Database and SQL

02-11-2020

```
library(tidyverse)
library(DBI)
```

What is a database? It is what google says

> a structured set of data held in a computer, especially one that is accessible in various ways.

A relational database is a type of database that stores and provides access to data points that are related to one another. Relation databases are administrated by a Relational Database Management System (RDBMS). The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

There are many RDBMS - MySQL (owned by Oracle) - PostgreSQL (open source) - SQL Server (microsoft) - SQLite (open source, single file)

What is SQL? Structured Query Language (or SQL) is a standard language for accessing and manipulating relational databaes. However, each RDMBS may have their own extension of the SQL language and their implmentation may vary too.

Connect to a databse

We are going to use a popular database called Sakila https://www.jooq.org/sakila.

The Sakila database is a nicely normalised schema modelling a DVD rental store, featuring things like films, actors, film-actor relationships, and a central inventory table that connects films, stores, and rentals.

In the following, we are going to use both `sqlite` and `postgresql`.

SQLite

The database is called `sakila.sqlite`. You could either git clone from lectures repo or download using the code

```
if (!file.exists("sakila.sqlite") || file.size("sakila.sqlite") == 0) {
  download.file(
    "https://github.com/UCDavis-STA-141B-Winter-2020/sta141b-lectures/raw/master/02-11/sakila.sqlite",
    destfile = "sakila.sqlite")
}
```

The file format is `.sqlite` which is one of the very common relational database formats, espeically for simple problems.

```
sakila_lite <- dbConnect(RSQLite::SQLite(), dbname = "sakila.sqlite")
sakila_lite %>% dbListTables()
```

```
##  [1] "actor"                "address"              "category"
##  [4] "city"                 "country"              "customer"
##  [7] "customer_list"        "film"                 "film_actor"
## [10] "film_category"        "film_list"            "film_text"
## [13] "inventory"            "language"             "payment"
## [16] "rental"               "sales_by_film_category" "sales_by_store"
## [19] "sqlite_sequence"      "staff"                "staff_list"
## [22] "store"
```

Postgresql

I have also uploaded the Sakila database to a postgres server owned by the department. (You'll need to either on the campus or over UCD vpn to connect to it)

```
sakila_psql <- dbConnect(RPostgres::Postgres(),
  dbname = "sakila",
  user = "psqluser", password = "secret", host = "alan.ucdavis.edu"
)
sakila_psql %>% dbListTables()
```

How not to use SQL?

`dplyr` provides an excellent interface for users without any SQL background to query databases.

```
# number of rental transactions
sakila_lite %>%
  tbl("rental") %>%
  count() %>%
  collect()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1 16044
```

`sakila_lite %>% tbl("rental")` creates a virtual table rather loading the whole table into memory.

```
sakila_lite %>%
  tbl("rental") %>%
  class()
```

```
## [1] "tbl_SQLiteConnection" "tbl_dbi"              "tbl_sql"
## [4] "tbl_lazy"             "tbl"
```

```
sakila_lite %>%
  tbl("rental") %>%
  colnames()
```

```
## [1] "rental_id"    "rental_date"  "inventory_id" "customer_id"  "return_date"
## [6] "staff_id"     "last_update"
```

Sakila queries

https://datamastery.gitlab.io/exercises/sakila-queries.html

- Which actors have the first name Scarlett?

```
sakila_lite %>%
  tbl("actor") %>%
  filter(str_to_lower(first_name) == str_to_lower("Scarlett")) %>%
  collect()
```

```
## # A tibble: 2 x 4
##   actor_id first_name last_name last_update
##      <int> <chr>      <chr>     <chr>
## 1       81 SCARLETT   DAMON     2019-04-11 18:11:48
## 2      124 SCARLETT   BENING    2019-04-11 18:11:48
```

Suppose we want to make the result a bit more beautiful.

```
sakila_lite %>%
  tbl("actor") %>%
  filter(str_to_lower(first_name) == str_to_lower("Scarlett")) %>%
  collect() %>%
  mutate(first_name = str_to_title(first_name), last_name = str_to_title(last_name))
```

```
## # A tibble: 2 x 4
##   actor_id first_name last_name last_update
##      <int> <chr>      <chr>     <chr>
## 1       81 Scarlett   Damon     2019-04-11 18:11:48
## 2      124 Scarlett   Bening    2019-04-11 18:11:48
```

