of large databases from sources such as Google, Facebook, and Twitter, which could not be accommodated by scaling up the existing SQL databases of the time, and indeed such databases still cannot. Google produced a "white" paper proposing its BigTable solution to deal with large data flows, and the classification of NoSQL databases was born.

Basically, relational databases consist of a set of tables, normally linked with a main (termed primary) key and related to each other by a set of common fields via a schema. A schema is a kind of blueprint that defines what the tables consist of and how they are to be joined together. A common analogy is a set of sheets in an Excel file, readers familiar with Excel will recognize this parallel example.

NoSQL data stores are often deemed to be schema-less. This means it is possible to add different datasets with some fields not being present and other ones occurring. The latter is a particular difficulty with relational databases, as the schema has to be amended and the database rebuilt. Data stores conventionally comprising of text fall into this category, as different instances of text frequency throw up varying metadata and textual structures.

There are, of course, some databases that are termed NoSQL, but nevertheless have quite a rigorous schema and SQL-like (if not actual conventional SQL) query languages. These usually fall into the classification of columnar databases and I will discuss these briefly later in the chapter.

Possibly the biggest difference between SQL and NoSQL databases is the way they scale to support increasing demand. To support more concurrent usage, SQL databases scale vertically whereas NoSQL ones can be scaled horizontally, that is, distributed over several machines, which in the era of Big Data is responsible for their new-found popularity.

Interfacing to databases

Database access is usually via a separate task called a database management system (DBMS) that manages simultaneous requests to the underlying data. Querying data records presents no problems, but adding, modifying, and deleting records impose "locking" restrictions on the operations.

There is a class of simple databases that are termed "file-based" and aimed at a single user. A well-known example of this is SQLite and we will discuss the Julia support package later.

With SQL databases one other function of the DBMS is to impose consistency and ensure that updates are transactional safe. This is known by the acronym ACID (Atomicity, Consistency, Isolation and Durability). This means that the same query will produce the same results and that a transaction such as transferring money will not result in the funds disappearing and not reaching the intended recipient.

Although it may seem that all databases should operate in this manner, the results returned by querying search engines, for example, do not always need to produce the same result set. So we often hear that NoSQL data stores are governed by
the CAP rule, that is Consistency, Availability, and Partition Tolerance. We are speaking of data stores spread over a number of physical servers, and the rule is that to achieve consistency we need to sacrifice 24/7 availability or partition tolerance; this is sometimes called the two out of three rule.
In considering how we might interface with a particular database system we can (roughly) identify four mechanisms :-

1. A DBMS will be bundled as a set of query and maintenance utilities that communicate with the running database via a shared library. This is exposed to the user as a set of routines in an application program interface (API). The utilities will often be written in C and Java, but scripts can also be written in Python, Perl, and of course Julia. In fact Julia with its zero-overhead ccall() routine is ideal for this form of operation, and we will term the packages that use this as their principal mode of interface as wrapper packages. Often there is a virtual one-to-one correspondence between the Julia package routines and those in the underlying API. The main task of the package is to mimic the data structures that the API uses to accept requests and return results.
2. A second common method of accessing a database is via an intermediate abstract layer, which itself communicates with the database API via a driver that is specific for each individual database. If a driver is available, then the coding is the same regardless of the database that is being accessed. The first such system was Open Database Connectivity (ODBC) developed by Microsoft in the 1990s. This was an early package and remains one of the principal means of working

with databases in Julia. There are couple of other intermediate layers: Java Database Connectivity (JDBC) and Database Interface (DBI). The former naturally arises from Java and necessitates a compatible JNDI driver, whereas the latter was introduced in Perl and uses DBD drivers that have to be written for each individual database and in each separate programming language. All three mechanisms are available in Julia, although ODBC is the most common.

3. When we considered graphics in the previously, one mode of operation was to abrogate the responsibility of producing the plots to Python via matplotlib using the JavaPlot package. The same approach can be utilized for any case where there is a Python module for a specific database system, of which there are many. Routines in the Python module are called using the PyCall package, which will handle the interchange of datatypes between Julia and Python.
4. The mode access is by sending messages to the database, to be interpreted and acted on by the DBMS. This is typified by usage over the Internet using HTTP requests, typically GET or POST. The most common form of messaging protocols are called Representational State Transfer (RESTful), although in practice it is possible to use similar protocols, such as SOAP or XMLRPC. Certain database APIs may expose a shared library and also a REST-type API. The REST one is clearly an advantage when accessing remote servers, and the ubiquitous provision of HTTP support in all operating systems makes this attractive, especially in situations where firewall restrictions are in place.

Other considerations

In the remainder of this chapter, I'll cover all the preceding means of access to database systems looking at, in most cases, a specific example using a Julia package. There is one class of systems that is outside the scope of this chapter and this is where the dataset essentially comprises a set of separate text documents organised as a directory structure (folders and file), which can be implemented either as a physical (real) or logical directory structure. Such a system is less common, but EMC's Documentum system may be seen as an example of this type.

Also, we will not be dealing in detail with XML-based database systems, as we discussed the general principles when we looked at working with files and the reader is encouraged to re-read that chapter 6 if dealing with XML data. In terms of the mechanism for storing XML data, these are sometimes held individually as part of a directory structure as records in a document data store, such as MongoDB, or alternatively as Character Large Objects (CLOBS) in a relational database, such as Oracle. What we need to be concerned with is the speed of retrieval for queries. Exhaustive searches on large datasets would lead to unacceptable performance penalties, so we look at the underlying database to provide the

necessary metadata and indexing, regardless of the means by which the data is stored.

One example that links these together is BaseX, which is an open source, lightweight, high performance, and scalable XML Database engine and incorporates XPath/ XQuery processing. There is a REST API and several language implementations, although there is not yet one that is implemented in Julia. However, Python does have one such module: the PyCall module, mentioned previously and this can be employed here when working with BaseX.

Relational Databases

The primary difference between relational and non-relational databases is the way data is stored. Relational databases where not the first architectures to be implemented, those based on single (and then multiple) indices and values preceded them and as we will see later, these are making something of a comeback with the constraints of handing large datasets.

Relational data is tabular by nature, and hence stored in tables with rows and columns. Tables can be related to one another and cooperate in data storage as well as swift retrieval.

Data storage in relational databases aims for higher normalization — breaking up the data into smallest logical tables (related) to prevent duplication and gain tighter space utilization.

While normalization of data leads to cleaner data management, it often adds a little complexity, especially to data management where a single operation may have to span numerous related tables and since the databases are on a single server and partition tolerance is not an option, in terms of the CAP classification they are consistent and accessible.

Building and loading

Before looking at some of the approaches to handling relational data in Julia, I'm going to create a simple script that generates a SQL load file.

The dataset we will use comprises a set of "quotes", of which there are numerous examples online. We will find this data useful a later chapter, so we will create a database here.

There are only three (text) fields separated by tabs (rather than comma) and separate records per line, for example:

category <TAB> author <TAB> quote

Some examples are:

Classics *Aristophanes* *You can't teach a crab to walk straight.*
Words of Wisdom *Voltaire* *Common sense is not so common.*

For the script `etj.jl`, we will assume a simple command line will be:

etl.jl quodata.tsv [> loader.sql] # i.e the output is to be piped to a database

We wish to create a table for the quotes and another for the categories.

This is not totally normalized as there may be duplication in authors but this denormalization saves a table join.

The downside is that we can extract quotes by category more easily than by author, which will require definition of a foreign index (with corresponding DB maintenance penalty) or an exhaustive search.

The following SQL file (build.sql) will create the two tables we require:

```
create table categories (
   id integer not null,
   catname varchar(40) not null,
   primary key(id)
);

create table quotes (
   id integer not null,
   cid integer not null,
   author varchar(100),
   quoname varchar(250) not null,
   primary key(id)
);
```

In building the load file, we need to handle single quotes ('), which are used in SQL for text delimiters. The usual convention is to double them up (''), but some loaders also accept escaped backslashing (\').

Our script's command line is quite simple to check so we will work with it directly. In the case of the more complex command lines, Julia has an `ArgParse` package which has similarities with the Python `argparse module`, but some important differences too.

The command line script can be implemented as:

```
#! /Users/malcolm/bin/julia
using DelimitedFiles

nargs = length(ARGS);
if nargs == 1
    tsvfile = ARGS[1];
else
    println("usage: etl.jl tsvfile");
    exit();
end

# One liner to double up single quotes
escticks(s) = replace(s, "'" => "''");

# Read all file into a matrix (using DelimitedFiles)
# The first dimension is number of lines
qq = readdlm(tsvfile, '\t')
n = size(qq)[1];

# Store all categories in a dictionary
j = 0;
cats = Dict{String,Int64}();

# Main loop to load up the quotes table
for i = 1:n
  cat = qq[i,1];
  if haskey(cats,cat)
    jd = cats[cat];
 else
    global j = j + 1;
    jd = j;
    cats[cat] = jd;
 end
 sql = "insert into quotes values($i,$jd,";
 if (length(qq[i,2]) > 0)
   sql *= string("'", escticks(qq[i,2]), "',");
 else
   sql *= string("null,");
end
sql *= string("'", escticks(qq[i,3]), "');");
println(sql);
end

# Now dump the categories
for cat = keys(cats)
  jd = cats[cat];
   println("insert into categories($jd,'$cat');");
end
```

Notes:

1. The file is read using the DelimitedFiles package so we can specify the field separator as a TAB.
2. Because the quotes table will have an index into the categories table we will store all new categories in a hash (i.e. a `Dict()`)
3. So each category is check to see if it is in the hash using `haskey()` otherwise the category index is bumped up and the new category stored
4. At present the category has to be designation as `global`, hopefully this will not be forever.
5. This same approach can be used for Authors, making the relational normalisation better, but authors names are sometimes present differently, so we can search the database with an exhaustive query
6. Any input lines containing single quotes (`'`) will need to have these doubled up using the one-liner `escticks()` function
7. After writing the SQL statements for adding records to the quotes table, the same for the categories table can be created from the hash.
8. We are assuming here that the categories do **NOT** contain single quotes, otherwise we would need to apply `escticks()` here as well.

This produces an intermediate SQL load file that can be used with the most standard loaders that can output to STDOUT and piped into the loader. Indeed if the input file argument was also optional, it could be part of a Unix command chain.

The merits of using this approach is that it can be used to load any relational database with a SQL interface, and also it is easy to debug if the syntax is incorrect or we have failed to accommodate some particular aspect of the data (UTF-8, dealing with special characters, and so on.)

On the downside, we have to drop out of Julia to complete the load process. However, we can deal with this by either spawning the (specific) database load command line as a task or inserting the entries in the database on a line-by-line basis via the particular Julia DB package we are using.

Interfacing

By a native interface, with respect to interfacing with databases, I am referring to the case under which a package makes calls to an underlying shared library API rather than requiring some metabroker such as `ODBC` or `JDBC` or using calls to *(say)* a Python, or similar, module. This is a little different to the notion of native vs wrapper packages which has be

propounded previously when discussing Julia modules themselves.

SQLite

As an example, we are going to look at the case of `SQLite`, which is a simple DBMS-less style system. It will be built from source and a wide variety of precompiled binaries, all of which are obtainable from the main website at www.sqlite.org, along with installation instructions, which in some cases, is little more than unzipping the download file.

SQLite is usually designated as sqlite3 to differentiate it from the prior version and the command line is therefore: `sqlite3`

As a dataset, we will work with the "queries" tables for which we created a build and load file in the previous section.

