Databases

Jim Harner

6/7/2019

The RPostgreSQL package provides an R interface to the PostgreSQL relational database. It depends on the `DBI` package, which provides a general interface between R and database management systems.

```r
# The container name housing PostgreSQL is `postgres`.
host.name = "postgres"
library(RPostgreSQL)
```

```
## Loading required package: DBI
```

We will also be using `psql`, which is a UNIX command line interface (CLI) to PostgreSQL.

2.4 Databases

This section introduces relational data base management systems and NoSQL databases. The relational model is essential for multi-user transactional data, but it does not scale for big data. NoSQL databases are often distributed across a cluster.

Two concepts are central to databases, but unfortunately are ambiguous.

- ACID (Atomicity, Consistency, Isolation, and Durability) is central to relational databases, whereas

- CAP (Consistency, Availability, and Partition Tolerance) as well as ACID are important for distributed databases.

2.4.1 RDBMS

The example in this section is based on the `dataexpo` data in Paul Murrell's Introduction to Data Technologies.

A *relational data base management system* (RDBMS) is based on Codd's *relational model* (RM), which in turn is based on *relational algebra*. It uses Structured Query Language (SQL) as a query language.

A single logical operation on a database is called a *transaction*. A single transaction can involve multiple changes, e.g., debiting one account and crediting another when funds are transferred in a bank. To perform these operations safely, certain properties must be met.

RDBMS should maintain the ACID properties:

- Atomicity: transactions are all or nothing;

- Consistency: transactions bring the database from one valid state to another;

- Isolation: concurrent transactions maintain state as if they are serial transactions;

- Durability: a committed transaction maintains state even if there are crashes, power failures, etc.

We will be using PostgreSQL—an open source DBMS that is stable and feature rich. PostgreSQL has a command-line interface for making queries called `psql`. We have several built in databases on our PostgreSQL server—`airlines` and `dataexpo`.

The Data Expo data set consists of seven atmospheric measurements at locations on a 24 by 24 grid averaged over each month for six years (72 time points). The elevation (height above sea level) at each location is also included in the data set.

The table schema for `dataexpo` is defined as follows.

```
date_table ( ID [PK], date, month, year )

location_table ( ID [PK], longitude, latitude, elevation )

measure_table ( date [PK] [FK date_table.ID],
                location [PK] [FK location_table.ID],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )
```

The `dataexpo` database can be invoked using `psql` in RStudio's `bash` shell as follows:

```
psql -h postgres dataexpo
```

The `-w` option causes a prompt for your password, but is not needed for the Dockerized version of this course. `psql` is in `/usr/bin`, which is in the `PATH` environmental variable, i.e., it is not necessary to invoke by `/usr/bin/psql`.

The `-h` option specifies the host, which in this case is `postgres`. It is not needed if the Postgres is on the same machine as RStudio, but in the Dockerized version Postgres is in a separate container called `postgres`.

Databases typically are only setup by the database administrator (DBA). Once established you can populate it with tables if you have write permissions. Tables could be added to the `dataexpo` database by the following command if they are not already there. But don't since the database is populated.

```
# Do not run!
psql -h postgres dataexpo < dataexpo.sql
```

`dataexpo.sql` is in your working directory and it contains code for constructing tables (and their schema) and inserting the data into these tables. The order of creating tables (`CREATE TABLE`) is important since a table must be present before it can be referenced.

If you have not done so, enter interactive mode in a terminal by:

```
psql -h postgres dataexpo
```

Try it in RStudio's shell.

Once in interactive mode, the `psql` commands for listing the tables in the database are `\d` and for specific information about a specific table `\d table`. At the `dataexpo` prompt type:

```
\d

            List of relations
 Schema |      Name      | Type  |  Owner
--------+----------------+-------+---------
 public | date_table     | table | rstudio
 public | location_table | table | rstudio
 public | measure_table  | table | rstudio
(3 rows)

\d date_table

         Table "public.date_table"
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 id     | integer               | not null
 date   | date                  |
 month  | character varying(10) |
 year    | integer                |
Indexes:
    "date_table_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "measure_table" CONSTRAINT "measure_date_table_fk" FOREIGN KEY (date) REFERENCES date_table(i

\q
```

The last command quits `psql`.

