14. SQL queries

Principles of Data Science with R

Dr. Uma Ravat

PSTAT 10

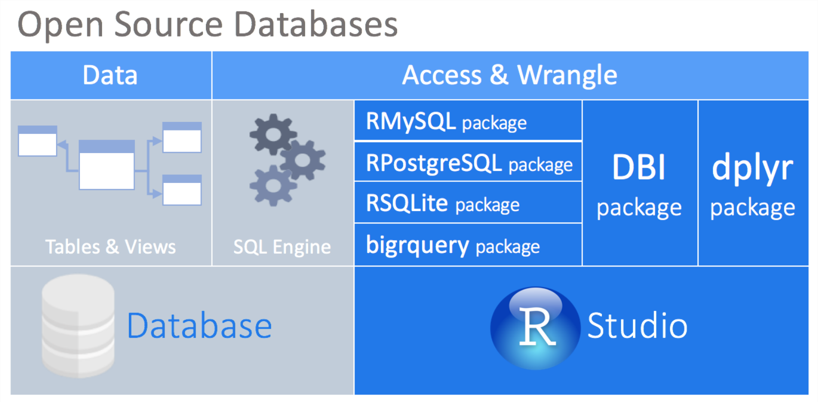
Announcement

1. Quiz 6 this week on Canvas
2. HW will be released tomorrow as usual, due next Wednesday

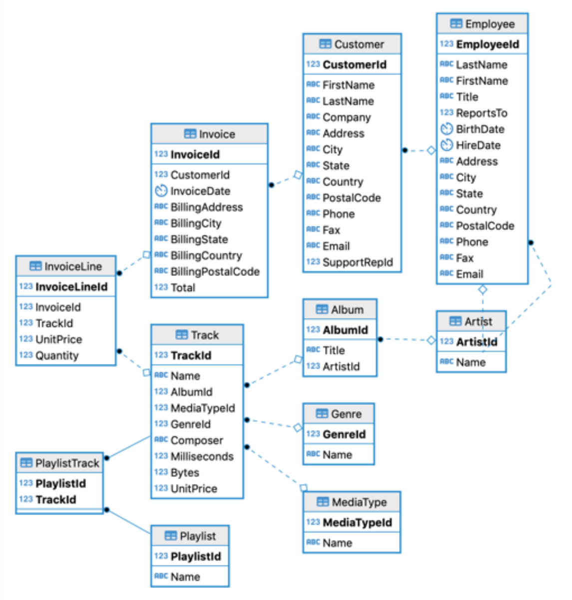
Next we will see...

- Integrity constraints (Primary and Foreign Keys)
- More SQL queries

SQLite RDBMS and Rstudio



ChinookDB Entity-Relationship Diagram (ER Diagram)



Connecting to a DB

- (Install) and load the `DBI` (Database interface) package
- (Install) and load DBI complaint DataBase Connectivity driver package for the database you will be using
  - For SQLite RDBMS, we will use the `SQLite()` driver from the `RSQLite` R package
  - For Postgres RDBMS, use the `Postgres()` driver from the `RPostgres` package
  - ...

```
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`

Field metadata

Unlike a data frame, there is extra information in a database table that expresses relational information between tables.

```
dbGetQuery(chinook_db, "pragma table_info(Customer)")
```

```
##    cid         name         type notnull dflt_value pk
## 1    0   CustomerId      INTEGER       1         NA  1
## 2    1    FirstName NVARCHAR(40)       1         NA  0
## 3    2     LastName NVARCHAR(20)       1         NA  0
## 4    3      Company NVARCHAR(80)       0         NA  0
## 5    4      Address NVARCHAR(70)       0         NA  0
## 6    5         City NVARCHAR(40)       0         NA  0
## 7    6        State NVARCHAR(40)       0         NA  0
## 8    7      Country NVARCHAR(40)       0         NA  0
## 9    8   PostalCode NVARCHAR(10)       0         NA  0
## 10   9        Phone NVARCHAR(24)       0         NA  0
## 11  10          Fax NVARCHAR(24)       0         NA  0
## 12  11        Email NVARCHAR(60)       1         NA  0
## 13  12 SupportRepId      INTEGER       0         NA  0
```

Primary key

The **primary key** is a *unique identifier* of the rows in a table.

- Two rows cannot have the same primary key

```
# paste() is used so that the SQL statement is easy to read
dbGetQuery(chinook_db,
           paste(
       "SELECT  CustomerId, FirstName, LastName, City, Country",
       "FROM Customer",
       "LIMIT 2"
                )
          )
```

```
##   CustomerId FirstName  LastName                City Country
## 1          1      Luís Gonçalves São José dos Campos  Brazil
## 2          2    Leonie    Köhler           Stuttgart Germany
```

```
# No need for paste though
dbGetQuery(chinook_db,
       "SELECT CustomerId, FirstName, LastName, City, Country
       FROM Customer
       LIMIT 2")
```

```
##   CustomerId FirstName  LastName                City Country
## 1          1      Luís Gonçalves São José dos Campos  Brazil
## 2          2    Leonie    Köhler           Stuttgart Germany
```

```r
dbExecute(chinook_db,
    paste("INSERT into Customer",
        "(CustomerId, FirstName, LastName, Email)",
        "VALUES",
        "(1, 'Luis','Armstrong','LuisArmstrong@pstat.ucsb.edu')"
        )
    )
```

```
## Error: UNIQUE constraint failed: Customer.CustomerId
```

CustomerId is the **primary key** and must be unique.

Multi-column primary key

Primary key's can consist of multiple columns if it takes multiple columns to identify a row in a table. But, two rows cannot have the same primary key.

