Assignment / Explore Query Planning

Connor Clancy - clancy.co@northeastern.edu

Spring 2023

Libary Imports

```
library(RSQLite)
library(RMySQL)
```

```
## Loading required package: DBI

##
## Attaching package: 'RMySQL'

## The following object is masked from 'package:RSQLite':
##
##     isIdCurrent
```

```
library(readr)
library(sqldf)
```

```
## Loading required package: gsubfn

## Loading required package: proto

## sqldf will default to using MySQL
```

```
options(sqldf.driver = "SQLite")
```

Connect to SQLite Database

```
fpath = getwd()
dbfile = "/sakila.db"

# connect to the database if exists, else create a new database
lcon <- dbConnect(RSQLite::SQLite(), paste0(fpath, dbfile))
```

```
dbGetQuery(lcon, "SELECT * FROM film LIMIT 5;")
```

```
##   film_id            title
## 1       1 ACADEMY DINOSAUR
## 2       2   ACE GOLDFINGER
## 3       3 ADAPTATION HOLES
## 4       4 AFFAIR PREJUDICE
## 5       5      AFRICAN EGG
##
## 1                      A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in Th
## 2                  A Astounding Epistle of a Database Administrator And a Explorer who must Find a Ca
## 3                      A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack
## 4                          A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Monk
## 5 A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a Forensic Psychologist in
##   release_year language_id original_language_id rental_duration rental_rate
## 1         2006           1                   NA               6        0.99
## 2         2006           1                   NA               3        4.99
## 3         2006           1                   NA               7        2.99
## 4         2006           1                   NA               5        2.99
## 5         2006           1                   NA               6        2.99
##   length replacement_cost rating                 special_features
## 1     86            20.99     PG Deleted Scenes,Behind the Scenes
## 2     48            12.99      G          Trailers,Deleted Scenes
## 3     50            18.99  NC-17          Trailers,Deleted Scenes
## 4    117            26.99      G   Commentaries,Behind the Scenes
## 5    130            22.99      G                   Deleted Scenes
##           last_update
## 1 2006-02-15 05:03:42
## 2 2006-02-15 05:03:42
## 3 2006-02-15 05:03:42
## 4 2006-02-15 05:03:42
## 5 2006-02-15 05:03:42
```

Connect to MySQL Database

```
db_user <- 'admin'
db_password <- 'Northea$tern23'
db_name <- 'sakila'
db_host <- 'cclancy-cs5200.cbowkysg1oyc.us-east-2.rds.amazonaws.com'
db_port <- 3306
mscon <- dbConnect(MySQL(), user = db_user, password = db_password,
                   dbname = db_name, host = db_host, port = db_port)
```

```
dbGetQuery(mscon, "SELECT * FROM film LIMIT 5;")
```

```
## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 0 imported as
## numeric

## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 4 imported as
## numeric

## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 5 imported as
## numeric

## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 6 imported as
## numeric

## Warning in .local(conn, statement, ...): Decimal MySQL column 7 imported as
## numeric

## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 8 imported as
## numeric

## Warning in .local(conn, statement, ...): Decimal MySQL column 9 imported as
## numeric

## Warning in .local(conn, statement, ...): unrecognized MySQL field type 7 in
## column 12 imported as character
##   film_id            title
## 1       1 ACADEMY DINOSAUR
## 2       2   ACE GOLDFINGER
## 3       3 ADAPTATION HOLES
## 4       4 AFFAIR PREJUDICE
## 5       5      AFRICAN EGG
##
## 1                      A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in Th
## 2                  A Astounding Epistle of a Database Administrator And a Explorer who must Find a Ca
## 3                      A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack i
## 4                          A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Mon
## 5 A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a Forensic Psychologist in
##   release_year language_id original_language_id rental_duration rental_rate
## 1         2006           1                   NA               6        0.99
## 2         2006           1                   NA               3        4.99
## 3         2006           1                   NA               7        2.99
## 4         2006           1                   NA               5        2.99
## 5         2006           1                   NA               6        2.99
##   length replacement_cost rating                 special_features
## 1     86            20.99     PG Deleted Scenes,Behind the Scenes
## 2     48            12.99      G          Trailers,Deleted Scenes
## 3     50            18.99  NC-17          Trailers,Deleted Scenes
## 4    117            26.99      G   Commentaries,Behind the Scenes
## 5    130            22.99      G                   Deleted Scenes
##           last_update
## 1 2006-02-15 05:03:42
## 2 2006-02-15 05:03:42
## 3 2006-02-15 05:03:42
## 4 2006-02-15 05:03:42
## 5 2006-02-15 05:03:42
```