To start SQLite, assuming it is on the execution path, we type: `sqlite3 [dbfile]`.

If the database file (`dbfile`) does not exist, it will be created, otherwise SQLite will open the file; if no filename is given, SQLite will work with an in-memory database that can be later saved to disk.

SQLite has a number of options that can be listed by typing `sqlite3 -help`

The SQLite interpreter accepts direct SQL commands terminated by a `;`

In addition to this, instructions to SQLite can be applied using a special set of commands prefixed by a dot (.), all are listed with the .help command.

Usually these commands can be abbreviated to the short forms so long as it is unique. For example `.read`can be shortened to `.re`.

Note that command is case sensitive and must be written in lowercase.

So we can use the following sequence of instructions to create a database from our quotes build/load scripts:

```
[bash]> sqlite3
# This is relative from a  shell on OSX (or Linux) and ...
# the prompt will depend on your own O/S setup

sqlite> .read build.sql
sqlite> .read load.sql
```

```
sqlite> .save quotes.db
sqlite> .ex
```

On OS X or Linux, we can alternatively pipe the build scripts and loader script as:

```
[bash]> cat build.sql | sqlite3 quotes.db
[bash]> julia etl.jl quodata.tsv | sqlite3 quotes.db
```

The quotes.db file is provided with the accompanying code to this chapter

So let us begin by running a simple SQL query to determine the number of records in the `quotes` table.

```
julia> using SQLite

# SQLiteDB() etc., are not exported, so fully qualify the call
julia> db = SQLite.DB("quotes.db")
SQLite.DB("quotes.db")

# Check which tables are in the database
julia> SQLite,tables(db)
2 rows × 1 columns
```

	name String?
1	categories
2	quotes

```
# Dump the column types for the table: "quotes"
julia> SQLite.columns(db,"quotes")
4 rows × 6 columns
```

	cid Int64?	name String?	type String?	notnull Int64?	dflt_value Any	pk Int64?
1	0	id	integer	1	missing	1
2	1	cid	integer	1	missing	0
3	2	author	varchar(100)	0	missing	0
4	3	quoname	varchar(250)	1	missing	0

Queries against the database should be recast into a dataframe :-

```
# Get the number or quotes in the database
julia> sql = "select count(*) from quotes";
julia> df = DataFrame(SQLite.Query(db, sql);
```

```
julia> df[1]
1-element Array{Union{Missing, Int64},1}: 36
```

Similarly we can display these, *set a limit as the first 10*:

```
julia> select * from quotes limit 10
10 rows × 4 columns
```

	id	cid	author	quoname
	Int64⍰	Int64⍰	String⍰	String⍰
1	1	1	Hofstadter's Law	It always takes longer than you expect, even when you take Hofstadter's Law into account.
2	2	2	Noelie Altito	The shortest distance between two points is under construction.
3	3	3	Scott's Law	Adding manpower to a late software project makes it later
4	4	2	Shaw's Principle	Build a system that even a fool can use, and only a fool will want to use it.
5	5	4	Adolf Hiltler	The great mass of the people will more easily fall victims to a big lie than a small one
6	6	5	G. B. Shaw	There is no satisfaction in hanging a man who does not object to it
7	7	1	Heller's Law	The first myth of management is that it exists
8	8	3	missing	There are two ways to write error-free programs. Only the third one works.
9	9	2	Fingle's Creed	Science is true. Don't be misled by facts.
10	10	1	missing	Today is the tomorrow you worried about yesterday

Not all of the quotes have an author (i.e. some are anonymous) and the query returns a `missing` type in this case.

The package Feather.jl can be used to store *snapshots* of queries, which can then be reloaded into a data frame.

```
julia> using Feather
julia> Feather.write("QuoSnap01.feather", df)
"QuoSnap01.feather"
```

Retrieving from a feather file is "lazy"; that is the metadata is read but records are all fetched when they are referenced. So feather files can hold very large datasets with little memory overhead.

```
julia> dfx = Feather.read("QuoSnap01.feather");
julia> size(dfx)
(5, 3)

julia> dfx[1,1] Get first quote
"It always takes longer than you expect, even when you take Hofstadter's
```

```
Law into account."
```

Let us find all quotes from Oscar Wilde, by specifying a WHERE clause:

```
julia> sql = "select q.quoname from quotes q ";
julia> sql *= " where q.author = 'Oscar Wilde'";

# Alternatively, if we do not wish to retain the dataframe
# ... we can just pipe it using the |> operator
julia> SQLite.Query(db,sql) |> DataFrame
6 rows × 1 columns
```

	quoname String?
1	The only way to get rid of a temptation is to yield to it.
2	There is only one thing in the world worse than being talked about, and that is not being talked about
3	I am not at all cynical, I have merely got experience, which, however, is very much the same thing
4	To love oneself is the beginning of a lifelong romance
5	We are all in the gutter, but some of us are looking at the stars
6	London society is full of women of the very highest birth who have, of their own free choice, remained thirty-five for years

To display category information, in addition to the information from the `quotes` table, requires specifying a join of the two tables:

```
# Just get the first 5 entries
julia> sql="select q.quoname,q.author,c.catname from quotes q ";
julia> sql *= "join categories c on q.cid = c.id limit 5";
julia> df = DataFrame(SQLite.Query(db,sql))
5 rows × 3 columns
```

	quoname String?	author String?	catname String?
1	It always takes longer than you expect, even when you take Hofstadter's Law into account.	Hofstadter's Law	Words of Wisdom
2	The shortest distance between two points is under construction.	Noelie Altito	Science
3	Build a system that even a fool can use, and only a fool will want to use it.	Shaw's Principle	Science
4	The great mass of the people will more easily fall victims to a big lie than a small one	Adolf Hiltler	Politics
5	There is no satisfaction in hanging a man who does not object to it	G. B. Shaw	Books & Plays

In addition to running queries the SQLite module can be used for executing DML (database

manipulation language) statements such as `insert, update, delete` and also to `create` and `drop` tables.

Finally we should note that it is possible to specify parametrised SQL statements using the `SQLite.Stmt()` routine to construct and prepare such and `SQLIte.bind!()` to *bind* values to parameter placeholders in the *prepared* statement.

Julia's DB API

In the previous edition of this SQLite support was presented via a DBD (database driver) to the DB-API. This was a database independent API, similar to the Plots API discussed in the previous chapter, an approach which has found widespread favour in Perl5 and to a lesser extent in Python.

It was proposed as an approach in Julia by John Miles White, and subsequently by Eric Davies, and SQLite and (partially) MySQL driver modules were provided. However the DBI module has fallen into a state of neglect and (to my mind) will remain so.

We will see that although Julia lacks a wide variety of native database packages, MySQL and Postgres are supported and with the rise of NoSQL/NewSQL databases, the usefulness of an API is debatable.

MySQL

MySQL is probably the most utilised open-source database, with web-based packages such as Wordpress, Joomla etc., almost exclusively using it as their backend datastore.

This is NOT the case in the Enterprise where Oracle, and on Windows servers Sqlserver , still are then most widely used and some time in the not to distant future, hopefully, Julia will have native interface packages to these database too.

Bundles such as Xampp (*Linux*), Mamp (*OSX*) and Wamp (*Windows*) can be download free and will install a MySQL dbms, together with an Apache web server and PHPinterpreter, together with a `phpMyAdmin` web application which can be used to create and maintain MySQL databases.

Note that following on from Oracle purchasing MySQL (as part of its previous sale to Sun Systems), the original developers, lead by Monty Widenius, recreated an alternative clone, MariaDB from what remained open source and from a users viewpoint MariaDB, acts entirely as MySQL, in particular w.r.t. the interface API. Indeed some of the major systems

providers, such as Redhat, are now bundling MariaDB, rather than MySQL, was part of their distros. So it this text when we refer to MySQL where actually mean MySQL AND MariaDB.

We are going to look at three different ways that Julia can be interfaced with MySQL as these can be carried over to many other database systems, which are:

- ODBC
- Native Julia
- PyCall

Chinook

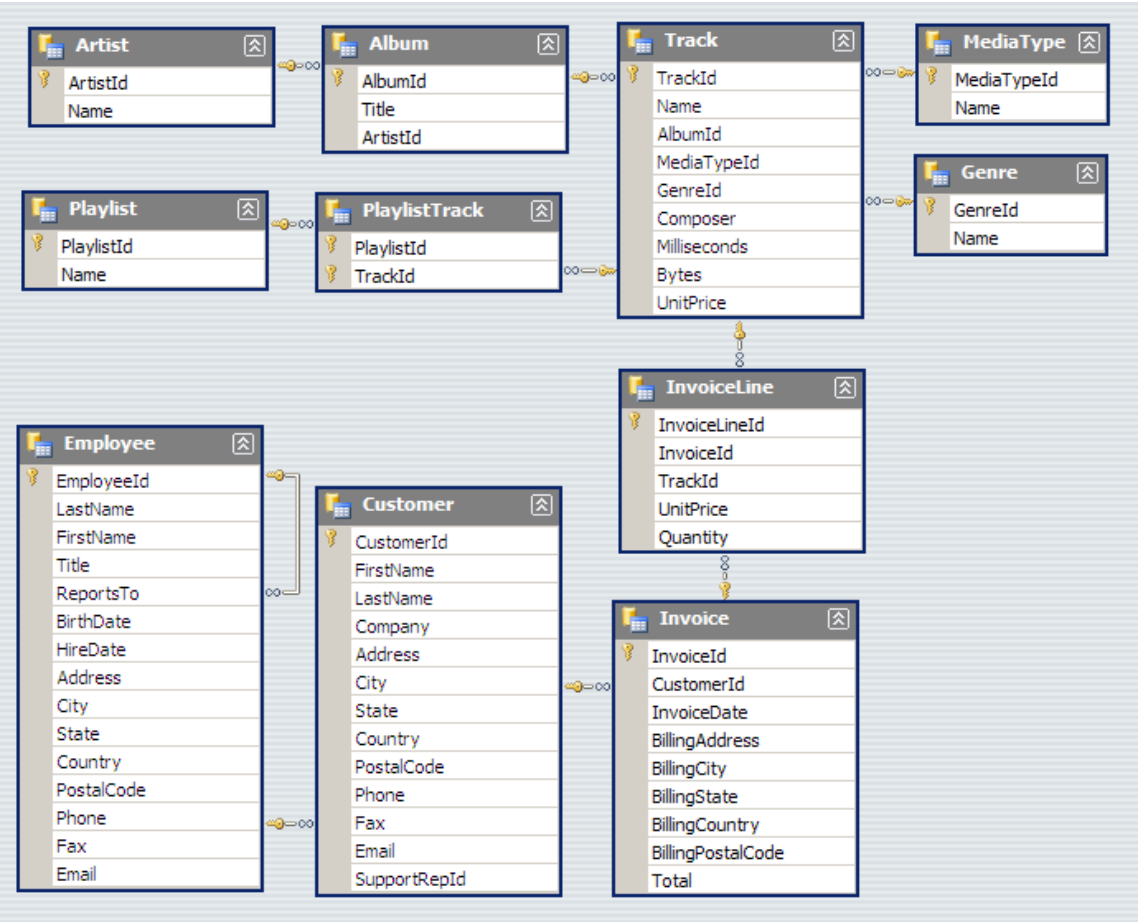
To look at some more sophisticated queries than in the previous section, I am going to introduce the Chinook database (https://chinookdatabase.codeplex.com) dating back to 2012 but still available. It is an open source equivalent to the Northwinds dataset that accompanied Microsoft Office. (*Chinook are winds of the north American plains*).