To get help use:

- `\h` to list SQL commands;
- `\h command` to show the syntax for `command`;
- `\?` to list psql commands

You can run batch commands in `psql` by putting a SQL `--command` in quotes.

```
psql -h postgres  dataexpo --command "select * from location_table limit 5"
```

```
##  id | longitude | latitude | elevation
## ----+-----------+----------+-----------
##   1 |   -113.75 |    36.25 |   1526.25
##   2 |   -111.25 |    36.25 |   1759.56
##   3 |   -108.75 |    36.25 |   1948.38
##   4 |   -106.25 |    36.25 |   2241.31
##   5 |   -103.75 |    36.25 |   1692.75
## (5 rows)
```

Generally, we will connect to PostgreSQL through the R package `RPostgreSQL`.

Ordinarily, we would use:

```
tryCatch({
  drv <- dbDriver("PostgreSQL")
  con <- dbConnect(drv, dbname='dataexpo')

  dbListConnections(drv)

  dbListTables(con)
  dbListFields(con, "location_table")

# more R code
},
finally = {
  dbDisconnect(con)
  dbUnloadDriver(drv)
})
```

This provides safe coding in case there is a network problem. However, in order to get printed output in the `try` part, we will use regular R code.

```
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, host = "postgres", dbname = 'dataexpo')
dbListConnections(drv)
```

```
## [[1]]
## <PostgreSQLConnection>
```

```r
# list the tables and the fields within the location_table
dbListTables(con)
```

```
## [1] "date_table"     "location_table" "measure_table"
```

```r
dbListFields(con, "location_table")
```

```
## [1] "id"        "longitude" "latitude"  "elevation"
```

We use `dbGetQuery` here to select all columns from the `location_table` and return the results in a data frame.

```r
# dbGetQuery returns a data.frame which can be used directly
meas <- dbGetQuery(con, "select * from location_table")
class(meas)
```

```
## [1] "data.frame"
```

```r
head(meas)
```

```
##   id longitude latitude elevation
## 1  1   -113.75    36.25   1526.25
## 2  2   -111.25    36.25   1759.56
## 3  3   -108.75    36.25   1948.38
## 4  4   -106.25    36.25   2241.31
## 5  5   -103.75    36.25   1692.75
## 6  6   -101.25    36.25    865.19
```

```r
rm(meas)
```

We now consider an alternative approach to select the data from the `location_table`.

```r
# dbSendQuery returns a PostgreSQLResult
measures <- dbSendQuery(con, "select * from location_table")
dbGetStatement(measures)
```

```
## [1] "select * from location_table"
```

```r
# We can then fetch directly from the PostgreSQLResult
fetch(measures, n = 10)
```

```
##    id longitude latitude elevation
## 1   1   -113.75    36.25   1526.25
## 2   2   -111.25    36.25   1759.56
## 3   3   -108.75    36.25   1948.38
## 4   4   -106.25    36.25   2241.31
## 5   5   -103.75    36.25   1692.75
## 6   6   -101.25    36.25    865.19
## 7   7    -98.75    36.25    472.50
## 8   8    -96.25    36.25    231.69
## 9   9    -93.75    36.25    334.38
## 10 10    -91.25    36.25     82.44
```

```r
# The default number of records to retrieve is 500 per fetch
```

We can fetch records 50 at a time until the data is depleted.

```r
while (!dbHasCompleted(measures)) {
  chunk <- fetch(measures, n = 50)
  print(nrow(chunk))
}
```

```
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 50
## [1] 16
```

```
class(measures)
```

```
## [1] "PostgreSQLResult"
## attr(,"package")
## [1] "RPostgreSQL"
```

```
dbClearResult(measures)
```

```
## [1] TRUE
```

```
# n=-1 fetches all the remaining records
# dbFetch(measures, n=-1)
```

In principle, it would be possible to extract data from the tables of interest and use R functions to join as needed. However, this would be far less efficient than selecting directly from the database. The following example illustrates this.

