# Assignment / Explore Query Planning

Jiaxing Tan

2023-03-20

```
library(pacman)
p_load(RMySQL, quietly=T)
```

```
## Installing package into 'C:/Users/PC/Documents/R/win-library/4.1'
## (as 'lib' is unspecified)
```

```
p_load(tidyverse)
p_load(lubridate)
p_load(RSQLite)
```

```
rootDB <- "databases"
dbfile = "sakila.db"

path <- getwd()
rootPath <- file.path(path, rootDB)
if (dir.exists(rootPath)) {
  message("Database root folder is already exist.")
  # return()
} else {
  if(!dir.create(rootPath)) stop("Given path in configDB is not exist. Please input a valid path for da
}
```

```
## Database root folder is already exist.
```

```
conn <- dbConnect(RSQLite::SQLite(), file.path(rootPath, dbfile))
```

Only for check purpose, not part of assignment.

```
SELECT name
FROM sqlite_schema
WHERE type ='table'
AND name NOT LIKE 'sqlite_%'
LIMIT 5
```

Table 1: 5 records

| name |
|---|
| actor |
| address |
| category |
| city |
| country |

```
# db_cred <- yaml::read_yaml('dbconfig.yml')
db_user <- "admin"
db_password <- "cs5200db"
db_name <- "sakila"
db_host <- "cs5200database.cbjmmcav6ldb.us-west-2.rds.amazonaws.com"
db_port <- 3306

mySQLConn <-  dbConnect(RMySQL::MySQL(),
                        user = db_user,
                        password = db_password,
                        dbname = db_name,
                        host = db_host,
                        port = db_port)
```

Only for check purpose, not part of assignment.

```
SELECT table_name
FROM information_schema.tables
WHERE table_type='BASE TABLE'
  AND table_schema = 'sakila'
LIMIT 5
```

Table 2: 5 records

| TABLE_NAME |
|---|
| actor |
| address |
| category |
| city |
| country |

## Bulk loading MySql database

```
sqlStatement <- "
SELECT name
FROM sqlite_schema
WHERE type ='table'
AND name NOT LIKE 'sqlite_%'
"
```

```
tableName <- dbGetQuery(conn, sqlStatement)

transmitDB <- function(tbName) {
   tmpTable <- dbReadTable(conn, tbName)
   dbWriteTable(mySQLConn, tbName, tmpTable, overwrite = T)
}

lapply(tableName$name, transmitDB)
```

## Question 1

(10 pts / 10 min) 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.

```
sqlStatement <- "
SELECT
   `type`,
   `name`,
   `tbl_name`,
   `sql`
FROM sqlite_master
WHERE `type` = 'index';
"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##    type                              name      tbl_name  sql
## 1 index    sqlite_autoindex_film_actor_1    film_actor <NA>
## 2 index sqlite_autoindex_film_category_1 film_category <NA>
```

No UD-index in sqlite DB

Number of films per category

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id
    FROM film
)
,t2 AS (
    SELECT *
    FROM film_category
)
,t3 AS (
    SELECT *
    FROM category
)

SELECT t3.name, count(distinct t1.film_id) as file_tally
FROM t1 join t2 ON t1.film_id = t2.film_id
```

```
JOIN t3 ON t2.category_id = t3.category_id
GROUP BY t3.name

"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##           name file_tally
## 1       Action         64
## 2    Animation         66
## 3     Children         60
## 4     Classics         57
## 5       Comedy         58
## 6  Documentary         68
## 7        Drama         62
## 8       Family         69
## 9      Foreign         73
## 10       Games         61
```

## 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).

```
sqlStatement = "
SELECT DISTINCT
    TABLE_NAME,
    INDEX_NAME
FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_SCHEMA = 'sakila'
"
bs = dbGetQuery(mySQLConn, sqlStatement)
bs
```

