14. SQL queries

Principles of Data Science with R

Dr. Uma Ravat

PSTAT 10

We saw...

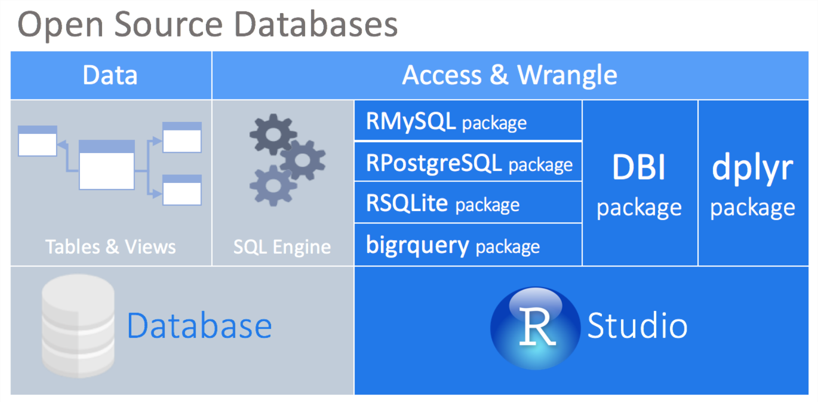
- Databases store massive amounts of data that cannot fit in memory.
- SQL(Structured Query Language) is used to manipulate relational databases
- SQLite is the SQL implementation we will use, provided by the `RSQLite` package.
- Three parts of the relational data model
    1. Manipulative
        - SQL for create, update, delete tables, databases and user access
        - SQL for select, insert, update, delete data in tables
    2. Structural
        - ER diagram, Database schema
        - Primary keys, Foreign keys
    3. Integrity
        - Entity integrity: integrity of each relation
        - Referential integrity : integrity between relations

Next we will see...

Manipulative Part of relational databases

- More SQL queries
- All parts of a SQL query

SQLite RDBMS and Rstudio



Open Source Databases

Data

Access & Wrangle

Tables & Views

SQL Engine

RMySQL package

RPostgreSQL package

RSQLite package

bigrquery package

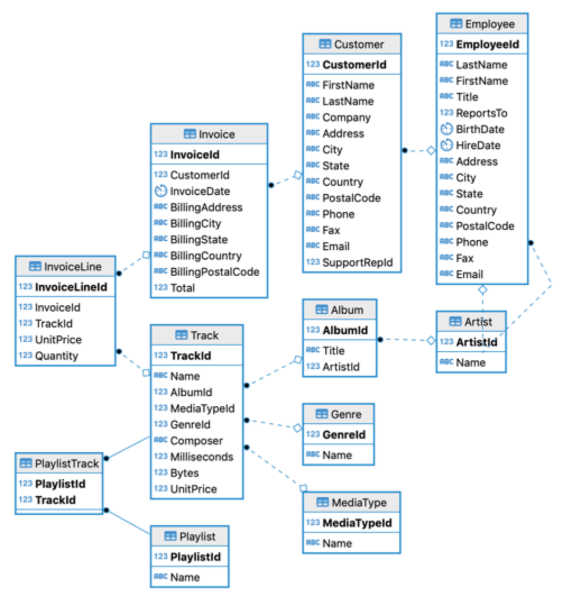
DBI package

dplyr package

Database

R Studio

ChinookDB Entity-Relationship Diagram (ER Diagram)



Connecting to a DB

```r
library(DBI)
library(RSQLite)
drv = dbDriver("SQLite") # the driver for the db you want to connect to
chinook_db = dbConnect(drv, # the driver to use
              dbname="./data/Chinook_Sqlite.sqlite") #path to the db file
```

`chinook_db` is an `R` object that represents a connection to the database file `Chinook_Sqlite.sqlite`

SQL queries

```
SELECT columns
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC | DESC]
  LIMIT offset, count;
```

`WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT` are all optional

We saw `SELECT`, `FROM`, `ORDER BY`, `LIMIT`

More SQL queries

Recall SELECT

```
dbGetQuery(chinook_db,
                        "SELECT count(*) FROM track")
```

```
##   count(*)
## 1     3503
```

What are all the fields for every track?

```
dbListFields(chinook_db, "track")
```

```
## [1] "TrackId" "Name" "AlbumId" "MediaTypeId" "GenreId"
## [6] "Composer" "Milliseconds" "Bytes" "UnitPrice"
```

```
track_sel <- dbGetQuery(chinook_db,
                        "SELECT * FROM track")
str(track_sel)
```

```
## 'data.frame': 3503 obs. of 9 variables:
## $ TrackId : int 1 2 3 4 5 6 7 8 9 10 ...
## $ Name : chr "For Those About To Rock (We Salute You)" "Balls to the Wall"
"Fast As a Shark" "Restless and Wild" ...
## $ AlbumId : int 1 2 3 3 3 1 1 1 1 1 ...
## $ MediaTypeId : int 1 2 2 2 2 1 1 1 1 1 ...
## $ GenreId : int 1 1 1 1 1 1 1 1 1 1 ...
## $ Composer : chr "Angus Young, Malcolm Young, Brian Johnson" NA "F. Baltes,
S. Kaufman, U. Dirkscneider & W. Hoffman" "F. Baltes, R.A. Smith-Diesel, S.
Kaufman, U. Dirkscneider & W. Hoffman" ...
## $ Milliseconds: int 343719 342562 230619 252051 375418 205662 233926 210834
203102 263497 ...
## $ Bytes : int 11170334 5510424 3990994 4331779 6290521 6713451 7636561
6852860 6599424 8611245 ...
```

Suppose we only want the first five records for TrackId, Name, AlbumId, Milliseconds, Bytes, UnitPrice from Track table

```
dbGetQuery(chinook_db,
           "SELECT TrackId, Name, AlbumId, Milliseconds, Bytes, UnitPrice
           FROM  track
           limit 5")
```

```
## TrackId Name AlbumId Milliseconds Bytes
## 1 1 For Those About To Rock (We Salute You) 1 343719 11170334
## 2 2 Balls to the Wall 2 342562 5510424
## 3 3 Fast As a Shark 3 230619 3990994
## 4 4 Restless and Wild 3 252051 4331779
## 5 5 Princess of the Dawn 3 375418 6290521
## UnitPrice
## 1 0.99
## 2 0.99
## 3 0.99
## 4 0.99
## 5 0.99
```

SELECT, expanded

In the first line of SELECT, we can directly specify **computations** that we want performed

```
SELECT columns or computations
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC | DESC]
  LIMIT offset, count;
```

Main tools for computations:

MIN, MAX, COUNT, SUM, AVG or any math formula

Example

To calculate the average Milliseconds, Bytes and Max UnitPrice

```
dbGetQuery(chinook_db,
     "SELECT AVG(Milliseconds), AVG(Bytes), MAX(UnitPrice)
      FROM Track")
```

```
##   AVG(Milliseconds) AVG(Bytes) MAX(UnitPrice)
## 1          393599.2   33510207           1.99
```

To replicate this simple command on an imported data frame:

```
mean(track_sel$Milliseconds, na.rm=TRUE)
```

```
## [1] 393599.2
```

```
mean(track_sel$Bytes, na.rm=TRUE)
```

```
## [1] 33510207
```