Note: SQLite doesn't support transforming title case but Postgresql does.

```
sakila_psql %>%
  tbl("actor") %>%
  filter(str_to_lower(first_name) == str_to_lower("Scarlett")) %>%
  mutate(first_name = str_to_title(first_name), last_name = str_to_title(last_name)) %>%
  collect()
```

- Which actors have the last name Johansson?

```
sakila_lite %>%
  tbl("actor") %>%
  filter(str_to_lower(last_name) == "johansson") %>%
  collect()
```

```
## # A tibble: 3 x 4
##   actor_id first_name last_name last_update
##      <int> <chr>      <chr>     <chr>
## 1        8 MATTHEW    JOHANSSON 2019-04-11 18:11:48
## 2       64 RAY        JOHANSSON 2019-04-11 18:11:48
## 3      146 ALBERT     JOHANSSON 2019-04-11 18:11:48
```

- How many distinct actors last names are there?

```
sakila_lite %>%
  tbl("actor") %>%
  summarize(n = n_distinct(last_name)) %>%
  collect()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   121
```

- Which last names are not repeated?

```
sakila_lite %>%
  tbl("actor") %>%
  count(last_name) %>%
  filter(n == 1) %>%
  collect()
```

```
## # A tibble: 66 x 2
##    last_name     n
##    <chr>     <int>
##  1 ASTAIRE       1
##  2 BACALL        1
##  3 BALE          1
##  4 BALL          1
##  5 BARRYMORE     1
##  6 BASINGER      1
##  7 BERGEN        1
##  8 BERGMAN       1
##  9 BIRCH         1
## 10 BLOOM         1
## # ... with 56 more rows
```

- Which last names appear more than once?

```
sakila_lite %>%
  tbl("actor") %>%
  count(last_name) %>%
  filter(n > 1) %>%
  collect()
```

```
## # A tibble: 55 x 2
##    last_name     n
##    <chr>     <int>
##  1 AKROYD        3
##  2 ALLEN         3
##  3 BAILEY        2
##  4 BENING        2
##  5 BERRY         3
```

```
##  6 BOLGER        2
##  7 BRODY         2
##  8 CAGE          2
##  9 CHASE         2
## 10 CRAWFORD      2
## # ... with 45 more rows
```

- Which actor has appeared in the most films?

```
sakila_lite %>%
  tbl("film_actor") %>%
  count(actor_id) %>%
  arrange(desc(n)) %>%
  head(1) %>%
  inner_join(tbl(sakila_lite, "actor"), by = "actor_id") %>%
  collect()
```

```
## # A tibble: 1 x 5
##   actor_id     n first_name last_name last_update
##      <int> <int> <chr>      <chr>     <chr>
## 1      107    42 GINA       DEGENERES 2019-04-11 18:11:48
```

- What is that average running time of all the films in the sakila DB?

```
sakila_lite %>%
  tbl("film") %>%
  summarize(m = mean(length)) %>%
  collect()
```

```
## Warning: Missing values are always removed in SQL.
## Use `mean(x, na.rm = TRUE)` to silence this warning
## This warning is displayed only once per session.
```

```
## # A tibble: 1 x 1
##       m
##   <dbl>
## 1  115.
```

- What is the average running time of films by category?

```
sakila_lite %>%
  tbl("film") %>%
  left_join(tbl(sakila_lite, "film_category"), by = "film_id") %>%
  group_by(category_id) %>%
  summarize(mean_length = mean(length)) %>%
  left_join(tbl(sakila_lite, "category"), by = "category_id") %>%
  select(name, mean_length) %>%
  collect()
```

```
## # A tibble: 16 x 2
##    name        mean_length
```

```
##    <chr>             <dbl>
##  1 Action             112.
##  2 Animation          111.
##  3 Children           110.
##  4 Classics           112.
##  5 Comedy             116.
##  6 Documentary        109.
##  7 Drama              121.
##  8 Family             115.
##  9 Foreign            122.
## 10 Games              128.
## 11 Horror             112.
## 12 Music              114.
## 13 New                111.
## 14 Sci-Fi             108.
## 15 Sports             128.
## 16 Travel             113.
```