Tasks

Question 1

Ensuring that no user-defined indexes exist (delete all user-defined indexes, if there are any), find the number of films per category. The query should return the category name and the number of films in each category. Show us the code that determines if there are any indexes and the code to delete them if there are any.

```
delete_idx <- function (tables, dbcon, database) {

  if (database == "mysql") {
    for (t in tables) {
      df_idx <- dbGetQuery(dbcon, sprintf("SHOW INDEXES FROM %s
                                          WHERE Key_name != 'PRIMARY'
                                          AND Key_name NOT LIKE '%%fk%%';", t))

      for (i in df_idx$Key_name) {
        dbExecute(dbcon, sprintf("DROP INDEX %s ON %s;", i, t))
      }
    }
  } else if (database == "sqlite") {
    for (t in tables) {
      df_idx <- dbGetQuery(dbcon, sprintf("SELECT name FROM sqlite_master
                                          WHERE type == 'index'
                                          AND tbl_name == '%s'
                                          AND name NOT LIKE '%%autoindex%%'",t))

      for (i in df_idx$name) {
        dbExecute(dbcon, sprintf("DROP INDEX %s", i))
      }
    }
  }

}
```

```
delete_idx(c('film', 'category', 'film_category'), lcon, "sqlite")
```

```
dbGetQuery(lcon, "
           SELECT
             c.category_id,
             c.name AS category_name,
             COUNT(f.film_id) AS film_count
           FROM film AS f
           INNER JOIN film_category AS j
             ON f.film_id = j.film_id
           INNER JOIN category AS c
             ON c.category_id = j.category_id
          GROUP BY
             c.category_id,
             c.name
          ORDER BY
             COUNT(f.film_id) DESC;
           ")
```

```
##    category_id category_name film_count
## 1            15        Sports         74
## 2             9       Foreign         73
## 3             8        Family         69
## 4             6   Documentary         68
## 5             2     Animation         66
## 6             1        Action         64
## 7            13           New         63
## 8             7         Drama         62
## 9            10         Games         61
## 10           14        Sci-Fi         61
## 11            3      Children         60
## 12            5        Comedy         58
## 13            4      Classics         57
## 14           16        Travel         57
## 15           11        Horror         56
## 16           12         Music         51
```

Question 2

Ensuring that no user-defined indexes exist (delete all user-defined indexes, if there are any), execute the same query (same SQL) as in (1) but against the MySQL database. Make sure you reuse the same SQL query string as in (1)

```
delete_idx(c('film', 'category', 'film_category'), mscon, "mysql")
```

```
## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 3 imported as
## numeric
```

```
## Warning in .local(conn, statement, ...): unrecognized MySQL field type 6 in
## column 8 imported as character
```

```
## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 3 imported as
## numeric
```

```
## Warning in .local(conn, statement, ...): unrecognized MySQL field type 6 in
## column 8 imported as character
```

```
## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 3 imported as
## numeric
```

```
## Warning in .local(conn, statement, ...): unrecognized MySQL field type 6 in
## column 8 imported as character
```

```
dbGetQuery(mscon, "
           SELECT
             c.category_id,
             c.name AS category_name,
             COUNT(f.film_id) AS film_count
           FROM film AS f
           INNER JOIN film_category AS j
             ON f.film_id = j.film_id
           INNER JOIN category AS c
             ON c.category_id = j.category_id
          GROUP BY
             c.category_id,
             c.name
          ORDER BY
```

```
             COUNT(f.film_id) DESC;
           ")
```

```
## Warning in .local(conn, statement, ...): Unsigned INTEGER in col 0 imported as
## numeric
```

```
##    category_id category_name film_count
## 1           15        Sports         74
## 2            9       Foreign         73
## 3            8        Family         69
## 4            6   Documentary         68
## 5            2     Animation         66
## 6            1        Action         64
## 7           13           New         63
## 8            7         Drama         62
## 9           10         Games         61
## 10          14        Sci-Fi         61
## 11           3      Children         60
## 12           5        Comedy         58
## 13           4      Classics         57
## 14          16        Travel         57
## 15          11        Horror         56
## 16          12         Music         51
```

Question 3

Find out how to get the query plans for SQLite and MySQL and then display the query plans for each of the query executions in (1) and (2).

```
dbGetQuery(lcon, "
           EXPLAIN QUERY PLAN SELECT
             c.category_id,
             c.name AS category_name,
             COUNT(f.film_id) AS film_count
           FROM film AS f
           INNER JOIN film_category AS j
             ON f.film_id = j.film_id
           INNER JOIN category AS c
             ON c.category_id = j.category_id
          GROUP BY
             c.category_id,
             c.name
          ORDER BY
             COUNT(f.film_id) DESC;
           ")
```

```
##   id parent notused
## 1  9      0       0
## 2 11      0       0
## 3 14      0       0
## 4 17      0       0
## 5 57      0       0
##                                                          detail
## 1 SCAN j USING COVERING INDEX sqlite_autoindex_film_category_1
## 2                SEARCH f USING INTEGER PRIMARY KEY (rowid=?)
## 3                 SEARCH c USING INTEGER PRIMARY KEY (rowid=?)
## 4                                 USE TEMP B-TREE FOR GROUP BY
## 5                                 USE TEMP B-TREE FOR ORDER BY
```

```
dbGetQuery(mscon, "
           EXPLAIN SELECT
             c.category_id,
             c.name AS category_name,
             COUNT(f.film_id) AS film_count
           FROM film AS f
           INNER JOIN film_category AS j
             ON f.film_id = j.film_id
           INNER JOIN category AS c
             ON c.category_id = j.category_id
          GROUP BY
             c.category_id,
             c.name
          ORDER BY
             COUNT(f.film_id) DESC;
           ")
```

```
##   id select_type table partitions   type                     possible_keys
## 1  1      SIMPLE     c       <NA>    ALL                           PRIMARY
## 2  1      SIMPLE     j       <NA>    ref PRIMARY,fk_film_category_category
## 3  1      SIMPLE     f       <NA> eq_ref                           PRIMARY
##                         key key_len                  ref rows filtered
## 1                      <NA>    <NA>                 <NA>   16      100
## 2 fk_film_category_category       1 sakila.c.category_id   62      100
## 3                   PRIMARY       2     sakila.j.film_id    1      100
##                             Extra
## 1 Using temporary; Using filesort
## 2                     Using index
## 3                     Using index
```