```
## [1] TABLE_NAME INDEX_NAME
## <0 rows> (or 0-length row.names)
```

No UD-index in MySQL DB

Number of films per category

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id
    FROM film
)
,t2 AS (
    SELECT *
    FROM film_category
)
,t3 AS (
```

```
    SELECT *
    FROM category
)

SELECT t3.name, count(distinct t1.film_id) as file_tally
FROM t1 join t2 ON t1.film_id = t2.film_id
JOIN t3 ON t2.category_id = t3.category_id
GROUP BY t3.name

"
bs = dbGetQuery(mySQLConn, sqlStatement)
head(bs, 10)
```

```
##           name file_tally
## 1       Action         64
## 2    Animation         66
## 3     Children         60
## 4     Classics         57
## 5       Comedy         58
## 6  Documentary         68
## 7        Drama         62
## 8       Family         69
## 9      Foreign         73
## 10       Games         61
```

## 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).

For sqlite

```
sqlStatement = "
EXPLAIN QUERY PLAN
WITH t1 AS (
    SELECT film_id
    FROM film
)
,t2 AS (
    SELECT *
    FROM film_category
)
,t3 AS (
    SELECT *
    FROM category
)

SELECT t3.name, count(distinct t1.film_id) as file_tally
FROM t1 join t2 ON t1.film_id = t2.film_id
JOIN t3 ON t2.category_id = t3.category_id
GROUP BY t3.name

"
```

```
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##   id parent notused
## 1  8      0       0
## 2 10      0       0
## 3 13      0       0
## 4 16      0       0
## 5 53      0       0
##                                                                 detail
## 1 SCAN film_category USING COVERING INDEX sqlite_autoindex_film_category_1
## 2                          SEARCH film USING INTEGER PRIMARY KEY (rowid=?)
## 3                     SEARCH category USING INTEGER PRIMARY KEY (rowid=?)
## 4                                          USE TEMP B-TREE FOR GROUP BY
## 5                                    USE TEMP B-TREE FOR count(DISTINCT)
```

For mySQL

```
sqlStatement = "
EXPLAIN
WITH t1 AS (
    SELECT film_id
    FROM film
)
,t2 AS (
    SELECT *
    FROM film_category
)
,t3 AS (
    SELECT *
    FROM category
)

SELECT t3.name, count(distinct t1.film_id) as file_tally
FROM t1 join t2 ON t1.film_id = t2.film_id
JOIN t3 ON t2.category_id = t3.category_id
GROUP BY t3.name

"
bs = dbGetQuery(mySQLConn, sqlStatement)
head(bs, 10)
```

```
##   id select_type         table partitions type possible_keys  key key_len  ref
## 1  1      SIMPLE      category       <NA>  ALL          <NA> <NA>    <NA> <NA>
## 2  1      SIMPLE film_category       <NA>  ALL          <NA> <NA>    <NA> <NA>
## 3  1      SIMPLE          film       <NA>  ALL          <NA> <NA>    <NA> <NA>
##   rows filtered                                 Extra
## 1   16      100            Using temporary; Using filesort
## 2 1000       10 Using where; Using join buffer (hash join)
## 3 1000       10 Using where; Using join buffer (hash join)
```

## 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?

Sqlite table "film_category" has already existing index (import with the db).

MySQL tables create from scratch and without any index.

These two plans take separate execution times.

For SQLite, because "film_category" table has an index "sqlite_autoindex_film_category_1" that cover all the information one need to finish the join operation, query engine could use the index table instead of the origin table to do the task for the sake of efficiency. And it has to "SCAN" the whole index table because every record need be joined here.

Then the engine look up every counterpart tuple in "film" table (nested loop) by rowid via binary-search for increasing search performance.

Ditto "category" but in the next nesting order.

Then the engine sort the table twice with "TEMP B-TREE" for both GROUP BY and DISTINCT operations.

For MySQL query, because type is ALL for each table, this output indicates that MySQL is generating a Cartesian product of all the tables. This takes quite a long time, because the product of the number of rows in each table must be examined. The intermediate table has 16 * 1000 * 1000 rows.

In addition, the Extra output contains additional information about how MySQL resolves the query. There are values 'Using filesort' and 'Using temporary' indicate large time cost for sort rows and space cost for store temporary table during the query execution.

To sum up, sqlite is much fast compared to MySQL because the "connecting table" has a covering index and each table has extra rowid column. These features can significantly speed up the lookup process.