- Is 'Unforgiven Zoolander' available for rent from Store 1?

```
uz <- sakila_lite %>%
  tbl("film") %>%
  filter(str_to_lower(title) == str_to_lower("Unforgiven Zoolander")) %>%
  select(film_id)
all_inventories_of_store1 <- sakila_lite %>%
  tbl("inventory") %>%
  filter(store_id == 1) %>%
  select(film_id, inventory_id, store_id)
not_yet_returned <- sakila_lite %>%
  tbl("rental") %>%
  filter(is.na(return_date)) %>%
  select(inventory_id)
uz %>%
  inner_join(all_inventories_of_store1, by = "film_id") %>%
  anti_join(not_yet_returned) %>%
  count() %>%
  collect()
```

```
## Joining, by = "inventory_id"
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     2
```

SQL

We just see some example queries of a relational database. Beind the scene, we are using a language called SQL. For example, in the last query, the SQL used is

```
uz %>%
  left_join(all_inventories_of_store1) %>%
  anti_join(not_yet_returned) %>%
  count() %>%
  show_query()
```

```
## Joining, by = "film_id"
```

```
## Joining, by = "inventory_id"
```

```
## <SQL>
## SELECT COUNT() AS `n`
## FROM (SELECT * FROM (SELECT `LHS`.`film_id` AS `film_id`, `RHS`.`inventory_id` AS `inventory_id`, `RI
## FROM (SELECT `film_id`
## FROM `film`
## WHERE (LOWER(`title`) = LOWER('Unforgiven Zoolander'))) AS `LHS`
## LEFT JOIN (SELECT `film_id`, `inventory_id`, `store_id`
## FROM `inventory`
## WHERE (`store_id` = 1.0)) AS `RHS`
## ON (`LHS`.`film_id` = `RHS`.`film_id`)
## ) AS `LHS`
## WHERE NOT EXISTS (
##   SELECT 1 FROM (SELECT `inventory_id`
## FROM `rental`
## WHERE (((`return_date`) IS NULL))) AS `RHS`
##   WHERE (`LHS`.`inventory_id` = `RHS`.`inventory_id`)
## ))
```

Why learning SQL when there is dplyr?

- SQL is everywhere (used in python, php, etc..)
- dplyr magics only read, desnot write
- Job interviews

In R, a sql query can be made by using `dbGetQuery`

```
sakila_lite %>%
  dbGetQuery("SELECT COUNT() AS `n` FROM `rental`")
```

```
##       n
## 1 16044
```

We could also make SQL query by sql block. In here, we are using the connection `sakila_lite`. The result will be printed directly.

```
SELECT COUNT() AS `n` FROM `rental`;
```

Table 1: 1 records

n
16044

In we need the output, set `output.var` to `rental_count`

```
SELECT COUNT() AS `n` FROM `rental`;
```

The output could be later used in R blocks

```
rental_count
```

```
##       n
## 1 16044
```

For comparsion, in Python, we use

```
import sqlite3
sakila_lite = sqlite3.connect('sakila.sqlite')
c = sakila.cursor()
c.execute("SELECT COUNT() AS `n` FROM `rental`")
c.fetchall()
```

SQLite supports both double quotes (which is the standard) and backticks to quote identifiers. Backticks are used in another popular database MySQL. Double quotes are used in Postgresql. It is always a good practice to quote the identifers.

```
-- MySQL style
SELECT COUNT() AS `n` from `actor`;
```

```
-- The standard way
SELECT COUNT(*) AS "n" from "actor";
```

SELECT

The SELECT statement is pretty much the `select()` function in `dplyr`.

```
SELECT "last_name" FROM "actor";
```

Table 2: Displaying records 1 - 10

last_name
AKROYD
AKROYD
AKROYD
ALLEN
ALLEN
ALLEN
ASTAIRE
BACALL
BAILEY
BAILEY

```
SELECT LOWER("last_name") AS "family_name" FROM "actor";
```