```
# Single column primary key
dbGetQuery(chinook_db, "pragma table_info(Customer)")
```

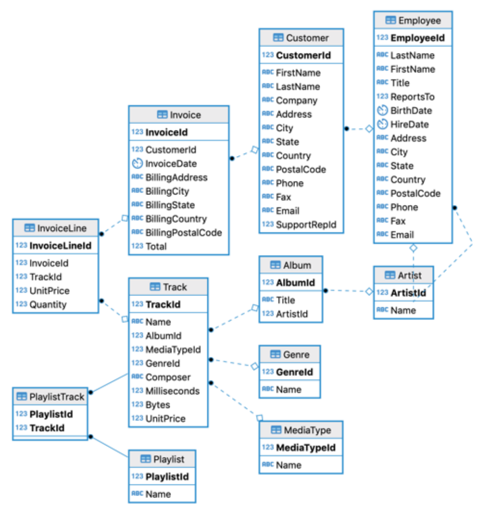
```
##    cid         name         type notnull dflt_value pk
## 1    0   CustomerId      INTEGER       1         NA  1
## 2    1    FirstName NVARCHAR(40)       1         NA  0
## 3    2     LastName NVARCHAR(20)       1         NA  0
## 4    3      Company NVARCHAR(80)       0         NA  0
## 5    4      Address NVARCHAR(70)       0         NA  0
## 6    5         City NVARCHAR(40)       0         NA  0
## 7    6        State NVARCHAR(40)       0         NA  0
## 8    7      Country NVARCHAR(40)       0         NA  0
## 9    8   PostalCode NVARCHAR(10)       0         NA  0
## 10   9        Phone NVARCHAR(24)       0         NA  0
## 11  10          Fax NVARCHAR(24)       0         NA  0
## 12  11        Email NVARCHAR(60)       1         NA  0
## 13  12 SupportRepId      INTEGER       0         NA  0
```

```
# Multi column primary key
dbGetQuery(chinook_db, "pragma table_info(PlayListTrack)")
```

```
##   cid       name    type notnull dflt_value pk
## 1   0 PlaylistId INTEGER       1         NA  1
## 2   1    TrackId INTEGER       1         NA  2
```

Primary key

Tables are not required to have a primary key, but most do. All the tables in Chinook have a primary key.



Foreign keys

The relationship between tables is expressed by primary keys and **foreign keys**.

Remember we are working with a relational database, following a relational data model.

```
dbGetQuery(chinook_db,
           "pragma foreign_key_list(Customer)")
```

```
##   id seq    table        from         to on_update on_delete match
## 1  0   0 Employee SupportRepId EmployeeId NO ACTION NO ACTION  NONE
```

A foreign key field *points to* the primary key of another table.

Foreign keys

Foreign keys must either point to an existing value or be NULL.

To enforce Foreign key constraints in SQLite RDBMS

```
# Required for foreign-key support otherwise foreign keys are not enforced
dbExecute(chinook_db, "pragma foreign_keys = on")
```

```
dbGetQuery(chinook_db,
           "SELECT max(EmployeeId) FROM Employee")
```

```
##   max(EmployeeId)
## 1               8
```

```
dbGetQuery(chinook_db,
           "SELECT max(CustomerId) FROM Customer")
```

```
##   max(CustomerId)
## 1              59
```

```
dbExecute(chinook_db,
    "INSERT INTO Customer
    (CustomerId, FirstName, LastName, Email, SupportRepId)
    VALUES
    (59, 'Luis', 'Armstrong', 'luisArmstrong@pstat.ucsb.edu', 88)")
```

```
## Error: UNIQUE constraint failed: Customer.CustomerId
```

```r
dbExecute(chinook_db,
    "INSERT INTO Customer
    (CustomerId, FirstName, LastName, Email, SupportRepId)
    VALUES
    (60, 'Luis', 'Armstrong', 'luisArmstrong@pstat.ucsb.edu', 10)")
```

```
## Error: FOREIGN KEY constraint failed
```

Interpretation of foreign key

- Each customer in Customer table *can* be assigned a support representative
- The support rep is an employee at the store and therefore has a unique id, EmployeeId
- This unique id,EmployeeId , is the primary key of the employee table

Thus real-world relationship is encoded by the relational model using primary and foreign key relationships.

Integrity Constraints

We have seen two examples of *integrity constraints*:

- Primary keys must be unique (and not NULL)
- Foreign keys must reference existing primary keys or be NULL

These constraints enforce the *integrity* of a database: no bad data or corrupted relationships.

Keys help maintain the integrity of the data

Database Schema

The **schema** of a database describes its *structure*:

- Names of all the tables
- Names of all fields in each table
- Primary key/foreign key relationships between tables
- Other metadata (data types of each field in each table, ...)

Basically everything other than the actual data itself.

Represented via E-R diagrams (Entity relationship)

We have been looking at parts of the schema with the `pragma` keyword.

```
dbGetQuery(chinook_db, "pragma table_info(customer)")
```

More SQL queries

SELECT

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

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

What are all the fields for every track?

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

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

```r
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,
      paste("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

```
dbGetQuery(chinook_db, paste("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,
           paste("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
```

WHERE

```
dbGetQuery(chinook_db,
           paste("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,
           paste("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
```

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,
           paste("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
```

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

- Integrity constraints (Primary and Foreign Keys)
- 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`