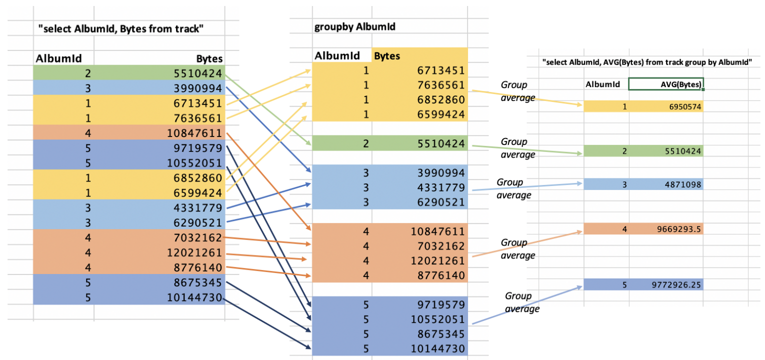
```
max(track_sel$UnitPrice, na.rm=TRUE)
```

```
## [1] 1.99
```

GROUP BY

We can use the GROUP BY option in SELECT to define aggregation groups

The GROUP BY statement groups rows that have the same values and performs aggregation (COUNT(), MAX(), MIN(), SUM(), AVG()) on those groups.



"select AlbumId, Bytes from track"

AlbumId	Bytes
2	5510424
3	3990994
1	6713451
1	7636561
4	10847611
5	9719579
5	10552051
1	6852860
1	6599424
3	4331779
3	6290521
4	7032162
4	12021261
4	8776140
5	8675345
5	10144730

groupby AlbumId

AlbumId	Bytes
1	6713451
1	7636561
1	6852860
1	6599424
2	5510424
3	3990994
3	4331779
3	6290521
4	10847611
4	7032162
4	12021261
4	8776140
5	9719579
5	10552051
5	8675345
5	10144730

"select AlbumId, AVG(Bytes) from track group by AlbumId"

AlbumId	AVG(Bytes)
1	6950574
2	5510424
3	4871098
4	9669293.5
5	9772926.25

```
dbGetQuery(chinook_db, "SELECT AlbumId, AVG(Bytes)
                      FROM Track
                      GROUP BY AlbumId
                      ORDER BY AVG(Bytes) DESC
                      LIMIT 10")
```

```
##    AlbumId AVG(Bytes)
## 1      253  536359244
## 2      229  535292434
## 3      227  529469291
## 4      231  514373372
## 5      228  512231374
## 6      254  492670102
## 7      226  490750393
## 8      261  453454450
## 9      251  306109250
## 10     249  268393262
```

(Note: the order of commands here matters; try switching the order of `GROUP BY` and `ORDER BY`, you'll get an error)

AS

We can use AS in the first line of SELECT to rename computed columns

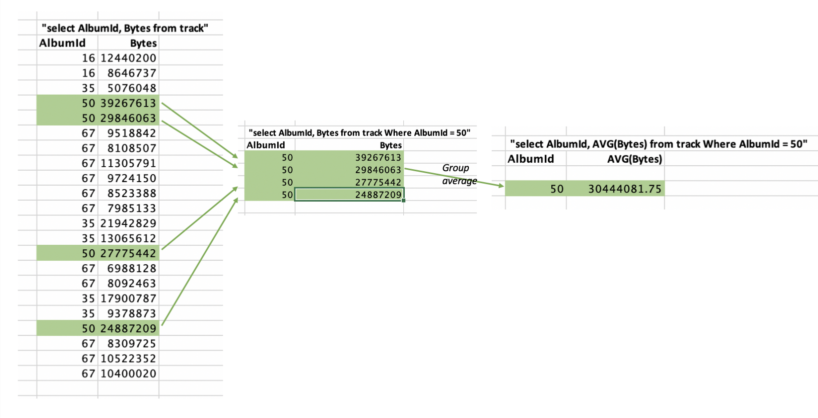
```
dbGetQuery(chinook_db,
           "SELECT AlbumId, AVG(Bytes) AS AvgBytes
            FROM Track
            GROUP BY AlbumId
            ORDER BY AVG(Bytes) DESC
            LIMIT 10")
```

```
##    AlbumId  AvgBytes
## 1      253 536359244
## 2      229 535292434
## 3      227 529469291
## 4      231 514373372
## 5      228 512231374
## 6      254 492670102
## 7      226 490750393
## 8      261 453454450
## 9      251 306109250
## 10     249 268393262
```

count, distinct, case insensitive

```r
# case insensitive
dbGetQuery(chinook_db, "Select count(DISTINCT(ALBUMID)) FROM track")
```

```
##   count(DISTINCT(ALBUMID))
## 1                      347
```