Table 3: Displaying records 1 - 10

family_name
akroyd
akroyd
akroyd
allen
allen
allen
astaire
bacall
bailey
bailey

For comparision,

```
sakila_lite %>%
  tbl("actor") %>%
  transmute(family_name = str_to_lower(last_name)) %>%
  collect()
```

```
## # A tibble: 200 x 1
##    family_name
##    <chr>
##  1 akroyd
##  2 akroyd
##  3 akroyd
##  4 allen
##  5 allen
##  6 allen
##  7 astaire
##  8 bacall
##  9 bailey
## 10 bailey
## # ... with 190 more rows
```

```
SELECT * FROM "actor";
```

Table 4: Displaying records 1 - 10

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINESS	2019-04-11 18:11:48
2	NICK	WAHLBERG	2019-04-11 18:11:48
3	ED	CHASE	2019-04-11 18:11:48
4	JENNIFER	DAVIS	2019-04-11 18:11:48
5	JOHNNY	LOLLOBRIGIDA	2019-04-11 18:11:48
6	BETTE	NICHOLSON	2019-04-11 18:11:48
7	GRACE	MOSTEL	2019-04-11 18:11:48

actor_id	first_name	last_name	last_update
8	MATTHEW	JOHANSSON	2019-04-11 18:11:48
9	JOE	SWANK	2019-04-11 18:11:48
10	CHRISTIAN	GABLE	2019-04-11 18:11:48

```
SELECT "rental_id", "last_update" FROM "rental";
```

Table 5: Displaying records 1 - 10

rental_id	last_update
1	2019-04-11 18:11:49
2	2019-04-11 18:11:49
3	2019-04-11 18:11:49
4	2019-04-11 18:11:49
5	2019-04-11 18:11:49
6	2019-04-11 18:11:49
7	2019-04-11 18:11:49
8	2019-04-11 18:11:49
9	2019-04-11 18:11:49
10	2019-04-11 18:11:49

ORDER BY Clause

It is equivalent to `arrange()` in `dplyr`

```
SELECT * FROM "actor" ORDER BY "last_name";
```

Table 6: Displaying records 1 - 10

actor_id	first_name	last_name	last_update
58	CHRISTIAN	AKROYD	2019-04-11 18:11:48
92	KIRSTEN	AKROYD	2019-04-11 18:11:48
182	DEBBIE	AKROYD	2019-04-11 18:11:48
118	CUBA	ALLEN	2019-04-11 18:11:48
145	KIM	ALLEN	2019-04-11 18:11:48
194	MERYL	ALLEN	2019-04-11 18:11:48
76	ANGELINA	ASTAIRE	2019-04-11 18:11:48
112	RUSSELL	BACALL	2019-04-11 18:11:48
67	JESSICA	BAILEY	2019-04-11 18:11:48
190	AUDREY	BAILEY	2019-04-11 18:11:48

```
SELECT * FROM "actor" ORDER BY "last_name" DESC;
```

Table 7: Displaying records 1 - 10

actor_id	first_name	last_name	last_update
186	JULIA	ZELLWEGER	2019-04-11 18:11:48
111	CAMERON	ZELLWEGER	2019-04-11 18:11:48
85	MINNIE	ZELLWEGER	2019-04-11 18:11:48
63	CAMERON	WRAY	2019-04-11 18:11:48
156	FAY	WOOD	2019-04-11 18:11:48
13	UMA	WOOD	2019-04-11 18:11:48
144	ANGELA	WITHERSPOON	2019-04-11 18:11:48
147	FAY	WINSLET	2019-04-11 18:11:48
68	RIP	WINSLET	2019-04-11 18:11:48
168	WILL	WILSON	2019-04-11 18:11:48

DISTINCT

DISTINCT operator to remove duplicates from a result set. It is equivalent to `distinct()` function in `dplyr`.

```
SELECT DISTINCT "last_name" FROM "actor";
```