Chinook comes with load files for SQLite, MySQL, Postgres, Oracle, and many others.

The data model represents a digital media store, including tables for artists, albums, media tracks, invoices, and customers.

The media-related data was created using real data from an iTunes library, although naturally customer and employee information was manually created using fictitious names, addresses, emails, and so on. Information regarding sales were auto-generated randomly.

The database schema is shown in the following figure:



In the Chinook schema, there are two almost separate subsystems:

- Artist / Album / Track / Playlist, and so on
- Employee / Customer / Invoice

These are linked together via the `InvoiceLine` (TrackId) table to the Track table.

Notice that the schema is not totally normalised as the *InvoiceLine* table can also be joined to *PlaylistTrack* and bypassing the *Track* table.

ODBC

ODBC (Open DataBase Connectivity) was a level of middleware introduced by Microsoft in 1992. It adds an extra layer between the application program, the database interface and the underlying database to impose a standard mode of access.

I am no great lover of ODBC but it is oft touted in Julia as a fallback solution to interfacing with a database in a case where no other exists, so we need to discuss it here.

The principle criticism is one of speed, *there are others* : especially in situations involving large datasets and distributed (networked) systems.

To use ODBC requires the installation of a specific ODBC driver and then the setting up of a connection string, by use of an administration interface.

When working with MySQL and MariaDB, connectors which work with either can be downloaded from the MariaDB downloads site: `https://downloads.mariadb.org/`

For Windows, ODBC comes as standard and is found by use of the control panel: "system and security" / "administration tools" group.

With Unix and OS X, there are two administration managers available: `unixODBC` and `iODBC`. The former is more standard in approach: setting up the drivers by means of editing configuration files, while the latter is more GUI-based. I prefer to use `iODBC` when working on OSX and `unixODBC` on Linux, but both work, and it is largely a matter of choice; `unixODBC` does have a separate GUI wrapper based on Qt, but I have found it as easy just to use the command utility.

The following queries have been run under Ubuntu, using unixODBC and the MySQL driver. For installation and setting up of the connection string, the reader is referred to the web page at `http://www.unixodbc.org`

Both unixODBC and iODBC require two configuration files: `odbcinst.ini` and `odbc.ini`, the former specifies the drivers and the latter the data sources.

These can be placed in the /etc directory, which requires admin privileges, or as hidden files in the user's home directory `.odbcinst.ini` and `.odbc.ini`.

So to interface with the Chinook database using a MySQL ODBC driver my configuration files look like the following:

```
#!/bin/bash
cat /etc/odbcinst.ini
[MySQL]
Description=MySQL ODBC Driver
Driver=/usr/lib/odbc/libmyodbc5a.so
Setup=/usr/lib/odbc/libmyodbc5S.so
```

```
#!bash
cat odbc.ini
[Chinook]
Description=Chinook Database
Driver=MySQL
Server=127.0.0.1
Database=Chinook
Port=3306
Socket=/var/run/mysqld/mysqld.sock
```

Getting the configuration files for the drivers and data sources correct can be a little tricky at first, so it is possible to use the odbcinst utility to check them and also to connect with isql, and run some queries:

```
# Run these from a Unix shell
odbcinst -q -d; # =>  [MYSQL]odbcinst -q -s; # =>  [Chinook]
isql -v Chinook malcolm mypasswd
```

Assuming you can now connect with isql, then using ODBC in Julia is straightforward:

```
julia> using ODBC

# Connect via the valid DSN
julia> dsn = ODBC.DSN("Chinook",usr="malcolm",pwd="mypasswd");
# Return the number of a dataframe
julia> df = ODBC.query(dsn, "select count(*) from Customers");
julia> println("Number of customers: , df[1])Number of customers:  59
```

To demonstrate some more complex queries, let's join the Customers and Invoicetables by the customer ID, and by running a group by query select the customers who are the highest spenders, e.g. spending more the $45

```
julia> sql = "select a.LastName, a.FirstName,";
julia> sql *= " count(b.InvoiceId) as Invs, sum(b.Total) as Amt";
julia> sql *= " from Customer a";
julia> sql *= " join Invoice b on a.CustomerId = b.CustomerId";
julia> sql *= " group by a.LastName having Amt >= 45.00";
julia> sql *= " order by Amt desc;";

julia> df = DataFrame(SQLite.Query(db,sql));

# Looping through the dataframe, to produce better output
julia> using Printf
julia> for i in 1:size(df)[1]
  LastName = df[:LastName][i]
  FirstName = df[:FirstName][i]
  Amt = df[:Amt][i]
  @printf "%10s %10s %10.2f\n" LastName FirstName Amt
```

```
end
      Holý     Helena      49.62Cunningham    Richard      47.62     Rojas
Luis      46.62    Kovács   Ladislav      45.62  O'Reilly       Hugh
45.62
```

As a second example lets look at some tracks from Richard Cunningham

```
julia> sql = "select a.LastName, a.FirstName, d.Name as TrackName";
julia> sql *= " from Customer a";
julia> sql *= " join Invoice b on a.CustomerId = b.CustomerId";
julia> sql *= " join InvoiceLine c on b.InvoiceId = c.InvoiceId";
julia> sql *= " join Track d on c.TrackId = d.TrackId";
julia> sql *= " where a.LastName = 'Cunningham' and a.FirstName =
'Richard'";
julia> sql *= " limit 5;";
julia> df = DataFrame(SQLite.Query(db,sql));

julia> for i in 1:size(df)[1]
  TrackName = df[:TrackName][i]
  @printf "%s\n" TrackName
end
Radio Free EuropePerfect CircleDrowning ManTwo Hearts Beat As OneSurrender
```

(Native) MySQL

In the previous edition MySQL was only partially supported via a DBD driver in the DB-API, viz. select statements where not implemented. Now there is a native MySQL package with wraps around the shared library and provides a comprehensive interface.

On my OSX-based Mac, I am using MAMP-Pro, and have loaded the relevant Chinook SQL file. One of the features of MAMP is that the `mysql.sock` file is in a non-standard location, normally is is to be around in `/tmp`. Fortunately the connection routine allows this to be specified, otherwise it would be necessary to setup an symbolic link which would be removed on reboot since `/tmp` is cleared.

```
# Using MAMP (on OSX) so need to specify where the mysql.sock is.
# The username/password are NOT really the ones I'm using
# We can set the database to use in the connect statement
julia> conn = MySQL.connect("localhost", "malcolm", "mypasswd",
           db="Chinook",
           unix_socket="/Applications/MAMP/tmp/mysql/mysql.sock")
MySQL Connection----------------Host: localhostPort: 3306User: malcolm
DB:   Chinook
```

The module implements some specific MySQL statements such as `show tables`

```
julia> df = DataFrame(MySQL.query(conn,"show tables"));
```

```
julia> tb = df[:,1]
julia> print("Chinook tables:: ")
julia> for i = 1:size(tb)[1]
  print(tb[i]," ")
end
Chinook tables:: Album Artist Customer Employee Genre Invoice InvoiceLine
MediaType Playlist PlaylistTrack Track
```

All the customers are from the USA, so let us see how many have a last name starting with C.

```
julia> sql = "select FirstName,LastName,Address,City,State"
julia> sql *= " from Customer where LastName like 'C%'";
julia> DataFrame(MySQL.query(conn, sql))
2 rows × 5 columns
```

	FirstName String	LastName String	Address String?	City String?	State String?
1	Kathy	Chase	801 W 4th Street	Reno	NV
2	Richard	Cunningham	2211 W Berry Street	Fort Worth	TX

By joining on the `Invoice` table we can sum up the `Total` field and see the overall total spent by these customers

```
julia> sql = "select a.FirstName,a.LastName, sum(b.Total) as 'Total
spent'";
julia> sql *= " from Customer a";
julia> sql *= " join Invoice b on a.CustomerId = b.CustomerId";
julia> sql *= " group by a.FirstName, a.LastName"
julia> sql *= " having a.LastName like 'C%'"
julia> MySQL.query(conn,sql) |> DataFrame
2 rows × 3 columns

# If we are done then it is polite to drop the connection
julia> MySQL.disconnect(conn);
```

	FirstName String	LastName String	Total spent DecFP…?
1	Kathy	Chase	37.62
2	Richard	Cunningham	47.62

Using PyCall

We have seen previously that Python can be used for plotting via the PyPlot package that interfaces with matplotlib. In fact, the ability to easily call Python modules is a very powerful feature in Julia and we can use this as an alternative method to connect to

databases.

Although Python is much slower when compared with Julia, this is not as bad an option as it may appear at first glance since the interface to the database DBMS will be written in C and compiled. In cases such as Oracle (currently) this may be an interesting stopgap but is no substitute to a direct wrapper around Oracle's OCI.

Our current MySQL setup already has the Chinook dataset loaded, we will execute a query to list the Genre table.

In Python, we will first need to download the MySQL Connector module. For Anaconda, this needs to use the source (independent) distribution rather than a binary package, and the installation is performed using the `setup.py` file.

The query (*in Python*) to list the Genre table would be:

```
# Run is in Python
# The mysql connector has to be installed
#
import mysql.connector as mc
cnx = mc.connect(user="malcolm", password="mypasswd")
csr = cnx.cursor()
qry = "SELECT * FROM Chinook.Genre"
csr.execute(qry)
for vals in csr:
    print(vals)
(1, u'Rock')
(2, u'Jazz')
(3, u'Metal')
(4, u'Alternative & Punk')
(5, u'Rock And Roll')
...
...
csr.close()
cnx.close()
```

We can execute the same in Julia by using the PyCall module to the mysql. connector module and the form of the coding is remarkably similar:

```
julia> using PyCall
julia> @pyimport mysql.connector as mc;
julia>  cnx = mc.connect(user="malcolm", password="mypasswd");
```

Any database that can be manipulated by Python is also available to Julia.

In particular, since the DBD driver for MySQL is not fully DBT compliant, let's look at this approach to run the previous (Python) query.

```
julia> query = "SELECT * FROM Chinook.Genre"
julia> csr[:execute](query)
julia> for vals in csr
  id = vals[1]
  genre = vals[2]
  @printf "ID: %2d, %s\n" id genre
end
ID: 1, Rock
ID: 2, Jazz
ID: 3, Metal
ID: 4, Alternative & Punk
ID: 5, Rock And Roll
...
...
csr[:close]()
cnx[:close]()
```

Note that the form of the call is a little different from the corresponding Python method, since Julia is not object-oriented, the methods for a Python object are constructed as an array of symbols. For example the `csr.execute(qry)` Python routine is called in Julia as `csr[:execute](qry)`.

Also, be aware that although Python arrays are *zero-based*, this is translated to *one-based* by PyCall, so the first value is referenced as `vals[1]`.

Java and JDBC

In chapter, *Interoperability*, we discussed the use of the JavaCall package to interface Julia with the JVM (*Java Virtual Machine*). This gives Julia access to the entire suite of JVM modules and one especially important is JDBC (*Java Database Connectivity*), so much so that a separate package JDBC.jl has been written in order to simplify its use.

JDBC is another middle layer that functions similar to ODBC, and naturally there is a JDBC driver, termed a connector, for both MySQL and MariaDB, which can be obtained from the latter's `download` page.

As an alternative to the MySQL examples used previously, I am going to discuss a pure Java database, Derby and how to build/load/query it using the *quotations* dataset which we created earlier.