"select AlbumId, Bytes from track"

AlbumId	Bytes
16	12440200
16	8646737
35	5076048
50	39267613
50	29846063
67	9518842
67	8108507
67	11305791
67	9724150
67	8523388
67	7985133
35	21942829
35	13065612
50	27775442
67	6988128
67	8092463
35	17900787
35	9378873
50	24887209
67	8309725
67	10522352
67	10400020

"select AlbumId, Bytes from track Where AlbumId = 50"

AlbumId	Bytes
50	39267613
50	29846063
50	27775442
50	24887209

"select AlbumId, AVG(Bytes) from track Where AlbumId = 50"

AlbumId	AVG(Bytes)
50	30444081.75

WHERE

```r
dbGetQuery(chinook_db,
           "SELECT  AlbumId, Avg(Bytes)
           FROM Track
           WHERE AlbumId = 50")
```

```
##   AlbumId Avg(Bytes)
## 1      50   30444082
```

We can use the `WHERE` option in `SELECT` to specify a subset of the rows to use (*pre-aggregation*/*pre-calculation*)

```
dbGetQuery(chinook_db,
           "SELECT AlbumId, MediaTypeId,AVG(Bytes) as AvgBytes
                      FROM Track
                      WHERE AlbumId <= 160
                      GROUP BY AlbumId
                      ORDER BY AvgBytes DESC
                      LIMIT 10")
```

```
##    AlbumId MediaTypeId AvgBytes
## 1       50           1 30444082
## 2      138           1 24822832
## 3      137           1 19120969
## 4       43           1 16221538
## 5       97           1 16089011
## 6      114           1 15975057
## 7      109           1 15934275
## 8      113           1 15521017
## 9      127           1 15194926
## 10      98           1 14851676
```

Note we used the alias AvgBytes for AVG(BYTES) in the ORDER BY.

HAVING

We can use the HAVING option in SELECT to specify a subset of the rows to display (*post-aggregation/post-calculation*)

```
dbGetQuery(chinook_db,
           "SELECT AlbumId, MediaTypeId,AVG(Bytes) as AvgBytes
            FROM Track
            WHERE AlbumId >= 160
            GROUP BY AlbumId
            HAVING AvgBytes >= 25000000
            ORDER BY AVG(Bytes) DESC
            LIMIT 10")
```

```
##    AlbumId MediaTypeId  AvgBytes
## 1      253           3 536359244
## 2      229           3 535292434
## 3      227           3 529469291
## 4      231           3 514373372
## 5      228           3 512231374
## 6      254           3 492670102
## 7      226           3 490750393
## 8      261           3 453454450
## 9      251           3 306109250
## 10     249           3 268393262
```

What happened?

```
dbGetQuery(chinook_db,
           paste("SELECT AlbumId, MediaTypeId,AVG(Bytes) as AvgBytes",
                      "FROM Track",
                      "WHERE AlbumId >= 160"
           ))
```

```
##   AlbumId MediaTypeId AvgBytes
## 1     160           1 65784686
```

```
dbGetQuery(chinook_db,
           paste("SELECT AlbumId, MediaTypeId,AVG(Bytes) as AvgBytes",
                      "FROM Track",
                      "WHERE AlbumId >= 160",
                      "GROUP BY AlbumId"
                      ))
```

```
##     AlbumId MediaTypeId  AvgBytes
## 1       160           1   5948116
## 2       161           1   7287240
## 3       162           1   8528704
## 4       163           1   6228242
## 5       164           1   6937701
## 6       165           1   7830875
## 7       166           1   7153672
## 8       167           1   7127553
## 9       168           1   6939521
## 10      169           1   7485054
## 11      170           2   4601224
## 12      171           2   5390211
## 13      172           2   5179599
## 14      173           2   5832677
```