## 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".

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title = 'ZORRO ARK'
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;

"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

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

## Question 6

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

```
sqlStatement = "
EXPLAIN QUERY PLAN
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title = 'ZORRO ARK'
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;

"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##   id parent notused
## 1  3      0       0
## 2  7      0       0
##                                                          detail
## 1                                                      SCAN film
## 2 SEARCH language USING INTEGER PRIMARY KEY (rowid=?) LEFT-JOIN
```

## Question 7

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

```
sqlStatement = "
DROP INDEX IF EXISTS TitleIndex
"
dbGetQuery(conn, sqlStatement)
```

```
## data frame with 0 columns and 0 rows
```

```
sqlStatement = "
CREATE INDEX IF NOT EXISTS TitleIndex ON film(title)
"
dbGetQuery(conn, sqlStatement)
```

```
## data frame with 0 columns and 0 rows
```

```
sqlStatement <- "
SELECT
   `type`,
   `name`,
   `tbl_name`,
   `sql`
FROM sqlite_master
WHERE `type` = 'index';
"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##    type                              name      tbl_name
## 1 index    sqlite_autoindex_film_actor_1    film_actor
## 2 index sqlite_autoindex_film_category_1 film_category
## 3 index                      TitleIndex          film
##                                     sql
## 1                                  <NA>
## 2                                  <NA>
## 3 CREATE INDEX TitleIndex ON film(title)\n
```

## Question 8

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

```
sqlStatement = "
EXPLAIN QUERY PLAN
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title = 'ZORRO ARK'
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;

"
bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##   id parent notused
## 1  4      0       0
## 2  9      0       0
##                                                          detail
## 1                  SEARCH film USING INDEX TitleIndex (title=?)
## 2 SEARCH language USING INTEGER PRIMARY KEY (rowid=?) LEFT-JOIN
```

## 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?

Obviously not the same. In (6) the query engine has to scan the whole table "file" to find the one we want with linear time. But in (8) it can search the record by using TitleIndex to quick locate the target tuples.

Let's measure Run-Time performance of (6) and (8)

First drop the TitleIndex.

```
sqlStatement = "
DROP INDEX IF EXISTS TitleIndex
"
dbGetQuery(conn, sqlStatement)
```

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title = 'ZORRO ARK'
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;

"
bt <- Sys.time()
bs = dbGetQuery(conn, sqlStatement)
et <- Sys.time()
t.loop <- et - bt

cat("Time elapsed: ", round((t.loop),3), " sec")
```

```
## Time elapsed:  0.004  sec
```

The above is the time cost for (6)

Then add back the index.

```
sqlStatement = "
DROP INDEX IF EXISTS TitleIndex
"
dbGetQuery(conn, sqlStatement)

sqlStatement = "
CREATE INDEX IF NOT EXISTS TitleIndex ON film(title)
"
dbGetQuery(conn, sqlStatement)
```

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title = 'ZORRO ARK'
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;

"
bt <- Sys.time()
bs = dbGetQuery(conn, sqlStatement)
et <- Sys.time()
t.loop <- et - bt

cat("Time elapsed: ", round((t.loop),3), " sec")
```

```
## Time elapsed:  0.003  sec
```

As list above, the plan in (8) is almost twice as fast as the plan in (6).

## 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).

```
sqlStatement = "
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title like '%gold%'
      AND LOWER(title) <> title
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;
"

bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##                    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?

```
sqlStatement = "
EXPLAIN QUERY PLAN
WITH t1 AS (
    SELECT film_id, title, language_id, length
    FROM film
    WHERE title like '%gold%'
      AND LOWER(title) <> title
)
,t2 AS (
    SELECT *
    FROM language
)
SELECT title, t2.name as language, length
FROM t1
LEFT JOIN t2
ON t1.language_id = t2.language_id;
"

bs = dbGetQuery(conn, sqlStatement)
head(bs, 10)
```

```
##   id parent notused
## 1  3      0       0
## 2 12      0       0
##                                                          detail
## 1                                                     SCAN film
## 2 SEARCH language USING INTEGER PRIMARY KEY (rowid=?) LEFT-JOIN
```

It didn't use the TitleIndex in film table, because there is a 'like' pattern matching search which invalidate the index search in this subquery.

Drop the TitleIndex.

```
sqlStatement = "
DROP INDEX IF EXISTS TitleIndex
"
dbGetQuery(conn, sqlStatement)
```

Disconnect

```
dbDisconnect(mySQLConn)
```

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

```
dbDisconnect(conn)
```