Derby

Derby is an Apache DB subproject; it is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0.

Some key advantages of Derby are that it . . .

- has a small footprint, about 3.5 Mb for the DB engine and embedded JDBC driver.
- is based on the Java, JDBC, and SQL standards.
- provides an embedded JDBC driver
- also supports the more familiar client/server mode with the Derby Network Client JDBC driver and Derby Network Server.
- is easy to install, deploy, and use.

The following instructions apply to OSX and Linux based systems, Windows will vary a little, reflecting differences in filesystems and setting environment variables.

```
#! /bin/bash
DH = $HOME/Derby  # Save some typing
export JAVA_HOME = $(/usr/libexec/java_home)
export PATH = $PATH:$DH/bin
export CLASSPATH = \
$DH/lib/derbytools.jar:$DH/lib/derbynet.jar:.

# Check that the setup is OK by using sysinfo
sysinfo
```

We should now be able to startup the Derby database from the command scripting the DERBY bin folder.

```
# Change to the Quotes directory to pickup the load files
cd /Users/malcolm/PacktPub/Chp09/Quotes
startNetworkServer &

# Now use the ij utility to look the dataset and ...
# ... run a query to see if the build/load was OK.
ij> connect 'jdbc:derby:Quotes;create=true';
ij> run 'build.sql';
ij> run 'qloader.sql';
ij> select count(*) from quotes;
36
```

After being created, an *in-memory* database can be kept (*persisted*) by using one of the backup system procedure such as:
`SYSCS_UTIL.SYSCS_BACKUP_DATABASE).`

It can then be restored as an in-memory database at a later time using the ***CALL*** command, or used as a normal file system database.

Since JDBC is an integrable part of the JVM it is possible to access the database without using Julia's JDBC module, i.e. just using *JavaCall,* `jcall` routine and the `@jimport` macro.

```
julia> using JavaCall
julia> jsd = @jimport java.sql.DriverManager;
julia> dbURL = "jdbc:derby:Quotes";
julia> conn = nothing;

julia> try
  db = jcall(jsd,"getConnection",JavaObject,(JString,),dbURL);
  jsp = @jimport java.sql.PreparedStatement;
  jsr = @jimport java.sql.ResultSet;
  sql = "select count(*) as K from quotes";
  stmt = jcall(db,"prepareStatement",isp,(JString,),sql);
  res = jcall(stmt,"executeQuery",jsr,());
  k = jcall(res,"getString",JString,(JString,),"K");
catch e
  println(e);
finally
  if conn != nothing
    jcall(conn,"close",Void,());
  end
end;

julia> println("\nNumber of quotes in database: $k);
Number of quotes in database: 36
```

The JDBC package uses of Java JDBC drivers to access databases from within Julia. Although it uses the `JavaCall.jl` package to call into Java it hides some of the details (viz. as above) from the programmer

The API provided provides a minimalistic Julian interface and supports Julia's DataStream methods directly.

```
# Create a cursor to the Quotes database
# ... and execute a SQL statement
julia> csr = cursor("jdbc:derby:Quotes")
julia> execute!(csr, "select * from my_table;")

# Then iterate over rows
julia> for row in rows(csr)
    #  access the data in 'row'
end
julia> close(csr);

# An alternative is to return a dataframe directly
```

```
# ... then then work with it.
julia> df = JDBC.load(DataFrame,
                      cursor(csr),
                      "select * from quotes")

# This works not only for DataFrames but for any Data.Sink
```

PostgreSQL

I have been a long time advocate of using Postgres as a relational database, working with it extensively. Postgres contained transactions, stored procedures, and triggers, long before these were added to MySQL

The old PostgreSQL.jl package was a DBD interface to the DB-API and has now been retired and replaced by LibPQ with is a wrapper around the PostgreSQL shared library, which is available after installing Postgres.

Installation is system dependent: I am using OSX, which has a DMG package. The database access is through a set of programs, so it is convenient to have these on the execution path and then it can be started using the `pg_ctl` program

```
# !/bin/bash
export PATH =
   /Applications/Postgres.app/Contents/Versions/latest/bin:$PATH
pg_ctl -D /data/psq -l logfile start
ps -ef | grep postgres   # use this to check it has started
```

We are going to work with Chinook; there is a load script available with the accompanying files. If is necessary to create an empty database using createdb and then build/load the tables with `psql`

```
#! /bin/bash
# The directory below is where my Chinook files are kept
cd /Users/malcolm/PacktPub/Chp09/Chinook/Dataset
createdb chinook
psql -d chinook -a -f  Chinook_PostgreSql.sql
```

`psql` when used interactively, is also the principal way to access, maintain and query the database.

There are a number of GUI's to work with Postgres (*some free*); - on OSX I can recommend the **Postico** app.

We can now connect via LibPQ which will return it results as a data stream:

```
julia> using LibPQ, DataStreams
julia> conn = LibPQ.Connection("dbname=chinook");
```

So feature of SQL in Postgres is that statements are case-insensitive, all covered to lowercase. With the Chinook dataset tables and fields are mixed case and so have to be quoted and in Julia the quotes need to be escaped!

```
julia> res = execute(conn, "SELECT * FROM \"MediaType\"");
julia> res = Data.stream!(res, NamedTuple)
(MediaTypeId = Union{Missing, Int32}[1, 2, 3, 4, 5],
  Name = Union{Missing, String}["MPEG audio file", "Protected AAC audio
file", "Protected MPEG-4 video file", "Purchased AAC audio file", "AAC
audio file"])

# Alternatively using a single routine : fetch!()
julia> res = fetch!(NamedTuple,
          execute(conn, "SELECT * as  FROM \"MediaType\""));
julia> res[:Name]
5-element Array{Union{Missing, String},1}:
"MPEG audio file"
"Protected AAC audio file"
"Protected MPEG-4 video file"
"Purchased AAC audio file"
"AAC audio file"
```

For complex queries, escaping all the tables and fields can become onerous. Fortunately it is possible to use a Julia multiline string, in which case single quotes are acceptable.

The query below finds the number of sales made by different employees .

```
julia> sqlx = """
select e."FirstName", e."LastName", count(i."InvoiceId") as "Sales"
 from "Employee" as e
 join "Customer" as c on e."EmployeeId" = c."SupportRepId"
 join "Invoice" as i on i."CustomerId" = c."CustomerId"
 group by e."EmployeeId"
 """;

# Loop through the NameTuple
julia> qry = fetch!(NamedTuple, execute(conn, sqlx))
julia> using Printf
julia> for i in 1:length(qry)
  @printf("%s %s has %d sales.\n",
     qry.FirstName[i], qry.LastName[i]. qry.Sales[i])
end
Margaret Park has 70 sales.
Jane Peacock has 97 sales.
Steve Johnson has 84 sales.
```

NoSQL databases

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model(s) address several issues that the relational model was not designed to when dealing with large volumes of structured, semi-structured, and unstructured data.

Types of NoSQL databases can roughly be classified under the following headings, although no taxonomy is perfect:

- **Key-value**: These are among the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value.
- **Document**: These pair each key with a complex data structure known as a document, which themselves may can contain many different key-value pairs or key-array pairs, or even nested documents.
- **Columnar**: These are optimized for queries over large datasets, and store columns of data together instead of rows.
- **Graphic**: These are used to store information about networks using a data model consisting of nodes, with associated parameters, and relations between the nodes.

NoSQL datastore technologies are especially prevalent in handling big data systems that have a large volume of data and require high throughputs,; often they are distributed over multiple physical servers.

We will briefly discuss Julia's (v1.0+) means of working with NoSQL databases both directly and by means of the REST (**RE**presentational **S**tate Transfer) **A**PI

Key-Value Datastores

Key-values (KV) systems are the earliest of databases, preceding relational ones.

The first were known as ISAM (index sequential access method) and continued in various forms when used in LDAP and Active Directory (AD) services, despite the onset of the SQL database era.

They have recently gained popularity as in-memory databases, IMDBs; which are database management systems that primarily relies on main memory for data storage rather than disk.

Because IMDBs on volatile memory devices will lose all stored information when the

device loses power or is reset, so maintain checkpointing, snapshots and transition logging.

A discussion of the (very) flexible KV database, *Redis*, follows and subsequently some of the others systems available via Julia is provided.

Redis

Redis uses a simple set of commands to create and retrieve records through a messaging system between server and client. It has a large number of datas structures and commands supporting them: simple scalars, lists, hashes and set, and these are all supportred by the Julia package.

Redis(.jl) also implements the Publish/Subscribe messaging paradigm where published messages are delivered to designated channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels and receive messages without knowledge of what (if any) publishers there are. This removes the need for polling of the database to retrieve uptodate information.

Setting up to use Redis is very straight-forward and there are two principal methods:

1. Get a Redis distribution and run it locally
2. Use a web-based system (available from Redis Labs)

In the first case a download are available from redis.io. This takes the form of a compressed archive , which when unzipped has utilities immediately ready to run. The binaries will execute on Linux and also OSX but not Windows. It is then a matter of putting these in a convenient folder and runnin`redis-server` as a detached process

There are some, usually quite outdated, Redis programs available for Windows; the reader is advised to google for these. Better still to use the second method, that is to register with `Redis Labs`, which is free and provides access for 30Mb of data. Connections are possible via Julia for either a local or a remote system, as is described below

To work with the command line it is necessary to use the redis-cli client utility, which is part of the method 1 download. There as also some Redis Management programs available, both web-based and as executables.

On registration Redis Labs provides a unique URL (via AWS), a port number on with Redis is running and an authentication password. A local installation does not need the password but clearly this is a necessity when working with a cloud-based service.

A simple session via `redis-cli` would be:

```
# The server, port and auth string are not mine
bash> redis-cli -p 99999 \
         -h redis-99999.z99.eu-west-1-2.ec2.cloud.redislabs.com

# It is also possible to do this via the CLI using -a switch
redis> auth aBcde18FGhiJklM77noPQrstuv
# PING will check all is working - the response if PONG
redis> PING
PONG
# Set a simple key-value, check it is set and the retrieve it
redis> set me "Malcolm Sherrington"
OK
redis> keys *
1) "me"
redis> get me
"Malcolm Sherrington"
```

With Redis.jl, we use the `RedisConnection()` routine - this defaults to using localhost and Redis' well-known port (6379) - but will accept parameters to connect to remote hosts.

So the above CLI session, in Julia, would be:

```
julia> const RLHOST =
     "redis-99999.z99.eu-west-1-2.ec2.cloud.redislabs.com";
julia> const RLPORT = 99999
julia> const RLAUTH = "aBcde18FGhiJklM77noPQrstuv"
julia> conn = RedisConnection(host=RLHOST, port=RLPORT, password=RLAUTH)
# Check with connection is alive and setup a key-values
julia> ping(conn)
PONG
julia> set(conn,"me","Malcolm Sherrington")
OK
# Note that the return status for the SET is the string "OK"
# List the keys
julia> keys(conn,*)
Set(AbstractString["me"])
# ... and return the value
julia> get(conn,"me")
"Malcolm Sherrington"
```

Redis has a number of SET commands from its various data structures and corresponding GET's, plus other associated message commands.

- scalars (*SET*/*GET*)
- multiple-scalars (MSET/MGET)
- lists (LSET/LINSERT/LPUSH/LPOP/LINDEX/LLEN)
- hashes (*HSET*/*HGET*/*HGETALL*)

- multi-hashes (HMSET/HMGET
- sets (*SADD*/*SCARD*)
- sorted-sets (ZADD/ZCARD/ZCOUNT)

There are a bewildering number of command, of which the move is a small subset.

