**Part II. Query Processing and Optimizer**

**Q1.** (1) Execute SQL queries S1 and S2 and provide the query plans for each:

 S1: SELECT \* FROM movie WHERE votes BETWEEN 0 AND 1000;

 S2: SELECT \* FROM movie WHERE votes BETWEEN 0 AND 50000;

(2) Given that S1 and S2 are similar in structure, provide an analysis of why their execution plans may differ. Explain why the two query plans are different.

**Query:**

EXPLAIN SELECT \* FROM movie WHERE votes BETWEEN 0 AND 1000;

EXPLAIN SELECT \* FROM movie WHERE votes BETWEEN 0 AND 50000;

**Output**:

"QUERY PLAN"

"Index Scan using movie\_votes\_idx on movie (cost=0.28..21.88 rows=330 width=41)"

" Index Cond: ((votes >= 0) AND (votes <= 1000))"

"QUERY PLAN"

"Seq Scan on movie (cost=0.00..44.67 rows=1843 width=41)"

" Filter: ((votes >= 0) AND (votes <= 50000))"

**Analysis:**

1. **Difference in Execution Plans:**
   * S1 utilizes an **Index Scan** using the index movie\_votes\_idx on the movie table.
   * S2 uses a **Sequential Scan** (Seq Scan) on the movie table.
2. **Reasons for Different Plans:**
   * **Index Utilization:** S1 benefits from an index scan because the range of votes (0 to 1000) is relatively small. The index movie\_votes\_idx efficiently filters the rows based on the votes column.
   * **Query Selectivity:** The selectivity of S1's condition is higher compared to S2, resulting in fewer rows to scan. This makes an index scan more efficient.
   * **Cost Estimation:** The cost estimation for S1's index scan is lower compared to S2's sequential scan. This indicates that the optimizer perceives the index scan as a more efficient access method for S1.

**Conclusion:**

* The difference in execution plans is primarily due to the selectivity of the votes range condition and the presence of an index on the votes column.
* S1 benefits from an index scan because of its higher selectivity, resulting in a more efficient plan compared to S2's sequential scan.

**Q2.** (1) Execute the SQL queries S3 and S4 and provide the query plans for each.

 S3: SELECT votes FROM movie WHERE votes < 1000;

 S4: SELECT \* FROM movie WHERE votes < 1000;

(2) Discuss the difference between the query plans of S3 and S4, specifically identifying which query benefits from an index-only plan and why.

Query:

EXPLAIN SELECT votes FROM movie WHERE votes < 1000;

EXPLAIN SELECT \* FROM movie WHERE votes < 1000;

Result:

"QUERY PLAN"

"Index Only Scan using movie\_votes\_idx on movie (cost=0.28..14.04 rows=329 width=4)"

" Index Cond: (votes < 1000)"

"QUERY PLAN"

"Index Scan using movie\_votes\_idx on movie (cost=0.28..21.04 rows=329 width=41)"

" Index Cond: (votes < 1000)"

**Discussion:**

1. **Difference in Query Plans:**
   * Both S3 and S4 utilize an **Index Scan** using the index movie\_votes\_idx on the movie table.
   * The main difference lies in the width of the returned data: S3 returns only the votes column, while S4 returns all columns (\*).
2. **Analysis:**
   * **Index Utilization:** Both queries benefit from an index scan due to the condition votes < 1000, which is effectively utilized by the index movie\_votes\_idx.
   * **Index Only Scan vs. Index Scan:** The difference in terminology between "Index Only Scan" and "Index Scan" lies in whether the query requires only columns present in the index or additional columns from the table.
     + S3 utilizes an "Index Only Scan" because it only requires the votes column, which is present in the index. This means that PostgreSQL can directly retrieve the required column values from the index without accessing the table rows.
     + S4 utilizes an "Index Scan" because it requires all columns (\*), some of which may not be present in the index. PostgreSQL needs to access both the index and the table to retrieve all the required columns.

**Conclusion:**

* Both S3 and S4 benefit from index scans due to the selectivity of the condition votes < 1000.
* S3 utilizes an "Index Only Scan" because it only needs the votes column, resulting in a more efficient plan compared to S4's "Index Scan" where additional columns from the table are required.

**Q3.** (1) Execute the SQL queries Q5, Q6 and Q7 and provide the query plan for each.

 S5: SELECT name FROM actor WHERE name like 'W%';

 S6: SELECT name FROM actor WHERE substr(name, 1, 1) = 'W';

 S7: SELECT name FROM actor WHERE name LIKE '%w';

(2) S5 and S6 yield identical results; however, their query plans differ. Explain the reasons behind the differing query plans.

(3) Delve into why the query plan for S7 does not leverage an available index, explaining the possible rational behind this decision.