Table 8: Displaying records 1 - 10

last_name
AKROYD
ALLEN
ASTAIRE
BACALL
BAILEY
BALE
BALL
BARRYMORE
BASINGER
BENING

```
sakila_lite %>%
  tbl("actor") %>%
  distinct(last_name)
```

```
## # Source:   lazy query [?? x 1]
## # Database: sqlite 3.30.1
## #   [/Users/Randy/Dropbox/Winter2020/STA141B/sta141b-lectures/02-11/sakila.sqlite]
##    last_name
##    <chr>
##  1 AKROYD
##  2 ALLEN
##  3 ASTAIRE
##  4 BACALL
##  5 BAILEY
##  6 BALE
##  7 BALL
```

```
##  8 BARRYMORE
##  9 BASINGER
## 10 BENING
## # ... with more rows
```

LIMIT

```
SELECT * FROM "actor" LIMIT 2;
```

Table 9: 2 records

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINESS	2019-04-11 18:11:48
2	NICK	WAHLBERG	2019-04-11 18:11:48

```
sakila_lite %>%
  tbl("actor") %>%
  head(2)
```

```
## # Source:   lazy query [?? x 4]
## # Database: sqlite 3.30.1
## #   [/Users/Randy/Dropbox/Winter2020/STA141B/sta141b-lectures/02-11/sakila.sqlite]
##   actor_id first_name last_name last_update
##      <int> <chr>      <chr>     <chr>
## 1        1 PENELOPE   GUINESS   2019-04-11 18:11:48
## 2        2 NICK       WAHLBERG  2019-04-11 18:11:48
```

WHERE

It is equivalent to `filter()` in `dplyr`.

- SQLite (and MySQL) allows double qoutes to quote string values but it is actually not the SQL standard. In SQL standard, strings are quoted in single quotes.
- In SQL standard, we should use `=` for comparison, but not `==`.

```
sakila_lite %>% tbl("film") %>% distinct(rating)
```

```
## # Source:   lazy query [?? x 1]
## # Database: sqlite 3.30.1
## #   [/Users/Randy/Dropbox/Winter2020/STA141B/sta141b-lectures/02-11/sakila.sqlite]
##   rating
##   <chr>
## 1 PG
## 2 G
## 3 NC-17
## 4 PG-13
## 5 R
```

```sql
SELECT * FROM "film" WHERE "rating" = 'PG' AND "length" = 90;
```

film_id	title	description
776	SECRET GROUNDHOG	A Astounding Story of a Cat And a Database Administrator who must Build a Techn

```sql
sakila_lite %>%
  tbl("film") %>%
  filter(rating == "PG" & length > 90) %>%
  collect()
```

```
## # A tibble: 131 x 13
##    film_id title description release_year language_id original_langua~
##      <int> <chr> <chr>       <chr>               <int>            <int>
##  1       6 AGEN~ A Intrepid~ 2006                    1               NA
##  2      12 ALAS~ A Fanciful~ 2006                    1               NA
##  3      13 ALI ~ A Action-P~ 2006                    1               NA
##  4      19 AMAD~ A Emotiona~ 2006                    1               NA
##  5      37 ARIZ~ A Brillian~ 2006                    1               NA
##  6      41 ARSE~ A Fanciful~ 2006                    1               NA
##  7      65 BEHA~ A Unbeliev~ 2006                    1               NA
##  8      72 BILL~ A Stunning~ 2006                    1               NA
##  9      74 BIRC~ A Fanciful~ 2006                    1               NA
## 10      84 BOIL~ A Awe-Insp~ 2006                    1               NA
## # ... with 121 more rows, and 7 more variables: rental_duration <int>,
## #   rental_rate <dbl>, length <int>, replacement_cost <dbl>, rating <chr>,
## #   special_features <chr>, last_update <chr>
```