See the online `command reference` documentation for a complete list and also the `commands.jl` file in the `Redis.jl/src` folder to see what has been implemented.

As an example, we will grab the stock market prices from `Quandl`, store them in Redis, retrieve them and display graphically.

It is necessary to decide on the form of the key and the type of data structure to use. The key needs to be a composition that reflects the nature of the data being stored.

Quandl returns stocks against a four (or less) character code and a set of values such as *Open, Close, High, Low, Volume*, plus the *Date*.

For Apple stocks, a choice of key for a closing price may be `APPL~Date~Open` where the `~` is a separator that will not occur in the data.

However, we could use a hash to store *Open, Close, High, Low* against the *Date*, but to retrieve this data we will need to make multiple queries.

Better would be to use a set of lists for each type of price (and the dates), so we only need a couple of queries to get the data, one for the price and a second for the dates.

.First we need a routine to get the stock data; this is via the HTTP package and a REST request, more of which later in this chapter and also in chapter 11.

The dataset is returned as a CSV file, which we know how to handle

A comment on Quandl : there are a limited number of queries to an IP-address anonymously (typically around 100). It is also possible to register with Quandl and increase this number, which is free for personal use.

The routine below handles both anonymous and authorised queries; again the key provided is not my real one.

The data is returned as a byte (UInt8) stream and CSV.read() can be used to convert this as a data frame. Typically this is in descending data order, so the routine reverse this.

```
julia> using HTTP,CSV
julia> const QURL = "https://www.quandl.com/api/v3/datasets/";
julia> const QAPIKEY = "ABCd1357EfgH2468yZ";
```

```
#=
This is NOT my real Quandl API key.
This could be set in the Julia startup file
=#

# Return the dataset as a dataframe, in ascending date order
julia> function quandl(qdset::String, apikey::String="")
  url = string(QURL,qdset)
  (length(apikey) > 0) && (url = string(url,"?apikey=",apikey))
  resp = HTTP.get(url);
  if resp.status == 200
    df = CSV.read(IOBuffer(resp.body));
    return sort!(df)
  else
    error("Can't get data from Quandl: $qdset")
  end
end
```

Get a couple of data frames for Apple (AAPL) and Microsoft (MSFT)

We are going to compare the closing prices of the two stocks and because the base prices are very different will scale each of stock to the price the first day, this way to demonstrate the underlying trends.

```
julia> qdf1 = quandl("WIKI/AAPL.csv", QAPIKEY);
julia> df = float.(qdf[:Close]);
julia> sf = 1.0/df[1];
julia> aapl = sf .* df;

julia> qdf2 = quandl("WIKI/MSFT.csv", QAPIKEY);
julia> df = float.(qdf[:Close]);
julia> sf = 1.0/df[1];
julia> msft = sf .* df ;
```

Quandl data (normally) finishes on the same date but may begin at different dates for different stocks. So we are going to plot the data for the period when **BOTH** stocks where listed

```
julia> n = minimum([length(aapl),length(msft)]);
julia> t = collect(1:n);.
```

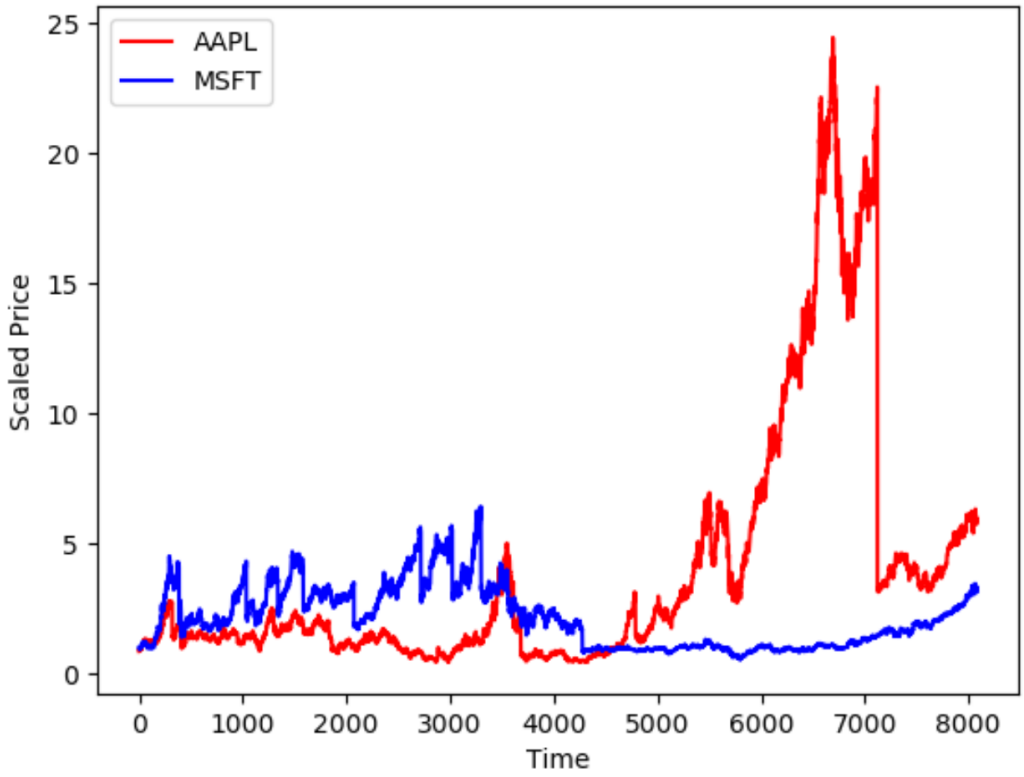
We will push these to Redis as a *list* using *keys* `AAPL~Close` and `MSFT~Close`

```
julia> conn = RedisConnection()julia> for i = n-1:-1:0
rpush(conn,'APPL~Close',aapl[end-i])
  rpush(conn,'MSFT~Close',msft[end-i])
end
```

And retrieve the values against these values and plot the results

```
# Redis lists are zero-based
julia> aapl_rdata = float.(lrange(conn,"AAPL~Close",0,n-1);
julia> msft_rdata = float.(lrange(conn,"MSFT~Close",0,n-1);

julia> plot(t, aapl_rdata, Color="red", label="AAPL")
julia> plot(t, msft_rdata, Color="blue", label="MSFT")
julia> xlabel("Time")
julia> ylabel("Scaled Price")
julia> legend()
```



Notice that Apple has performed better than Microsoft from around the start of the year 2000, probably on the back of iPads and iPhones and the return of a certain Mr Jobs.

The large drop in the AAPL price around the 7000 mark is due to a script issue by Apple to reduce the unit cost of their stock price, which corresponds to a reduction from $645.57 to $93.70 over the weekend 2014-06-06.

Other Key-Value systems

More than any other ecosystem group it seems that JuliaDatabases has had a series of fault starts, even to original name was highjacked by JuliaDB!

We have see that the DB-API project has been proposed, started and abandoned on more than one occasion and the merits of such an approach still remains a subject of oft heated discussion.

Below I've listed some key-value systems which, according to the github pages work with v1.0. In most cases this has been achieved by dropping support for Julia v0.6 and earlier, and if you are interested encourage you to look there.

Memcache

Memcache is an alternative to Redis, similar in operate but more limited in scope.; indeed Redis Labs offer Memcached databases as well as Redis ones.

Downloading and running locally and/or acquiring remote Memcached services is analogous to Redis.

Memcached provides only set/get functionality with a few associated routines such as append, prepend, incr and decl so is much more lightweight than Redis

However if simple key-values are required, this may be the better option.

The package is a pure Julia client and implements all commands as of v1.4.17
Both numbers, as well as strings are stored in a vanilla string format whereas other Julia types are serialized pre-storage.

LMDB

Lightning Memory-Mapped Database (LMDB) is an ultra-fast, ultra-compact key-value embedded data store developed by Symas for the OpenLDAP Project. It arose after Oracle acquired the Sleepycat BerkeleyDB software which was then being used by OpenLDAP as its datastore.

LMDB uses memory-mapped files, so it has the read performance of a pure in-memory database but still offering the persistence of standard disk-based databases, and is only limited to the size of the virtual address space.

LevelDB

LevelDB is Google's open source ondisk key-value storage library that provides an ordered mapping from string keys to binary values. In many applications where only key based accesses are needed, it tends to be a faster alternative than databases.

LevelDB was written in C++ with a C calling API included and this module provides a Julia interface to LevelDB using Julia's `ccall` mechanism.

Document databases

Document-oriented database are designed for storing, retrieving, and managing semi-structured data. The difference between these and purel key-value stores lies in the way the data is processed; in a KV store, the data is considered to be inherently opaque to the database, whereas a document-oriented system relies on internal structure in the document in order to extract metadata that the database engine uses for further optimization.

MongoDB

MongoDB is one of the most popular document databases that provides high performance, high availability, and automatic scaling. It is supported by a Julia package that is a wrapper around the C API, and to use this package it is necessary to have the Mongo C drivers installed and available on the library search path.

A record in MongoDB is a document, which is a data structure composed of field and value pairs and documents that are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents. Mongo itself is easy to install, and the server (daemon) process is run via the `mongod` executable. The location of the database is configurable, but by default is in /data/db that must exist and be writeable.

There is a Mongo shell client program (mongo) that can be used to add and modify records and as a query engine:

```
bash> mongo
MongoDB shell version: 2.6.7
> use test
switched to db test
> db.getCollectionNames()
[ "system.indexes" ]
```

If collection does not exist it is created by just inserting a record to it.
I have supplied a file `quotes.js` which comprises a set of insert statements such as:

db.quotes.insert({category:"Words of Wisdom",author:"Hofstadter's Law",quote:" It always takes longer than you expect, even when you take Hofstadter's Law into account"})

Its is also possible to alias the db.quotes by `qc = db.quotes` and then write the insert stament as

qc.insert({category:"Words of Wisdom",author:"Hofstadter's Law",quote:" It always takes longer than you expect, even when you take Hofstadter's Law into account"})

Loading ALL the quotes data is straight-forward on Linux/OSX, just redirect from the shell in the usual way and use the MongoDB client:

```
bash> cd /Users/malcolm/PacktPub/Chp09/Quotes
bash> mongo < quotes.js
```

If all goes well there should be a number of *documents* in the *quotes* collection.

```
bash> mongo
> qc = db.quotes
> qc.count()
36
```

Because Mongo is schema-less, in cases where the author is unknown, the field is not specified and also the category is provided in full as a text field and not as an ID.

The Mongo shell can also be used to retrieve records:

```
> qc.find({author:"Oscar Wilde"})
{"_id" : ObjectId("54db9741d22663335937885d"), "category" : "Books &
Plays", "author" : "Oscar Wilde", "quote" : "The only way to get rid of a
temptation is to yield to it." }
```

The object returned is clearly in JSON syntax (more accurately BSON); now our Mongo database has been loaded with a dataset we can start to use Julia on it.

The package I will use is `Mongoc.jl`. This imports the appropriate Mongo C-drivers and acts as an interface via these, most common O/S's have a driver.

Other packages (such as *Mongo.jl*/*LibBSON.jl*) have been written in the past but remain untouched after many years. However at the time of writing Scott Jones has provided a fork of these which need to be installed from master in his gitub account

```
(v1.1) pkg> add LibBSON#master
(v1.1) pkg> add https://github.com/ScottPJones/Mongo.jl#master
```

These do not load for me (*currently*) but are worth keeping an eye on in the future and

raising any issues.