Suppose we want to plot the average temperature (Kelvin) vs. the base elevation. First, we extract `surftemp` and then average and `elevation` grouped by multiples of 500. The required `select` statement involves joins, grouping, etc.

```
temp.avgs <- dbGetQuery(con,
    "select round(l.elevation/500)*500 base_elev, avg(m.surftemp) avg_temp
    from measure_table m
    join location_table l on m.location = l.id
    join date_table d on m.date = d.id
    where d.year = 1998
    group by base_elev
    order by base_elev")
temp.avgs
```

```
##   base_elev avg_temp
## 1         0 297.7069
## 2       500 296.2833
## 3      1000 296.0884
## 4      1500 294.3592
## 5      2000 291.8808
## 6      2500 295.1417
## 7      3500 289.5167
## 8      4000 288.6617
## 9      4500 288.9500
```

```
dbDisconnect(con)
```

```
## [1] TRUE
```

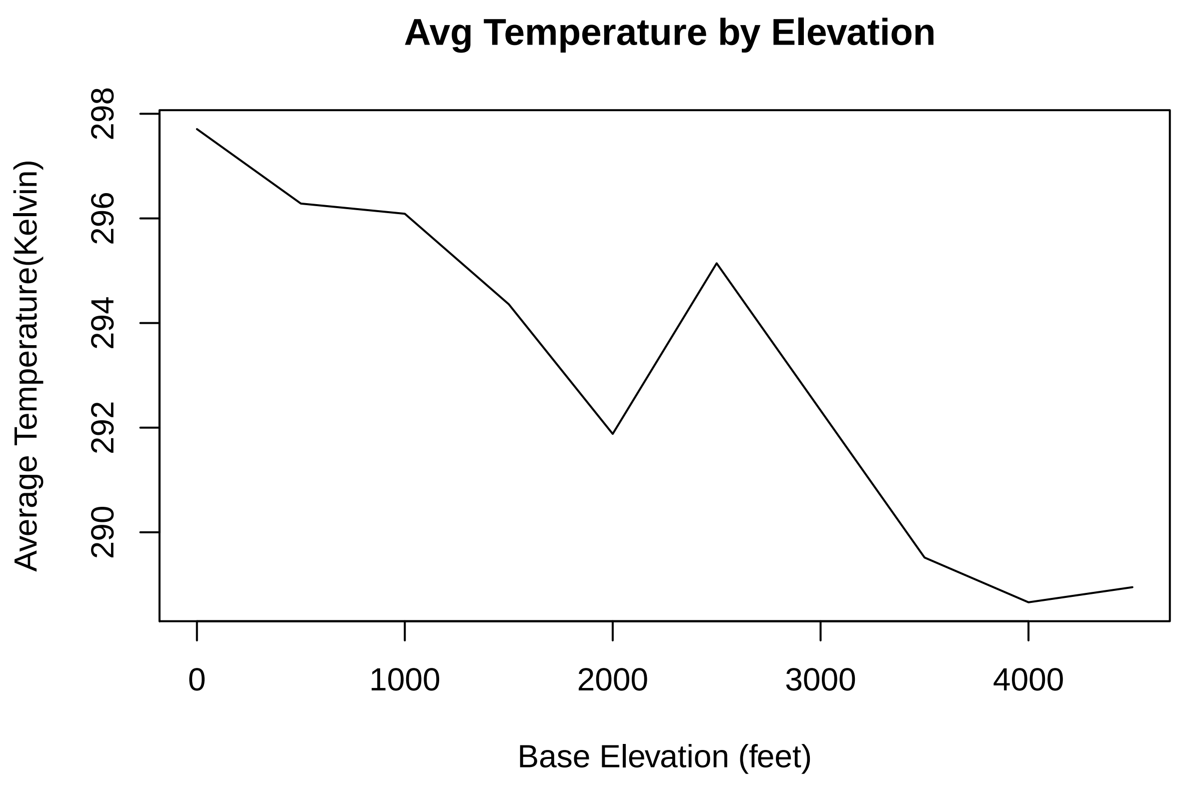
```r
dbUnloadDriver(drv)
```

```
## [1] TRUE
```

