Assignment / Explore Query Planning and Indexing

Xujia Qin

Jun 8th, 2025

Question 1

```
# Drop any user-defined indexes (ignore errors if they don't exist)
# First, list all indexes in the DB and drop those not part of the system
indexes <- dbGetQuery(conn, "
  SELECT name
  FROM sqlite_master
  WHERE type = 'index'
    AND sql IS NOT NULL;
")

for (index_name in indexes$name) {
  drop_sql <- sprintf("DROP INDEX IF EXISTS [%s];", index_name)
  dbExecute(conn, drop_sql)
}

# Run query to get number of films per language
query <- "
  SELECT l.name AS language, COUNT(f.film_id) AS film_count
  FROM film f
  JOIN language l ON f.language_id = l.language_id
  GROUP BY l.name;
"

result <- dbGetQuery(conn, query)
print(result)
```

```
##   language film_count
## 1  English       1000
```

Question 2

```
# Wrap the with EXPLAIN QUERY PLAN
explain_query <- "
  EXPLAIN QUERY PLAN
  SELECT l.name AS language, COUNT(f.film_id) AS film_count
  FROM film f
  JOIN language l ON f.language_id = l.language_id
  GROUP BY l.name;
"
```

```
query_plan <- dbGetQuery(conn, explain_query)
print(query_plan)
```

```
##   id parent notused                                        detail
## 1  7      0     216                                        SCAN f
## 2  9      0      45 SEARCH l USING INTEGER PRIMARY KEY (rowid=?)
## 3 12      0       0                 USE TEMP B-TREE FOR GROUP BY
```

Question 3

```
# Query to get title, category name, and length for the film "ZORRO ARK"
query3 <- "
  SELECT f.title, c.name AS category, f.length
  FROM film f
  JOIN film_category fc ON f.film_id = fc.film_id
  JOIN category c ON fc.category_id = c.category_id
  WHERE f.title = 'ZORRO ARK';
"

result3 <- dbGetQuery(conn, query3)
print(result3)
```

```
##       title category length
## 1 ZORRO ARK   Comedy     50
```

```
# Measure execution time without index
time_no_index <- system.time({
  result_no_index <- dbGetQuery(conn, query3)
})
print(time_no_index)
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

Question 4

```
# Query plan for retrieving title, category name, and length of "ZORRO ARK"
query4 <- "
EXPLAIN QUERY PLAN
SELECT f.title, c.name AS category, f.length
FROM film f
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
WHERE f.title = 'ZORRO ARK';
"

plan4 <- dbGetQuery(conn, query4)
print(plan4)
```

```
##   id parent notused
## 1  4      0     214
## 2  6      0      45
## 3  9      0      45
##                                                 detail
```

```
## 1 SCAN fc USING COVERING INDEX sqlite_autoindex_film_category_1
## 2                   SEARCH c USING INTEGER PRIMARY KEY (rowid=?)
## 3                   SEARCH f USING INTEGER PRIMARY KEY (rowid=?)
```

Question 5

```
# Create an index named TitleIndex on film.title
create_index_sql <- "
CREATE INDEX IF NOT EXISTS TitleIndex ON film (title COLLATE NOCASE);
"

dbExecute(conn, create_index_sql)
```

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

```
# Confirm index creation by listing indexes again
indexes_after <- dbGetQuery(conn, "
  SELECT name FROM sqlite_master
  WHERE type = 'index' AND name = 'TitleIndex';
")
print(indexes_after)
```

```
##         name
## 1 TitleIndex
```

Question 6

```
# Re-run the query from Question 3 with EXPLAIN QUERY PLAN to see if index is used
query6 <- "
EXPLAIN QUERY PLAN
SELECT f.title, c.name AS category, f.length
FROM film f
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
WHERE f.title = 'ZORRO ARK';
"

plan6 <- dbGetQuery(conn, query6)
print(plan6)
```

```
##   id parent notused
## 1  4      0     214
## 2  6      0      45
## 3  9      0      45
##                                                    detail
## 1 SCAN fc USING COVERING INDEX sqlite_autoindex_film_category_1
## 2                   SEARCH c USING INTEGER PRIMARY KEY (rowid=?)
## 3                   SEARCH f USING INTEGER PRIMARY KEY (rowid=?)
```

```
# Measure execution time with index
time_with_index <- system.time({
  result_with_index <- dbGetQuery(conn, query6)
})
print(time_with_index)
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

Question 7

Are the query plans the same in (4) and (6)? What are the differences?