So we will use use *Mongoc.jl,* which fortunately is solid and works with v1.0

```
julia> import Mongoc
julia> const mc = Mongoc

julia> client = mc.Client();   # Create a connection to Mongo
julia> db = client["test"];
julia> quotes = db["quotes"];

# Check that we are connected and can see the quotes collection
julia> length(quotes)
36
```

The previous query on Oscar Wilde is coded as:

```
# Note the need for triple quoting the BSON string
# Escaping in single "s (i.e. \") which work equally well
julia> mc.find_one(quotes, mc.BSON("""{"author" : "Oscar Wilde"}"""))
BSON("{ "_id" : { "$oid" : "5c5f2e27a5f0227251bb66bc" }, "category" :
"Books & Plays", "author" : "Oscar Wilde", "quote" : "The only way to get
rid of a temptation is to yield to it." }")
```

CRUD

Create/**R**ead/**U**pdate/**D**elete operations (CRUD) are all supported by Mongoc. We saw above the **R**ead `find_one()` method and will concentrate here on creating and deleting records. Updating is possible using `update_one()` and `update_many()` and the package supports transactions so a large update can be achieved by a combination of a delete and an insert.

We start by setting up the BSON representation for a couple of quotes:

```
julia> doc1 = mc.BSON()
julia> doc1["author"] = "Orson Welles";
julia> doc1["quote"] = "If you want a happy ending, it all depends on where
you stop your story";
julia> doc1["category"] = "Words of Wisdom";

julia> doc2 = mc.BSON()
julia> doc2["author"] = "Bo Bennett";
julia> doc2["quote"] = "Visualization is daydreaming with a purpose";
julia> doc2["category" ] = "Computing";
```

BSON docs can be created from passing JSON string and/or a Julia Dict as a parameter; alternatively the BSON object can be converted back to these formats using `as_json()` and

`as_dict()` respectively.

```
julia> mc.as_dict(doc2)
Dict{Any,Any} with 3 entries:
 "quote" => "Visualization is daydreaming with a purpose"
 "author" => "Bo Bennett"
 "category" => "Computing"
```

To insert a record we use the `push!()` routine

```
julia> res1 = push!(quotes, doc1)
Mongoc.InsertOneResult(BSON("{ "insertedCount" : 1 }"),
"5c5f5869ebf7a80449023ab2")

julia> oid1 = res1.inserted_oid
"5c5f5869ebf7a80449023ab2"

julia> length(quotes)
37
```

To delete a record we setup a selector by creating a selector.
The simplest way is to use the `oid` since this is unique to each document.

```
julia> selr = mc.BSON();
julia> selr["_id"] = oid1;
julia> mc.delete_one(quotes, selr)
BSON("{ "deletedCount" : 1 }")
```

Using the `empty!()` method will clear out a collection (or database) by leave it in place; to remove it completely use `destroy!()`

Making call to `destroy!(client)` is the way to terminate Mongoc.

It is possible to do multiple inserts by passing an array of *docs (as BSON)* to `append!()`

```
julia> append!(quotes, [doc1, doc2])
Mongoc.BulkOperationResult(BSON("{ "nInserted" : 2, "nMatched" : 0,
"nModified" : 0, "nRemoved" : 0, "nUpserted" : 0, "writeErrors" : [  ] }"),
0x00000001, Union{Nothing, String}["5c5f5a16ebf7a80449023ab3",
"5c5f5a16ebf7a80449023ab4"])

julia> length(quotes)
38
```

MongoDB also provides map-reduce operations to perform aggregation which is supported by Mongoc.jl

RESTful interfacing

REST refers to a software architecture style designed to create scalable web services. It has gained widespread acceptance across the Web, as a simpler alternative to *SOAP* and *WSDL*-based web services.

RESTful systems typically communicate over hypertext transfer protocol with the same HTTP verbs (*GET, POST, PUT, DELETE*, and so on) used by web browsers to retrieve web pages and send data to remote servers.

With the prevalence of web servers, many systems now feature REST APIs and can return plain text or structured information. A typical example of plain text might be a time-of-day service, but structured information is the more common for complex requests as it contains meta information to identify the various fields.

Historically this was returned as XML, which is still common in SOAP web services, but more popular recently is JSON, since this is more compact and ideal for the Web where bandwidth may be limited. As with XML, which we looked at earlier, the JSON representation can be converted into an equivalent Julia hash array (Dict) expression .

To access them we need a method to mimic the action of the web browser programmatically and to capture the returned response.

We saw earlier that this can be done using a task, such as `curl`, with the appropriate command line:

```
rts = chomp(read(`curl –s http://amisllp.com/now.php`, String))
```

These will run the REST web page now which is a simple PHP script on my corporate website, and will return the current (UK) date and time.

```
julia> println(rts);
2015-02-18 12:11:56
```

Alternatively in Julia, we can use the HTTP.jl package in the same fashion that was used in the routine provided to get the stock datasets from Quandl. This is an example of a *REST* service. The topic in this section is to explore how this can be used as access to a database.

REST has become so prevalent that even when systems provide a more conventional API, they often implement a REST as well, albeit this may be a little more restrictive.

JSON and BSON

Web databases tend to use JSON as the format for storing and retrieving records rather than the more verbose XML style. JSON is well supported in Julia via the JSON.jl package.

Below are some details of my company in JSON.

```
julia> company = """{
  "founder" : "Malcolm Sherrington",
  "company" : "AMIS Consulting LLP",
  "website" : "amisllp.com",
  "partners" : 5,
  "incorporated" : "1981-06-01",
  "areas_of_work" :
     ["Aerospace","Healthcare","Finance","Web Development"],
   "experience_years : [31, 26, 15, 21],
   "programming_languages" :
     ["Fortran","C","Perl","PHP","Julia","Python","R"]
}"""
"{\n\"founder\" : \"Malcolm Sherrington\",\n\"company\" : \"AMIS Consulting
LLP\",\n\"website\" : \"amisllp.com\",\n\"partners\" : 5,\n\"incorporated\"
: \"1981-06-01\",\n\"areas_of_work\" :
[\"Aerospace\",\"Healthcare\",\"Finance\",\"Web\"],\n\"experience\" : [31,
26, 15, 21],\n\"programming_languages\" :
[\"Fortran\",\"C\",\"Perl\",\"PHP\",\"Julia\",\"Python\",\"R\"]\n}"
```

This can be parsed as:

```
julia> import JSON
julia>  amis = JSON.parse(company)
Dict{String,Any} with 8 entries:
"areas_of_work"  =>
     Any["Aerospace", "Healthcare", "Finance", "Web"]
"partners" => 5
"incorporated"  => "1981-06-01"
"founder" => "Malcolm Sherrington"
"company" => "AMIS Consulting LLP"
"website" => "amisllp.com"
"programming_languages" =>
    Any["Fortran", "C", "Perl", "PHP", "Julia", "Python", "R"]
"experience_years" => Any[31, 26, 15, 21]
```

Parsing returns the details as a `Dict`. Data types are inferred where possible but arrays

are returned as *Any[],* since JSON does not impose restrictions on mixed types, so it may be necessary to cast or broadcast the type(s):

```
julia> using Dates
julia> Date(amis["incorporated"])
1981-06-01
julia> Integer.(amis["experience_years"])
4-element Array{Int64,1}:
 31
 26
 15
 21
```

BSON

BSON was met previously in the section on MongoDB. It is the binary encoding of JSON-like documents which Mongo popularised when storing collections of documents: one reason being that it adds support for data types like Date and binary that are not covered by JSON.

- BSON is a serialization format encoding format while JSON is in a human-readable standard file format.
- BSON is designed such that it (*normally*) needs less space than JSON, but it is not extremely efficient.
- Also it is designed in a way that it has a comparatively faster encoding and decoding technique.

In Julia JSON support was well established whereas handling BSON was patchy. Recently Mike Innes has created a BSON.jl package which seems to be solid. As we saw Mongoc.jl package has another implementation of BSON.

Web Databases (CouchDB)

Examples of common systems that provide a REST API are:

- **Rick**: This is an alternative key-value datastore, which operates more as a conventional on-disk database than the in-memory Redis system
- **CouchDB**: is an open source system, now part of the Apache project, It uses JSON-style documents and a set of key-value pairs for reference
- **Couchbase**: is a priority product which takes the CouchDB protocol. There is a Lite version which is free for personal use (originally named *TouchBase*) and Couchbase-Mobile which targets Android (viz mobile phones). It has both a

REST API and a conventional one.

- **Neo4j:** is a prominent example of a graphics database that stores nodes, parameters, and interconnect relationships between nodes in a sparse format rather than as conventional record structures.
- **BigQuery**: is a RESTful web service that enables interactive analysis of massively large datasets working in conjunction with Google Storage. It is an Infrastructure as a Service (IaaS) that may be used complementarily with MapReduce.

Now let's will look peek into the *HTTP.jl* package and see how we can use this with *CouchDB*

HTTP package

Working with web-based systems has had a checkered history in the past in Julia. Previously there was a suite of modules covering client access, web servers and web sockets. The client-side was replace but a Requests.jl package with has been deprecated and incorporated into a new package HTTP.jl which covers all three aspects of working on the web.

Client access is the most common when accessing RESTful services and we will look at it here, the others will be discussed in chapter 11, "*Working in the cloud*"

In fact we saw a simple use of HTTP in getting data with the `quandl()` routine developed in the Redis section and using the "GET" protocol. This is used in acquiring financial data (and elsewhere) in modules such as *MarketData.jl*, *Temporal.jl* etc..

The primarily call is HTTP.request with returns an HTTP response

```
HTTP.request(method, url [, headers [, body]]; options])
```

The methods make be one of GET, POST, PUT, HEAD, PATCH, DELETE and for convenience an aliases for each is defined, such as

```
HTTP.get(. . .) => HTTP.request(GET, . . .)
```

The response returned comprises 3 parts, split from the stream, by the routine:

- **status**: the HTTP status code - this is normally 200 for OK but others are possible
- **header**: additional information about the response returned as a Dict, such as a server type, content-length and content-type (*very useful!*)
- **body**: the actual content to be displayed by the browser or to be handled by the program. This is returned as a {UInt8} vector and normally will need to be

changed to a string before processing.

There are a number of options to set basic-authorisation, retry count, timeout etc., not all options are valid with the different methods and full details in to be found in the HTTP.jl documentation.