I am assuming you have basic knowledge or `select`. We use `dbGetQuery` in order to get a data frame directly—in this case `temp.avgs`.

Now plot the data frame.

```r
plot(temp.avgs, type="l",
  xlab="Base Elevation (feet)", ylab="Average Temperature(Kelvin)",
  main=" Avg Temperature by Elevation")
```



As the base elevation increases, the average temperature tends to decrease as expected.

2.4.2 NoSQL

NoSQL (Not only SQL) databases are widely used when storing big data and real-time web data. NoSQL databases:

- are not based on the relational model;
- perform well on clusters;
- do not have a fixed schema;
- are usually open source;
- are specialized for web applications and big data.

Some NoSQL databases support a SQL-like query language.

Why NoQSL? SQL databases:

- have an impedance mismatch between the relational model and the application model;
- do not run well on clusters.

It would be impossible to run Web 2.0 companies, e.g., Google, Facebook and Twitter, using a RDBMS.

Aggregate Data Models

Modern applications need data grouped into units for ACID purposes. Aggregated data is easy to manage over a cluster. Inter-aggregate relationships are handled via map-reduce operations. Often materialized views are precomputed.

Data Distribution

Two models are used for distributing data across a cluster.

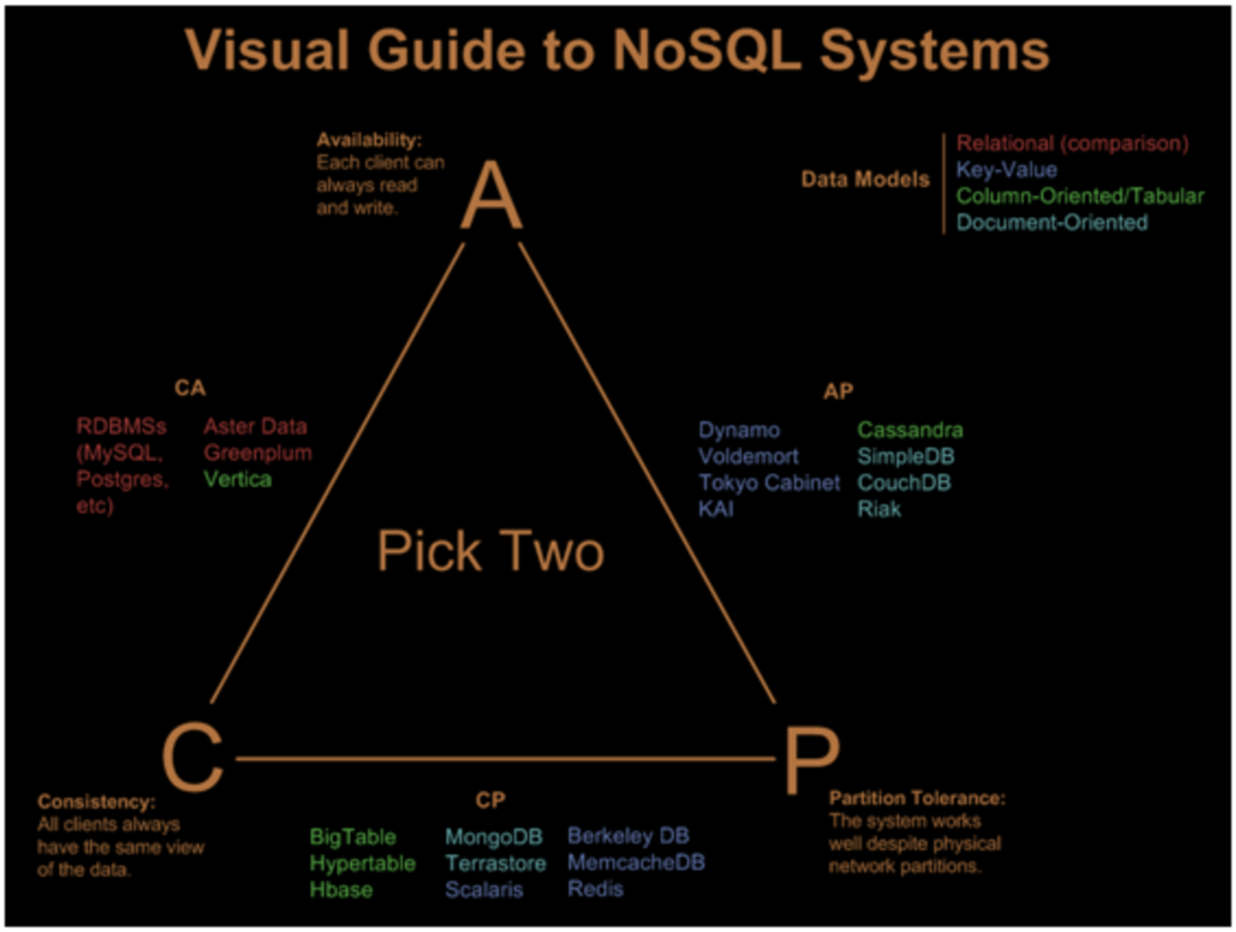
- sharding: segment the data by primary key into shards, each stored on a separate node.
- replication: copy all data to multiple servers either as master-slave or peer-to-peer.