No, the query plans differ between q4 and q6.

- In **Q4** (before creating the `TitleIndex`), the plan shows:
  - `SCAN fc USING COVERING INDEX sqlite_autoindex_film_category_1`: scanning the `film_category` table using an automatic index.
  - `SEARCH c USING INTEGER PRIMARY KEY (rowid=?)`: efficient lookup on `category` table by primary key.
  - `SEARCH f USING INTEGER PRIMARY KEY (rowid=?)`: efficient lookup on `film` table by primary key.

  Notably, the plan does **not** show using an index on `film.title` for filtering. The lookup of film rows is done by primary key (rowid) after scanning film_category.
- In **Q6** (after creating `TitleIndex`), the plan shows:
  - `SEARCH f USING INDEX TitleIndex (title=?)`: SQLite uses the user-defined index on `film.title` to directly find matching film(s).
  - `SEARCH fc USING COVERING INDEX sqlite_autoindex_film_category_1 (film_id=?)`: efficient lookup on `film_category` by film_id.
  - `SEARCH c USING INTEGER PRIMARY KEY (rowid=?)`: efficient lookup on `category`.

How do you know from the query plan whether it uses an index?

- The **presence of `SEARCH ... USING INDEX TitleIndex`** on the `film` table shows an explicit use of your custom index.
- The **absence of a "SCAN" or full scan** on the `film` table and the use of `SEARCH ... USING INTEGER PRIMARY KEY` means SQLite is relying on rowids (primary keys) rather than an index on the `title`.
- In Question 4, the filtering by title was not done by an index, causing more work in locating the correct film row.
- Using an index typically improves query performance by reducing the number of rows SQLite needs to examine. In this case, the index on `title` helps SQLite jump directly to matching records, rather than checking every film.

Question 8 - Execution Time Comparison

- Without index (execution time logged in Q3)
- With index (execution time logged in Q6)
- The query without an index took approximately 0.002 seconds to run, while the query with the `TitleIndex` on the `film.title` column executed in just 0.001 seconds.
- This confirms that the index improves query performance, even a little bit, making the database lookup for the film title much faster by avoiding full table scans.

Question 9

```
# Query to find all films with "gold" (case-insensitive) in the title
query9 <- "
  SELECT title, language.name AS language, length
  FROM film
```

```
  JOIN language ON film.language_id = language.language_id
  WHERE title LIKE '%GOLD%' COLLATE NOCASE;
"

# Execute the query
gold_result <- dbGetQuery(conn, query9)
print(gold_result)
```

```
##                    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 10

```
# Query plan for the "gold" title search
query_plan_gold <- "
EXPLAIN QUERY PLAN
SELECT title, language.name AS language, length
FROM film
JOIN language ON film.language_id = language.language_id
WHERE title LIKE '%GOLD%' COLLATE NOCASE;
"

# Execute the query plan
plan_result_gold <- dbGetQuery(conn, query_plan_gold)
print(plan_result_gold)
```

```
##   id parent notused                                           detail
## 1  3      0     216                                        SCAN film
## 2  8      0      45 SEARCH language USING INTEGER PRIMARY KEY (rowid=?)
```

```
# Query plan for the prefix "gold" title search
query_plan_goldprefix <- "
EXPLAIN QUERY PLAN
SELECT title, language.name AS language, length
FROM film
JOIN language ON film.language_id = language.language_id
WHERE title LIKE 'GOLD%' COLLATE NOCASE;
"

# Execute the query plan
plan_result_goldprefix <- dbGetQuery(conn, query_plan_goldprefix)
print(plan_result_goldprefix)
```

```
##   id parent notused                                                 detail
## 1  4      0     166 SEARCH film USING INDEX TitleIndex (title>? AND title<?)
## 2 15      0      45      SEARCH language USING INTEGER PRIMARY KEY (rowid=?)
```

Reflections:

- LIKE '%gold%' Uses index: No
  - Why: When the pattern starts with a wildcard (% or _), SQLite can't predict where in the index to start scanning, so it defaults to a full table scan.
- LIKE 'gold%' Uses index: Yes
  - When the pattern starts with a fixed string (like 'Gold%'), SQLite can use the index on that column to do an efficient range scan. This is because the index is ordered, so SQLite can jump right to the entries starting with 'Gold' and scan forward until it hits something that doesn't match.