As an example I have setup a simple webpage `testpage.html` on our corporate web server which displays an image of the Earth, *try it out*: `http://amisllp.com/testpage.html`

```
julia> testpage = "http://amisllp.com/testpage.html";
julia> r = HTTP.get(url, retry=false, readtimeout=30);
julia> r.status
200
julia> r.headers
6-element Array{Pair{SubString{String},SubString{String}},1}:
           "Date" => "Mon, 11 Feb 2019 11:45:03 GMT"
         "Server" => "Apache"
  "Last-Modified" => "Mon, 11 Feb 2019 09:36:27 GMT"
  "Accept-Ranges" => "bytes"
 "Content-Length" => "320"
   "Content-Type" => "text/html"

julia> String(r.body)
"<html>\n<head>\n<title>Hello World</title>\n<style
type=\"text/css\">\nh1\t{text-indent: 160px; font-family:Arial, Helvetica,
Sans-Serif; font-size:48px;}\np\t{font-
size:16px;}\n</style>\n</head>\n<body bgcolor = \"#ffffdc\" text =
\"#000000\">\n<br/>\n<h1>Hello, World!</h1>\n<p><img
src=\"images/earth.png\" border=\"0\"</p>\n</body>\n</html>\n"
```

An alternate way is via the HTTP.open() routine and which sends a HTTP Request Message and opens an IO stream from which the Response can be read and processed sequentially.

```
julia> HTTP.open("GET", testpage) do http
  i = 0
  while (!eof(http) && (i <= 8))
    i = i + 1
    s = read(http)
    (i > 1) && println(s)
  end
end
HTTP.Messages.Response:
"""
HTTP/1.1 200 OK
Date: Mon, 11 Feb 2019 09:52:34 GMT
Server: Apache
Last-Modified: Mon, 11 Feb 2019 09:36:27 GMT
```

```
Accept-Ranges: bytes
Content-Length: 320
Content-Type: text/html
```

CouchDB

The Apache organisation supplies a number of binary downloads for CouchDB for Linux, OS X, and Windows. It is written in *Erlang*, building from source is a little more involved but presents no real difficulties.

Starting up the server daemon on the localhost (127.0.0.1) runs the service on the 5984 default port. Querying this confirms that it is up and running:

```
bash> curl http://localhost:5984
{"couchdb":"Welcome","version":"2.3.0","git_sha":"07ea0c7","uuid":"8887b231
903e5088c8f92b490528a2c2","features":["pluggable-storage-
engines","scheduler"],"vendor":{"name":"The Apache Software Foundation"}}
```

Next, we need to create a database comprising the quotes dataset using either the PUT or POST command and show it exists:

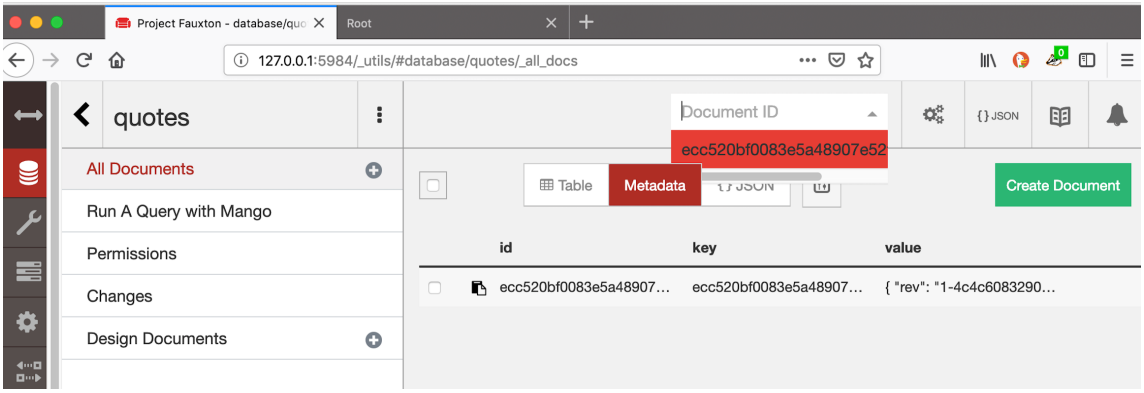
```
bash> curl -X PUT http://localhost:5984/quotes
bash> curl http://localhost:5984/_all_dbs
["quotes"]
```

To add records to quotes, we have to specify the content type as a JSON string:

```
bash> curl -H 'Content-Type: application/json' \
-X POST http://127.0.0.1:5984/quotes \
-d '{category:"Computing",author:"Scott's Law",
"quote":"Adding manpower to a late software project makes it later"})'
```

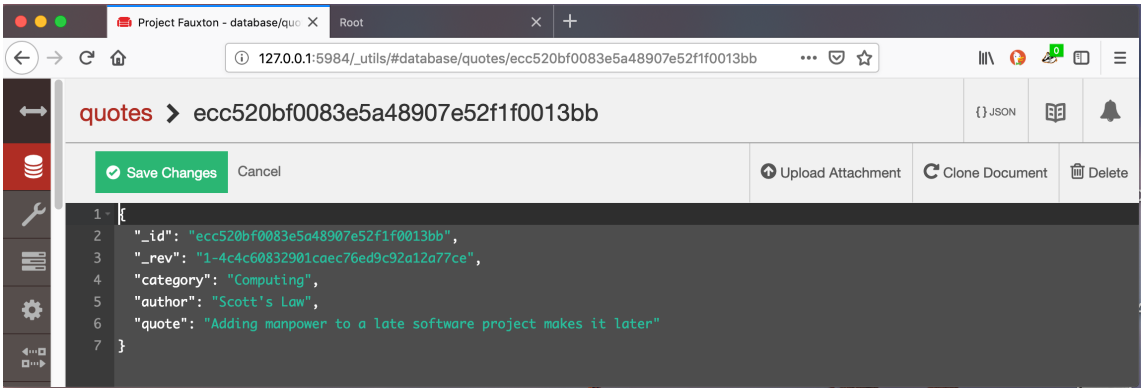
To do this using `curl` is possible but not very flexible; so CouchDB contains a utility IDE called *Futon* that can run in the browser using http://locahost:5984/_ utils:

The main page of Futon displays the existing databases and has an open in the menu bar to create a database. The menu bar changes with the context of the current page being displayed but there is also a set of "*fixed*" sidebar icons which dispatch to various useful CouchDB pages.



The figure above shows the state of the quotes database create which now has a single document inserted via the curl command. It can be seen that there is a *[Create Document]* button which opens a separate screen and assigns a value for `_id` in the new document as part of a JSON string. The remaining fields need to be entered in this string and then the document saved.

Clicking on the document which has already been added shows the format required:



This screen can be used to amend exiting fields: *author*, *category* and/or *quote* - do not change *_id* or *_rev*, these are external to the CouchDB system .

So armed with the HTTP and JSON modules we can now interact with the CouchDB datastore in Julia.

`_all_dbs` returns a list of the current databases. Assuming the status is OK (200), *which should be checked*, then we are interested in the response body.

Recall that this will be in the form of byte array, which when converted is represented as a

JSON string, and this is more easily handled as a Dict by using `JSON.parse()`

```
julia> using HTTP, JSON
julia> cdp ="http://localhost:5984"
julia> dbs = String(HTTP.get(cdp*("/_all_dbs")).body)
julia> JSON.parse(dbs)
1-element Array{Any,1}:
"quotes"
```

From the *quotes* database the _all_docs page returns the number of documents in the database (*total_rows*) and an array of each document, listing their ids, key and an array which contains the revision history.

The difference between the `id` and `key` is that the latter can be specified when the document is created or changed to something more meaningful and is used to aid in composing queries. A key need not be unique, unlike the id, indeed this is a way to group documents together under the banner of a common key.

```
julia> json = JSON.get("http://127.0.0.1:5984/quotes/_all_docs");
"{\"total_rows\":1,
\"offset\":0,
\"rows\":[\n{\"id\":\"ecc520bf0083e5a48907e52f1f0013bb\",
\"key\":\"ecc520bf0083e5a48907e52f1f0013bb\",
\"value\":{\"rev\":\"1-4c4c60832901caec76ed9c92a12a77ce\"}}\n]}"

julia> rec = JSON.parse(json)
Dict{String,Any} with 3 entries:
 "rows" =>
Any[Dict{String,Any}("key"=>"ecc520bf0083e5a48907e52f1f0013bb","id"=>"ecc52
0bf0083e5a48907e52f1f0013bb","value"=>Dict{String,Any}("rev"=>"1-4c4c608329
01c…
 "offset" => 0
 "total_rows" => 1

julia> rec["rows"][1]["key"]
"ecc520bf0083e5a48907e52f1f0013bb"
```

The document can then be retrieved by using its key:

```
julia> db = "quotes";
julia> key = "ecc520bf0083e5a48907e52f1f0013bb";
julia> json = String(HTTP.get("$cdp/$db/$key").body);
julia> doc = JSON.parse(json)
Dict{String,Any} with 5 entries:
"quote"   => "Adding manpower to a late software project makes it later"
 "_rev"   => "1-4c4c60832901caec76ed9c92a12a77ce"
"author"  => "Scott's Law"
```

```
"_id"     => "ecc520bf0083e5a48907e52f1f0013bb"
category" => "Computing"

julia> using Printf
julia> @printf "%s [%s]" doc["quote"] doc["author"]
Adding manpower to a late software project makes it later
[Scott's Law]
```

CouchDB does not provide a method of bulk loading documents, so one would need to be written. From a TSV or CSV this is quite straight-forward, read each line from the input file, transform as a JSON string and use `HTTP.put()` to add it to an existing database

For NodeJS fans there is a `couchimport` script which can be install using `npm`

```
bash> npm install -g couchimport
```

`couchimport` assumes TSV as the default format but other field separators can be specified by the `--delimiter` switch; for a full list of options use `--help`

Using the quotes.tsv file, this can be loaded into CouchDB as:

```
bash> cat quotes.tsv | \
          couchimport --url http://localhost:5984 --db quotes

couchimport
-----------
 url : "http://localhost:5984"
 database : "quotes"
 delimiter : "\t"
 buffer : 500
 parallelism : 1
 type : "text"
-----------
couchimport Written ok:35 - failed: 0 - (35) +0ms
couchimport {documents:35, failed:0, total:35, totalfailed:0 }
couchimport writecomplete { total:35, totalfailed:0 } +0ms
couchimport Import complete +70ms
```

and the load confirmed in Julia:

```
julia> doc = JSON.parse(String(HTTP.get(cdp*("//quotes")).body))
julia> doc["doc_count"]
35
```

With a more extensive set of records we can query the *quotes* database by creating and posting a selector: the following will find all quotes by Noeilie Altito - there is only one! :

```
{"selector": {"author": "Noelie Altito"}}
{
 "_id": "ecc520bf0083e5a48907e52f1f01bcdd",
 "_rev": "1-91125503fb20070a72a29bc481d81e60",
 "category": "Science",
 "author": "Noelie Altito",
 "quote": "The shortest distance between two points is under construction."
}
```

The query language supports a number of select operators, such as $gt, $eq, $and, $or etc.; one of he more useful is `$regex` which can be used in wildcard queries.

The following selector will find quotes in categories: "Computing" and "Classics"

```
{"selector": {"category": {"$regex": "^C"}}}
```

JuliaDB

JuliaDB is a package for working with persistent data sets which that can

- Load multi-dimensional datasets quickly and incrementally.
- Index the data and perform filter, aggregate, sort and join operations.
- Save results and load them efficiently later.
- Use built-in parallelism to operate on a single machine or cluster.
- Provide distributed array and table data structures with functions to load data from CSV.

JuliaDB ties together several existing packages including *Dagger.jl* and *IndexedTables.jl.*

The parallel/distributed features of JuliaDB are available by either . . .

- starting Julia with N workers: `julia -p N` or
- calling `addprocs(N)` before using JuliaDB

Multiple processes may not be benificial for datasets with less than a few million rows.

JuliaDB can operate on a large group of CSV files and it will build and save an index of the contents of those files; optionally it will “ingest” the data, which converts it to a more efficient *mmap*-able file format.

It is then possible open and operate on a dataset and JuliaDB will handle loading and storing only the necessary blocks from and to disk, making it possible to handle both dense and sparse data of any size and dimension. Because of adhering to the DataStreams protocols it is also possible to work on queries from backend databases.