- The `IN` operator

```sql
SELECT * FROM "film" WHERE "rating" IN ('PG', 'PG-13');
```

film_id	title	description
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The
6	AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler in An
7	AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Jet Boa
9	ALABAMA DEVIL	A Thoughtful Panorama of a Database Administrator And a Mad Scientist who must
12	ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef who must Vanquish a Boy in Austral
13	ALI FOREVER	A Action-Packed Drama of a Dentist And a Crocodile who must Battle a Feminist in
18	ALTER VICTORY	A Thoughtful Drama of a Composer And a Feminist who must Meet a Secret Agent i
19	AMADEUS HOLY	A Emotional Display of a Pioneer And a Technical Writer who must Battle a Man in
28	ANTHEM LUKE	A Touching Panorama of a Waitress And a Woman who must Outrace a Dog in An A
33	APOLLO TEEN	A Action-Packed Reflection of a Crocodile And a Explorer who must Find a Sumo W

```
sakila_lite %>%
  tbl("film") %>%
  filter(rating %in% c("PG", "PG-13"))
```

```
## # Source:   lazy query [?? x 13]
## # Database: sqlite 3.30.1
## #   [/Users/Randy/Dropbox/Winter2020/STA141B/sta141b-lectures/02-11/sakila.sqlite]
##    film_id title description release_year language_id original_langua~
##      <int> <chr> <chr>       <chr>              <int>            <int>
##  1       1 ACAD~ A Epic Dra~ 2006                   1               NA
##  2       6 AGEN~ A Intrepid~ 2006                   1               NA
##  3       7 AIRP~ A Touching~ 2006                   1               NA
##  4       9 ALAB~ A Thoughtf~ 2006                   1               NA
##  5      12 ALAS~ A Fanciful~ 2006                   1               NA
##  6      13 ALI ~ A Action-P~ 2006                   1               NA
##  7      18 ALTE~ A Thoughtf~ 2006                   1               NA
##  8      19 AMAD~ A Emotiona~ 2006                   1               NA
##  9      28 ANTH~ A Touching~ 2006                   1               NA
## 10      33 APOL~ A Action-P~ 2006                   1               NA
## # ... with more rows, and 7 more variables: rental_duration <int>,
## #   rental_rate <dbl>, length <int>, replacement_cost <dbl>, rating <chr>,
## #   special_features <chr>, last_update <chr>
```

- The `LIKE` operator

See https://www.w3schools.com/sql/sql_like.asp

```
SELECT "title" FROM "film" WHERE "title" LIKE '%victory%';
```

Table 12: 2 records

title
ALTER VICTORY
VICTORY ACADEMY

Remark: in SQLite, the `LIKE` operator is case insensitive. However, it is not the case for other DBs.

```
SELECT "title" FROM "film" WHERE "title" LIKE '%victory%';
```

In Postgres, there is a `ILIKE` (case insensitive `LIKE`) operator

```
SELECT "title" FROM "film" WHERE "title" ILIKE '%victory%';
```

- REGEX

Different servers use different operators to match regular expression. For MySQL, it is the REGEXP operator. For Postgresql, it is SIMILAR TO. For SQLite, it simiply doesn't support regex.

Though, Postgresql's implementation of regular expression is a bit different from the standard regex, see for example https://www.postgresql.org/docs/9.0/functions-matching.html

CASE

Similar to `case_when()` in `dplyr`.

```
SELECT
  "film_id",
  "title",
  CASE
    WHEN "length" < 60 THEN 'short'
    WHEN "length" < 90 THEN 'mid'
    ELSE 'long'
  END "len"
FROM "film";
```

Table 13: Displaying records 1 - 10

film_id	title	len
1	ACADEMY DINOSAUR	mid
2	ACE GOLDFINGER	short
3	ADAPTATION HOLES	short
4	AFFAIR PREJUDICE	long
5	AFRICAN EGG	long
6	AGENT TRUMAN	long
7	AIRPLANE SIERRA	mid
8	AIRPORT POLLOCK	short
9	ALABAMA DEVIL	long
10	ALADDIN CALENDAR	mid

JOIN operations

- Inner Join - selects records that have matching values in both tables.

```
SELECT a."inventory_id", b."customer_id"
  FROM "inventory" a
  JOIN "rental" b ON a."inventory_id" = b."inventory_id"
  ORDER BY a."inventory_id"
```

Table 14: Displaying records 1 - 10

inventory_id	customer_id
1	431
1	518
1	279
2	411
2	170
2	161
2	581
2	359
3	39
3	541

```
inner_join(
  tbl(sakila_lite, "inventory"),
  tbl(sakila_lite, "rental"),
  by = "inventory_id") %>%
  select(inventory_id, customer_id) %>%
  arrange(inventory_id) %>%
  collect()
```

```
## # A tibble: 16,044 x 2
##    inventory_id customer_id
##           <int>       <int>
##  1            1         431
##  2            1         518
##  3            1         279
##  4            2         411
##  5            2         170
##  6            2         161
##  7            2         581
##  8            2         359
##  9            3          39
## 10            3         541
## # ... with 16,034 more rows
```

We could also join a more sophisticated subquery.