What happened?

```
dbGetQuery(chinook_db,
           "SELECT AlbumId, MediaTypeId,AVG(Bytes) as AvgBytes
            FROM Track
            WHERE AlbumId >= 160
            GROUP BY AlbumId
            HAVING AvgBytes >= 25000000
            ORDER BY AVG(Bytes) DESC
            LIMIT 10")
```

```
##    AlbumId MediaTypeId  AvgBytes
## 1      253           3 536359244
## 2      229           3 535292434
## 3      227           3 529469291
## 4      231           3 514373372
## 5      228           3 512231374
## 6      254           3 492670102
## 7      226           3 490750393
## 8      261           3 453454450
## 9      251           3 306109250
## 10     249           3 268393262
```

WHERE with String operators

Pattern Matching on String Operators:

- **LIKE operator:** Allows wildcards to be used
  - % : matches any sequence of zero or more characters
  - _ : matches any single character
- the p% pattern will match any strings that begin with p
  - e.g.: pstat, pineapple, pop
- the %al pattern matches any string that ends with al
  - e.g.: pal, bridal, opal
- the %ul% pattern matches any string that contains ul
  - e.g.: ultimate and forceful
- the pattern r_n will match the strings run, ran, ron
- the pattern __rd matches the strings yard, ward, herd, etc.
  - then what is the difference between __rd and %rd ?
  - %rd would also match longer words ending in rd, like bernard, heard, etc

Like operator

```
dbGetQuery(chinook_db,
            "SELECT Name, Albumid, composer
              FROM track
              WHERE composer LIKE '%Smith%'
              ORDER BY albumid
              LIMIT 3")
```

```
##                    Name AlbumId
## 1    Restless and Wild       3
## 2 Princess of the Dawn       3
## 3        Killing Floor      19
##
## 1 F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. Dirkscneider &
## 2                                             Deaffy & R.A. S
## 3                                                           A
```

IN operator

```r
dbGetQuery(chinook_db,
            "SELECT name, albumid, mediatypeid
             FROM track
             WHERE mediatypeid IN (2, 3)
             LIMIT 5")
```

```
##                    Name AlbumId MediaTypeId
## 1     Balls to the Wall       2           2
## 2       Fast As a Shark       3           2
## 3     Restless and Wild       3           2
## 4  Princess of the Dawn       3           2
## 5 Welcome to the Jungle      90           2
```

LIMIT offset

to get 10 rows starting from the 11th row in the track

```
dbGetQuery(chinook_db,
            "SELECT trackId, name
             FROM track
             LIMIT 10 OFFSET 10")
```

```
##    TrackId                     Name
## 1       11                   C.O.D.
## 2       12       Breaking The Rules
## 3       13 Night Of The Long Knives
## 4       14               Spellbound
## 5       15                  Go Down
## 6       16              Dog Eat Dog
## 7       17        Let There Be Rock
## 8       18           Bad Boy Boogie
## 9       19            Problem Child
## 10      20                 Overdose
```

Disconnecting from the database

After the end of a session, it is good practice to explicitly close your connection.

```
dbDisconnect(chinook_db)
```

```
# Try selecting data
dbGetQuery(chinook_db,
           "select CustomerId, FirstName, LastName from Customer")
```

```
## Error: Invalid or closed connection
```

Does this remove the database connection `chinook_db` in the R session?

We saw

- All parts of a SQL query

```
SELECT columns or computations
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC | DESC]
  LIMIT offset, count;
```

- Database tools for R
  - the R packages `RSQLite`, `DBI`
  - the database `Chinook_Sqlite.sqlite`