Additionally it works with Julia's distributed parallelism and also supports out-of-core computation (*via Dagger*).

There is an extensive tutorial `online` which uses a flight dataset of around 18Mb and comprising 230K lines in CSV format. The datafile and a Jupyter notebook are available and also I have provided them with the supporting files to this chapter. The notebook covers many of the aspects of JuliaDB and the reader is encouraged to run it.

Stock pricing

JuliaDB also installs with a folder including a set of financial stock prices in the subdirectory `test/sample`. I will use this dataset and have copied it to a `Files/Stocks` folder under the Chp09 files. These are for the daily closing prices in the years from 2010 to 2015 , as separate CSV files, for Google (*GOOGL*), Goldman-Sachs (*GS*), Coca Cola (*KO*) and Xerox (*XRX*)

The dataset(s) have 7 fields date and ticker, push the usual open, high, low, close and volumes (OHLCV) with which we are now familiar.

JuliaDB supports NDSparse arrays so by pointing at the folder containing all the CSV files, the data can be loaded into a single dataset with a single call.

```
julia> using JuliaDB, IndexedTables
julia> path =  joinpath(homedir(),
                        "PacktPub","Chp09","Files","Stocks")

# Indicate that date and ticker fields should be indexed.
julia> stockdata = loadndsparse(path,
                                indexcols=["date","ticker"])

2-d NDSparse with 288 values (5 field named tuples):date        ticker   |
open    high    low    close
volume──────────────────────────┼──────────────────────────────────────────
──────────────────2010-01-01  "GOOGL" | 626.95  629.51  540.99 626.75
1.78022e82010-01-01  "GS"    | 170.05  178.75  154.88 173.08
2.81862e82010-01-01  "KO"    | 57.16   57.4301 54.94  57.04
1.92693e82010-01-01  "XRX"   | 8.54    9.48    8.91   8.63
3.00838e82010-02-01  "GOOGL" | 534.602 547.5   531.75 533.02
1.03964e82010-02-01  "GS"    | 149.82  160.21  156.99 153.13
```

```
2.3197e82010-02-01  "KO"     | 54.51   55.92   53.09  54.38
2.28993e82010-02-01  "XRX"    | 8.75    9.4     9.31   8.97   3.10746e8
```

1. The other routine to load a regular table rather than sparse on is `loadtable()`
2. The returned datatype is a named tuple and not a data frame.
3. Lookups using date and ticker can use arrays, slices and ranges as indices.

```
# Single values may be shown in the usual way
julia> using Dates
julia> stockdata[Date("2010-06-01"), "GOOGL"]
(open = 480.43, high = 509.25, low = 457.83, close = 482.37,
 volume = 1.196056e8)

# Or we can define a date range and select a couple of stocks
julia> stockdata[Date("2012-01"):Dates.Month(1):Date("2014-12"), ["GOOGL",
"KO"]]
2-d NDSparse with 72 values (5 field named tuples):date        ticker   |
open     high     low     close
volume──────────────────────────┼──────────────────────────────────────────
──────────────────────2012-01-01  "GOOGL" | 652.94   670.25   584.0
665.41 1.47137e82012-01-01  "KO"     | 70.15    70.71    67.98   70.14
1.50116e82012-02-01  "GOOGL" | 584.94   625.6    619.77  580.83
9.46335e72012-02-01  "KO"     | 67.88    69.98    69.5    67.85
1.43833e82012-03-01  "GOOGL" | 622.26   658.589  653.49  622.4
9.45647e72012-03-01  "KO"     | 69.87    74.39    74.14   69.6
2.05125e82012-04-01  "GOOGL" | 640.77   653.14   616.082 646.92
1.17637e82012-04-01  "KO"     | 73.83    77.82    76.9    74.14  1.46185e8
```

To reduce the dataset we use the filter function, this returns a new table.

```
# Values of Goldman Sachs with closing prices in [100.0,140.0]
julia> filter(x ->  x.close >= 100.0 && x.close <= 140.0,
               stockdata[:, "GS"])
1-d NDSparse with 17 values (5 field named tuples):date       | open
high     low     close
volume──────────────┼──────────────────────────────────────────────────────
──────────2010-07-01 | 131.69  153.41   153.15  131.14
2.2909e82010-09-01 | 139.01  154.7    146.98  139.74  1.402e82011-06-01 |
139.95  140.0    133.8   136.17  1.35784e82011-07-01 | 133.43  139.25
136.6   136.65  1.13317e82011-08-01 | 136.92  137.34   117.8   134.15
2.3048e82011-09-01 | 115.55  115.55   98.4    112.16  1.66764e82011-11-01
| 103.49  109.257  96.01   103.54  1.50194e82012-02-01 | 112.61  118.66
118.13  113.45  1.1445e8
. . . . .
#=
Get records for Xerox where the first tracing day (of the month) is a
Friday
=#
```

```
julia> filter((1=>Dates.isfriday,
               2=>x->startswith(x, "X")), stockdata)

2-d NDSparse with 10 values (5 field named tuples):date        ticker |
open   high    low    close
volume──────────────────────────┼──────────────────────────────────────────
──────────────────2010-01-01  "XRX"   | 8.54   9.48    8.91   8.63
3.00838e82010-10-01  "XRX"   | 10.41  11.73   11.7   10.5
2.73763e82011-04-01  "XRX"   | 10.71  11.03   10.18  10.88
2.79543e82011-07-01  "XRX"   | 10.38  10.83   9.55   10.71
2.71723e82012-06-01  "XRX"   | 7.06   7.88    7.88   7.12
2.4474e82013-02-01  "XRX"   | 8.1    8.23    8.2    8.02
1.71174e82013-03-01  "XRX"   | 8.07   8.77    8.62   8.15
1.63994e82013-11-01  "XRX"   | 10.01  11.48   11.41  10.02
2.2431e82014-08-01  "XRX"   | 13.18  13.88   13.77  13.04
1.40391e82015-05-01  "XRX"   | 11.55  11.859  11.5   11.52  2.09801e8
```

We can use JuliaDB to efficiently perform some map-reduce operations. Let's define the daily spread `[High - Low]` and gain `[Open - Close]` and compute the average value for Google

```
julia> googl = stockdata[:, ["GOOGL"]];

julia> spread = map(x -> x.high - x.low, googl)
julia> round(reduce(+,(mean.(spread)))/length(spread), digits=4)
32.7057

julia> gain = map(x -> x.open - x.close, googl)
julia> round(reduce(+,(mean.(gain)))/length(gain), digits=4)
-0.3395
```

This the dataset is is only for one trading day per month (about 4.3%), these figures can hardly be seen to reflect accurately the actual figures but can be taken as indicative.

If we define a price ratio (ρ) as `[Open - Close]/[High - Low ]` then this must lie in the range : [-1.0,1.0] .

For a Weiner (Brownian) process the expected value E(ρ) = 0.5

```
# Compute E(ρ) for the Google stocks
julia> ρ = map(x -> (x.open - x.close)/(x.high - x.low), googl)
julia> round(reduce(+,(mean.(ρ)))/length(ρ),digits=4)
0.2758
```

For large datasets (and smaller ones too), summary statistics can be computed by applying efficient algorithms implemented in the OnlineStats module.

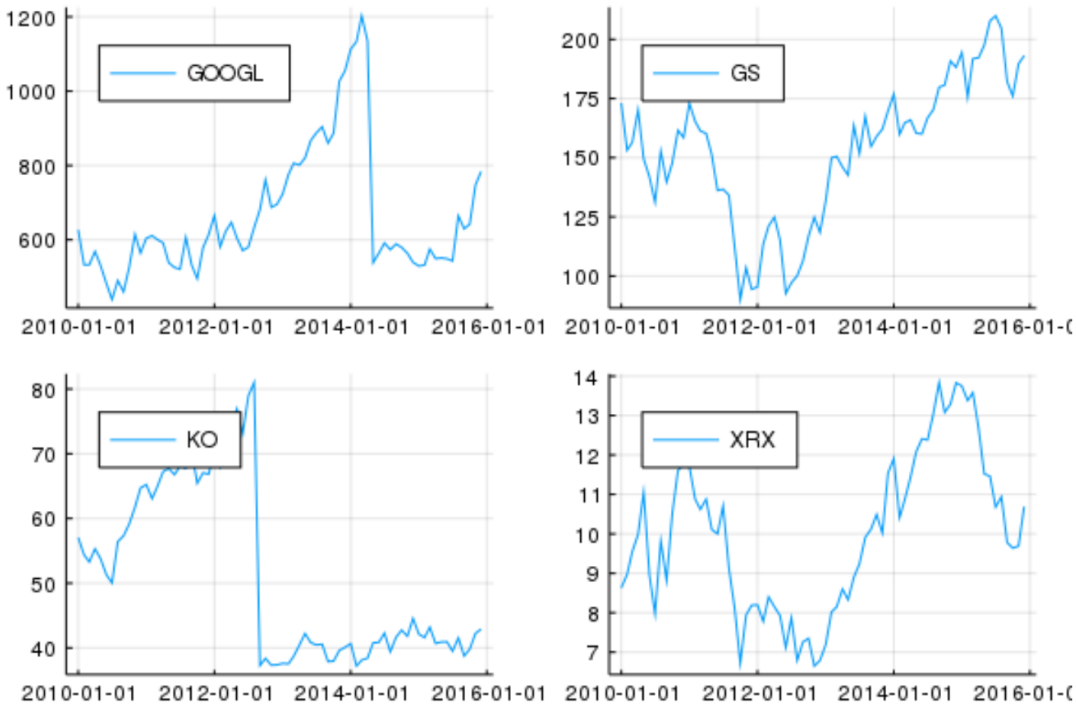
The following computes the mean closing prices grouped by ticker.

```
julia> using OnlineStats
julia> groupreduce(Mean(),stockdata,:ticker; select=:close)
1-d NDSparse with 4 values (Mean{Float64,EqualWeight}):ticker
|──────────┼──────────────────────────────────────────────────"GOOGL" | Mean: n=72
| value=663.283"GS"     | Mean: n=72 | value=152.9"KO"     | Mean: n=72 |
value=51.0185"XRX"    | Mean: n=72 | value=9.99694
```

Finally we can use StatsPlots to create a complex visualisation with a single call.

The `@df` macro is able to refer to columns simply by their name and we can work with these symbols as if they are regular vectors

```
julia> using StatsPlots
julia> @df stockdata plot(:date, :close, group=:ticker,
                          layout = 4, legend = :topleft)
```



Note the couple of pricing adjustments for Coca Cola in 2012 and Google in 2014: these are not due to general market falls since the behaviour of each is in different months and is not reflected in either Goldman Sachs or Rank Xerox.

Summary

This chapter has looked at the means by which Julia interacts with data held in databases and data stores. Until recently, the great majority of databases conformed to the relational model, the so-called SQL database.

However, the rapid explosion in data volumes accompanying the big data revolution has led to the introduction of a range of databases based on other data models. These are normally grouped under the heading NoSQL and are categorized as key-value, document, and graphic databases. With such a large field to cover, we identified some definitive examples in each category. Julia's approaches are largely specific to each individual case, and the appropriate packages and methods for loading, maintaining, and querying the different types of databases have been presented.

Later we will discuss working with various networked systems and look at developing Internet servers, working with web sockets, and messaging via email, SMS, and Twitter. Finally, we will explore the use of the cloud services such as those provided by Amazon and Google.

Index