- Find all the inventories where were rented and not retured.

```
SELECT a."inventory_id", b."customer_id"
  FROM "inventory" a
  JOIN (
    SELECT * FROM "rental" WHERE "return_date" IS NULL
  ) b ON a."inventory_id" = b."inventory_id"
  ORDER BY a."inventory_id"
```

Table 15: Displaying records 1 - 10

inventory_id	customer_id
6	554
9	366
21	111
25	590
70	108
81	236
97	512
106	44
112	349
177	317

- Left Join - returns all records from the left table, and the matched records from the right table

```sql
SELECT a."inventory_id", b."customer_id"
  FROM "inventory" a
  LEFT JOIN (
    SELECT * FROM "rental" WHERE "return_date" IS NULL
  ) b ON a."inventory_id" = b."inventory_id"
  ORDER BY a."inventory_id"
```

Table 16: Displaying records 1 - 10

inventory_id	customer_id
1	NA
2	NA
3	NA
4	NA
5	NA
6	554
7	NA
8	NA
9	366
10	NA

- Full Join - returns all records when there is a match in left or right table records.

SQLite doesn't support full join.

```sql
SELECT a."inventory_id", b."customer_id"
  FROM "inventory" a
  FULL JOIN (
    SELECT * FROM "rental" WHERE "return_date" IS NULL
  ) b ON a."inventory_id" = b."inventory_id"
  ORDER BY a."inventory_id"
```

- Semi Join - return all records in the left table which has a match in the right table.

```sql
SELECT *
  FROM "inventory" i
  WHERE EXISTS (
    SELECT * FROM "rental" r
      WHERE r."inventory_id" = i."inventory_id" AND "return_date" IS NULL
  )
```

Table 17: Displaying records 1 - 10

inventory_id	film_id	store_id	last_update
6	1	2	2019-04-11 18:11:48
9	2	2	2019-04-11 18:11:48
21	4	2	2019-04-11 18:11:48
25	5	2	2019-04-11 18:11:48
70	13	2	2019-04-11 18:11:48
81	17	1	2019-04-11 18:11:48

inventory_id	film_id	store_id	last_update
97	19	2	2019-04-11 18:11:48
106	21	2	2019-04-11 18:11:48
112	22	2	2019-04-11 18:11:48
177	39	2	2019-04-11 18:11:48

- Anti Join - remove all records in the left table which has a match in the right table.

```
SELECT *
  FROM "inventory" i
  WHERE NOT EXISTS (
    SELECT * FROM "rental" r
      WHERE r."inventory_id" = i."inventory_id" AND "return_date" IS NULL
  )
```

Table 18: Displaying records 1 - 10

inventory_id	film_id	store_id	last_update
1	1	1	2019-04-11 18:11:48
2	1	1	2019-04-11 18:11:48
3	1	1	2019-04-11 18:11:48
4	1	1	2019-04-11 18:11:48
5	1	2	2019-04-11 18:11:48
7	1	2	2019-04-11 18:11:48
8	1	2	2019-04-11 18:11:48
10	2	2	2019-04-11 18:11:48
11	2	2	2019-04-11 18:11:48
12	3	2	2019-04-11 18:11:48

Aggregate Functions

- AVG – calculate the average value of a set.
- COUNT – return the number of items in a set.
- SUM – return the sum all or distinct items of a set.
- MAX – find the maximum value in a set.
- MIN – find the minimum value in a set.

```
SELECT AVG("length") as "avg_length" FROM "film"
```

Table 19: 1 records

avg_length
115.272

Group By

```
SELECT "rating", AVG("length") AS "avg_length"
  FROM "film" GROUP BY "rating";
```

