CS387 Lab8

Utkarsh Ranjan and Utkarsh Pratap Singh

March 2023

1 Queries:

1. **Query :**

```sql
SELECT * FROM takes WHERE id='31820';
```

Query Plan:

```
                               QUERY PLAN
--------------------------------------------------------------
 Bitmap Heap Scan on takes
   Recheck Cond: ((id)::text = '31820'::text)
   ->  Bitmap Index Scan on takes_pkey
         Index Cond: ((id)::text = '31820'::text)
```

Reasoning :

- For a small number of rows, index scan is used while when number of rows is large seq scan is used.
- For intermediate case, bitmap scan is used such that a single block is only used once during a scan.
- Here because we needed to keep the number of rows small I chose to put condition on the primary key itself i.e, id

2. **Query:**

```sql
CREATE INDEX takes_course_id_idx ON takes(course_id);
CREATE INDEX takes_semester_year_idx ON takes(semester, year);

SELECT *
FROM takes
WHERE course_id = '974' AND year = 2003;
```

Query plan:

```
                           QUERY PLAN
-------------------------------------------------------------
 Bitmap Heap Scan on takes
   Recheck Cond: ((course_id)::text = '974'::text)
   Filter: (year = '2003'::numeric)
   ->  Bitmap Index Scan on takes_course_id_idx
         Index Cond: ((course_id)::text = '974'::text)
```

Reasoning: The query is using a selection query because it is filtering the records from the "takes" table based on the conditions "course_id = '974'" and "year = 2003". To perform this filtering efficiently, the PostgreSQL query planner would look for an index that has the columns used in the WHERE clause. In this case, there are two indexes created on the "takes" table - one on "course_id" and another on "semester" and "year". The query planner would choose the index that is more selective and can efficiently filter the records. In this case, the index on "course_id" would be more selective and hence, the query planner would use it to perform the selection.

3.
```sql
CREATE INDEX takes_course_id_idx ON takes(course_id);
CREATE INDEX takes_semester_idx ON takes(semester);

SELECT *
FROM takes
WHERE course_id = '974' OR semester = 'Fall';
```

Query plan:

```
                          QUERY PLAN
-------------------------------------------------------------
 Bitmap Heap Scan on takes
   Recheck Cond: (((course_id)::text = '974'::text) OR ((
    semester)::text = 'Fall'::text))
   ->  BitmapOr
         ->  Bitmap Index Scan on takes_course_id_idx
               Index Cond: ((course_id)::text = '974'::text)
         ->  Bitmap Index Scan on takes_semester_idx
               Index Cond: ((semester)::text = 'Fall'::text)
```

Reasoning: Similar to query 2.

4. **Query:**

```sql
SELECT *
FROM student
JOIN takes ON student.ID = takes.ID
WHERE student.tot_cred = 42;
```

Query plan:

```
                              QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   ->  Seq Scan on student
         Filter: (tot_cred = '42'::numeric)
   ->  Bitmap Heap Scan on takes
         Recheck Cond: ((id)::text = (student.id)::text)
         ->  Bitmap Index Scan on takes_pkey
               Index Cond: ((id)::text = (student.id)::text)
```

Reasoning:

- The query is joining two tables based on a column value and filtering the result set based on a condition on another column.

- Since the condition is selective and the join is performed on indexed columns, the nested loop algorithm is a good choice as it can efficiently perform the join operation and filter the result set using the index.

5. **Query:**

```sql
-- Execution (1)
EXPLAIN ANALYSE CREATE INDEX i1 ON takes(id, semester, year);

-- Execution (2)
EXPLAIN ANALYSE DROP INDEX i1;
```

Evaluation (1) Time: 61.19 ms
Evaluation (2) Time: 11.43 ms

6. **Query:**

```sql
CREATE TABLE takes2 (
    ID VARCHAR(5),
    course_id VARCHAR(8),
    sec_id VARCHAR(8),
    semester VARCHAR(6),
    year NUMERIC(4,0),
    grade VARCHAR(2)
);

INSERT INTO takes2 SELECT * FROM takes;
EXPLAIN ANALYSE INSERT INTO takes2 SELECT * FROM takes;
```

Query plan:

```
                                QUERY PLAN
------------------------------------------------------------
 Insert on takes2  (cost=0.00..520.00 rows=0 width=0) (actual
    time=20.239..20.240 rows=0 loops=1)
   ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000 width
    =24) (actual time=0.024..2.225 rows=30000 loops=1)
 Planning Time: 0.658 ms
 Execution Time: 20.263 ms
```

Execution Time: 20.263 ms

7. **Query:**

```sql
-- Record the start time
SELECT clock_timestamp();

-- Add primary key constraint
ALTER TABLE takes2 ADD CONSTRAINT pk_takes2 PRIMARY KEY (ID,
    course_id, sec_id, semester, year);

-- Record the end time
SELECT clock_timestamp();
```

Execution Time : 94.447 ms
Reasoning:

- Because EXPLAIN ANALYSE doesn't work with ALTER, we used clock_timestamp(), and then computed the difference in time step before and after the execution.