CAP Theorem

The CAP theorem, or Brewer's theorem, states that it is impossible for a distributed database to simultaneously provide all three of the following guarantees:

- Consistency: every read receives the most recent write or an error;
- Availability: every request receives a response, without guarantee that it contains the most recent version of the information;
- Partition tolerance: the system continues to operate despite arbitrary partitioning due to network failures.

Basically, you can choose any two, but cannot have all three. See the following figure:



Note: consistency in CAP is not the same as consistence in ACID.

The following are the possibilities:

- CA: uses a 2-phase commit with a block system (only possible in a single data center);
- CP: uses shards, but there is some risk of data becoming unavailable if a node fails;
- AP: may return inaccurate data, but the system is always available.

The following are the NoSQL database types:

- Key-value
  - Simplest API (get, put, delete, etc.)
  - Data in a blob
  - Not necessarily persistent
- Document
  - Similar to key-value, but with values in a known format
  - Structured data, e.g., JSON, BSON, or XML
  - Not necessarily persistent

- Column-family
  - Many columns associated with each row key
  - Column families related and often accessed together
- Graph
  - Entities and their relationships stored
  - Properties associated with entities
  - Properties and direction significance associated with edges
  - Easy transversal of relationships

Examples of NoSQL databases:

- Key-value: riak, memcached, redis
- Document: CouchDB, MongoDB
- Column-family: cassandra, HBase
- Graph: Neo4J, Infinite Graph

Why choose NoSQL? To improve:

- programmer productivity;
- data access performance by handling larger data sets, reducing latency, and/or improving throughput.

Selecting a NoSQL database:

- Key-value is used for session information, preferences, profiles, and shopping carts, i.e., data without relationships.
- Document databases are for content management, real-time analytics, and ecommerce. Avoid when aggregate data is needed.
- Column family databases are useful for write-heavy operations like logging.
- Graph databases are optimal for social networks, spacial data, and recommendation engines.