Table 20: 5 records

rating	avg_length
G	111.0506
NC-17	113.2286
PG	112.0052
PG-13	120.4439
R	118.6615

```
SELECT "rating", "rental_duration", AVG("length") AS "avg_length"
  FROM "film" GROUP BY "rating", "rental_duration";
```

Table 21: Displaying records 1 - 10

rating	rental_duration	avg_length
G	3	100.4286
G	4	102.2857
G	5	109.5758
G	6	128.0000
G	7	116.3448
NC-17	3	113.3243
NC-17	4	119.1515
NC-17	5	105.4186
NC-17	6	111.7895
NC-17	7	118.7000

SET Operators

- UNION and UNION ALL – combine result set of two or more queries into a single result set using the UNION and UNION ALL operators.
- INTERSECT – return the intersection of two or more queries using the INTERSECT operator.
- EXCEPT – subtract a result set from another result set using the EXCEPT operator

```
SELECT * FROM "film" where "film_id" <= 3
UNION
SELECT * FROM "film" where "film_id" <= 4;
```

film_id	title	description
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher i
2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must F

film_id	title	description
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberj
4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Mo
‘UNION ALL	‘ doesn’t remove du	plications

```
SELECT * FROM "film" where "film_id" <= 3
UNION ALL
SELECT * FROM "film" where "film_id" <= 4;
```

film_id	title	description
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The
2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Find a
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The
2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Find a
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in
4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Monkey i

```
SELECT * FROM "film" where "film_id" <= 5
INTERSECT
SELECT * FROM "film" where "film_id" >= 2;
```

film_id	title	description
2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Find a C
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in A
4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Monkey in
5	AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a Forens

```
SELECT * FROM "film" where "film_id" <= 5
EXCEPT
SELECT * FROM "film" where "film_id" >= 4;
```

film_id	title	description
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The
2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Find a
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in

Table Manipulation

```
# create a local empty SQLite database called mydb.sqlite
mydb <- dbConnect(RSQLite::SQLite(), dbname = "mydb.sqlite")
```

- delete a table permanently.

```
drop_sql <- sqlInterpolate(mydb, "DROP TABLE ?tablename;", tablename = "table1")
mydb %>% dbExecute(drop_sql)
```

Remark: the use of sqlInterpolate is to avoid SQL injection attack

- create table

```
mydb %>% dbCreateTable(
  "table1",
  tibble(fruit = character(0), count = integer(0))
)
mydb %>% dbReadTable("table1")
```

```
CREATE TABLE table2 (
    id int NOT NULL,
    last_name varchar(255) NOT NULL,
    first_name varchar(255),
    age int,
    PRIMARY KEY (id)
);
```

There is also `dbWriteTable` which export the whole data frame as a table of the database.

- INSERT – insert one or more rows into a table.

```
mydb %>% dbAppendTable(
  "table1",
  tibble(fruit = "apple", count = 2))

# alternatively
sql <- mydb %>% sqlAppendTable(
  "table1",
  tibble(fruit = "apple", count = 2), row.names = FALSE)
mydb %>% dbExecute(sql)


mydb %>% dbReadTable("table1")
```

```
INSERT INTO 'table2' (id, last_name, first_name, age)
  VALUES (1, "Lai", "Randy", 16);
```

```
INSERT INTO 'table2' (id, last_name, first_name)
  VALUES (2, "Lai", "Natalie");
```

```
mydb %>% dbReadTable("table2")
```

- UPDATE – update existing data in a table.

```
mydb %>% dbExecute("UPDATE table2 SET age = 33 WHERE id = 1;")
```

```
UPDATE table2 SET age = 33 WHERE id = 1;
```

```
mydb %>% dbReadTable("table2")
```

- DELETE – delete data from a table permanently.

```
mydb %>% dbExecute("DELETE FROM table2 WHERE id = 1;")
```

```
## [1] 0
```

```
DELETE FROM table2 WHERE id = 1;
```

```
mydb %>% dbReadTable("table2")
```

Check the `shiny_with_db` folder for a shiny app that connects to a remote postgresql server.

Reference

- SQL Tutorial https://www.sqltutorial.org/