8. **Query:**

```sql
DROP TABLE IF EXISTS takes2;

CREATE TABLE takes2 (
    ID VARCHAR(5),
    course_id VARCHAR(8),
    sec_id VARCHAR(8),
    semester VARCHAR(6),
    year NUMERIC(4,0),
    grade VARCHAR(2)
);

ALTER TABLE takes2 ADD PRIMARY KEY (id, course_id, sec_id,
    semester, year);

EXPLAIN ANALYSE INSERT INTO takes2 SELECT * FROM takes ;
```

Query plan:

```
                              QUERY PLAN
----------------------------------------------------------------
 Insert on takes2  (cost=0.00..520.00 rows=0 width=0) (actual
    time=86.496..86.497 rows=0 loops=1)
   ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000 width
    =24) (actual time=0.014..1.942 rows=30000 loops=1)
 Planning Time: 0.107 ms
 Execution Time: 86.525 ms
```

Execution Time : 86.525 ms

Observation: It takes more time to insert into the takes2 table after adding primary key constraint.

Reasoning:

- Creating and updating an index can take additional time during insertion.
- When we insert a row into a table with a primary key, PostgreSQL needs to validate that the key is unique and not null. This validation process takes time and can slow down the insertion rate.
- Primary keys are a type of constraint that enforce data integrity rules on the table. In addition to checking for uniqueness and nullability, constraints can also check for data types, ranges, and foreign key references. Enforcing these constraints during insertion can add overhead and slow down the process.

9. **Query:**

```sql
SELECT *
FROM student
JOIN takes ON student.ID = takes.ID
ORDER BY student.ID;
```

Query Plan:

```
                          QUERY PLAN
---------------------------------------------------------------

 Merge Join
   Merge Cond: ((student.id)::text = (takes.id)::text)
   ->  Index Scan using student_pkey on student
   ->  Index Scan using takes_pkey on takes
```

Reasoning:

- The query uses a merge join because it sorts both the "student" and "takes" tables by the join column "ID" and then merges them using the sorted order. The merge join algorithm works well when both the tables are already sorted on the join column, which reduces the amount of sorting needed and improves the query performance.
- In this case, the "student" and "takes" tables are sorted by the join column "ID" due to the ORDER BY clause, making the merge join an efficient choice for the query optimizer.

10. **Original Query :**

```sql
-- Original query with limit 10.
SELECT *
FROM student
JOIN takes ON student.ID = takes.ID
ORDER BY student.ID
LIMIT 10;
```

Observation: Join algorithm doesn't change on putting a limit of 10 on rows for the query in the Q9.

New Query:

```sql
-- New query with limit 10.
SELECT *
FROM student
JOIN takes ON student.ID = takes.ID
WHERE student.dept_name = 'Physics'
ORDER BY student.ID
LIMIT 10;
```

Observation: Join algorithm changes from Hash Join when query is run without limits to Nested Loop when query is run with a limit of 10.

Reasoning:

- In the second case because we put a condition on the rows, it reduced the number of rows into consideration.
- Thus Hash joined changed into nested loop on putting a limit on the number of rows, as it can just do nested loop without much overhead.

11. **Query:**

```sql
EXPLAIN ANALYSE SELECT dept_name, AVG(salary)
FROM instructor
GROUP BY dept_name
HAVING COUNT(*) > 1;
```

Query Plan:

```
                                   QUERY PLAN
--------------------------------------------------------------
 HashAggregate
   Group Key: dept_name
   Filter: (count(*) > 1)
   Batches: 1  Memory Usage: 40kB
   Rows Removed by Filter: 3
   ->  Seq Scan on instructor
```

Reasoning: This query uses hash aggregation because it involves grouping by the dept_name column and calculating the average salary for each group. The HAVING COUNT(*) 1 clause filters out the groups that have fewer than two rows.

12. **Query:**

```sql
EXPLAIN SELECT dept_name, COUNT(*) FROM instructor GROUP BY
    dept_name ORDER BY dept_name;
```

Query plan:

```
                                  QUERY PLAN
--------------------------------------------------------------
 GroupAggregate
   Group Key: instructor.dept_name
   ->  Sort
         Sort Key: instructor.dept_name
         ->  Seq Scan on instructor
```

Reasoning: This query uses group aggregation because it groups the results by the dept_name column and performs an aggregate function (COUNT(*)) on each group. The GROUP BY clause specifies that the query should return one row for each unique value in the dept_name column, which is a characteristic of group aggregation.