Question 4

Comment on the differences between the query plans? Are they the same? How do they differ? Why do you think they differ? Do both take the same amount of time?

The two plans are not the same. SQLite appears to have extra steps for the `GROUP BY` and `ORDER BY` clauses that MySQL is able to handle within one of the other steps in the plan. It appears to me that MySQL runs quicker.

Question 5

Write a SQL query against the SQLite database that returns the title, language and length of the film with the title "ZORRO ARK".

```
dbGetQuery(lcon, "
           SELECT
              f.title,
              l.name AS language,
              f.length
           FROM film AS f
           INNER JOIN language AS l
              ON f.language_id = l.language_id
           WHERE title = 'ZORRO ARK';
           ")
```

```
##       title language length
## 1 ZORRO ARK  English     50
```

Question 6

For the query in (5), display the query plan.

```
dbGetQuery(lcon, "
           EXPLAIN QUERY PLAN SELECT
              f.title,
              l.name AS language,
              f.length
           FROM film AS f
           INNER JOIN language AS l
              ON f.language_id = l.language_id
           WHERE title = 'ZORRO ARK';
           ")
```

```
##   id parent notused                                       detail
## 1  3      0       0                                       SCAN f
## 2  7      0       0 SEARCH l USING INTEGER PRIMARY KEY (rowid=?)
```

Question 7

In the SQLite database, create a user-defined index called "TitleIndex" on the column TITLE in the table FILM.

```
dbExecute(lcon,
           'CREATE UNIQUE INDEX IF NOT EXISTS TitleIndex
           ON film (title);')
```

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

Question 8

Re-run the query from (5) now that you have an index and display the query plan.

```
dbGetQuery(lcon, "
           EXPLAIN QUERY PLAN SELECT
              f.title,
              l.name AS language,
              f.length
           FROM film AS f
           INNER JOIN language AS l
              ON f.language_id = l.language_id
           WHERE title = 'ZORRO ARK';
           ")
```

```
##   id parent notused                                       detail
## 1  4      0       0    SEARCH f USING INDEX TitleIndex (title=?)
## 2  9      0       0 SEARCH l USING INTEGER PRIMARY KEY (rowid=?)
```

Question 9

Are the query plans the same in (6) and (8)? What are the differences? Is there a difference in execution time? How do you know from the query plan whether it uses an index or not?

No, the query plans are not the same. In (6) the query scans the whole table to find the movie, where as in (8) the query plan states that it is going to use the index created to complete the search. The query with the index performs faster.

Question 10

Write a SQL query against the SQLite database that returns the title, language and length of all films with the word "GOLD" with any capitalization in its name, i.e., it should return "Gold Finger", "GOLD FINGER", "THE GOLD FINGER", "Pure GOLD" (these are not actual titles).

```
dbGetQuery(lcon, "
           SELECT
              f.title,
              l.name AS language,
              f.length
           FROM film AS f
           INNER JOIN language AS l
              ON f.language_id = l.language_id
           WHERE lower(title) LIKE '%gold%';
           ")
```

```
##                     title language length
## 1          ACE GOLDFINGER  English     48
## 2    BREAKFAST GOLDFINGER  English    123
## 3              GOLD RIVER  English    154
## 4 GOLDFINGER SENSIBILITY  English     93
## 5          GOLDMINE TYCOON  English    153
## 6              OSCAR GOLD  English    115
## 7    SILVERADO GOLDFINGER  English     74
## 8              SWARM GOLD  English    123
```

Question 11

Get the query plan for (10). Does it use the index you created? If not, why do you think it didn't?

```
dbGetQuery(lcon, "
           EXPLAIN QUERY PLAN SELECT
              f.title,
              l.name AS language,
              f.length
           FROM film AS f
           INNER JOIN language AS l
              ON f.language_id = l.language_id
           WHERE lower(title) LIKE '%gold%';
           ")
```

```
##   id parent notused                                       detail
## 1  3      0       0                                       SCAN f
## 2  9      0       0 SEARCH l USING INTEGER PRIMARY KEY (rowid=?)
```

The query does not use the Index we created because we are using a wild card to search through the column. Since we have to search every single row for the wildcard string to see if it contains ‘gold’ SQLite appears to just choose to scan the whole table directly rather than using the index.