Query:

EXPLAIN SELECT name FROM actor WHERE name like 'W%';

EXPLAIN SELECT name FROM actor WHERE substr(name, 1, 1) = 'W';

EXPLAIN SELECT name FROM actor WHERE name LIKE '%w';

Result:

"QUERY PLAN"

"Seq Scan on actor (cost=0.00..111.31 rows=120 width=14)"

" Filter: ((name)::text ~~ 'W%'::text)"

"QUERY PLAN"

"Seq Scan on actor (cost=0.00..126.17 rows=30 width=14)"

" Filter: (substr((name)::text, 1, 1) = 'W'::text)"

"QUERY PLAN"

"Seq Scan on actor (cost=0.00..111.31 rows=60 width=14)"

" Filter: ((name)::text ~~ '%w'::text)"

**Discussion:**

1. **Difference in Query Plans for S5 and S6:**
   * Both S5 and S6 utilize a **Sequential Scan** (Seq Scan) on the actor table.
   * The filter conditions differ:
     + S5 uses the name LIKE 'W%' pattern match.
     + S6 uses the substr(name, 1, 1) = 'W' condition to compare the first character.
2. **Reasons for Different Query Plans:**
   * Despite both queries having similar intentions (matching names starting with 'W'), they differ in implementation:
     + S5 directly uses the LIKE pattern match.
     + S6 uses the substr() function to extract the first character and compares it to 'W'.
   * PostgreSQL's query planner may choose different optimization strategies for pattern matches and function calls, leading to different plans.
3. **Query Plan for S7:**
   * S7 also utilizes a Sequential Scan with a filter condition of name LIKE '%w'.
   * Despite the availability of an index on the name column, the optimizer opts for a Sequential Scan.
   * Possible reasons for not leveraging the index include:
     + Low selectivity of the pattern match %w, making the use of an index less efficient.
     + PostgreSQL's optimizer may decide that a Sequential Scan is more efficient due to factors like table size or other considerations.

**Conclusion:**

* The query plans for S5, S6, and S7 all utilize Sequential Scans with different filter conditions.
* While S5 and S6 have similar intentions, their query plans differ due to differences in implementation.
* S7's query plan also utilizes a Sequential Scan, likely due to factors such as the low selectivity of the pattern match or the optimizer's decision for efficiency.

**Q4.** (1) Execute the SQL query S8 and present its query plan.

 S8: SELECT a.name FROM actor a, movie m, casting c

WHERE m.title = 'Star Wars' and m.id=c.movieid and a.id=c.actorid

(2) Identify the join algorithms used in this query and provide concise explanations of each algorithm mentioned, highlighting their use cases and operational mechanism.

Query:

EXPLAIN SELECT a.name FROM actor a, movie m, casting c WHERE m.title = 'Star Wars' and m.id=c.movieid and a.id=c.actorid;

Result:

"QUERY PLAN"

" -> Nested Loop (cost=0.56..12.83 rows=9 width=4)"

" -> Index Scan using movie\_title\_idx on movie m (cost=0.28..8.29 rows=1 width=4)"

" Index Cond: ((title)::text = 'Star Wars'::text)"

" -> Index Only Scan using casting\_pk on casting c (cost=0.29..4.44 rows=9 width=8)"

" Index Cond: (movieid = m.id)"

" -> Index Scan using actor\_pk on actor a (cost=0.28..0.31 rows=1 width=18)"

" Index Cond: (id = c.actorid)"

**Analysis:**

1. **Nested Loop Join:**
   * **Operational Mechanism:** The Nested Loop Join algorithm iterates over each row from the outer relation and, for each row, searches for matching rows in the inner relation. It combines rows from both relations where the join condition is satisfied.
   * **Use Cases:** Nested Loop Join is suitable for small to moderate-sized tables or when there are effective indexes available for joining.
   * **In S8:** The Nested Loop Join is used to join the movie table (m) with the casting table (c) using the condition movieid = m.id. The result of this join is then further joined with the actor table (a) using the condition id = c.actorid.
2. **Index Scan:**
   * **Operational Mechanism:** Index Scan retrieves rows from a table using an index. It scans the index to find matching rows based on the given conditions.
   * **Use Cases:** Index Scan is used when there are effective indexes available for filtering rows based on the conditions.
   * **In S8:** Index Scan is used for the movie table (m) to find the row with the title 'Star Wars' and for the actor table (a) to find rows based on the actor IDs (id).

**Conclusion:**

* The query plan for S8 uses a Nested Loop Join algorithm to join the movie and casting tables first, and then the result of this join is joined with the actor table.
* Index Scans are utilized to efficiently retrieve rows from the movie and actor tables based on the given conditions.
* The Nested Loop Join algorithm is chosen in this scenario due to its suitability for small to moderate-sized tables and the presence of effective indexes for joining.

**Q5.** (1) Execute the SQL queries Q9 and Q10 and present the query plan for each.

S9: SELECT m.title FROM movie m WHERE m.title= 'Scrooge'

S10: SELECT /\*+ FULL(m)\*/ m.title FROM movie m WHERE m.title= 'Scrooge'

(2) Despite S9 and S10 generating the same result set, their query plans are different. Analyze and explain the differences between these plans, focusing on the underlying factors that contribute this difference.

Query:

EXPLAIN SELECT m.title FROM movie m WHERE m.title= 'Scrooge';

EXPLAIN SELECT /\*+ FULL(m)\*/ m.title FROM movie m WHERE m.title= 'Scrooge';

Result:

"QUERY PLAN"

"Index Only Scan using movie\_title\_idx on movie m (cost=0.28..4.29 rows=1 width=16)"

" Index Cond: (title = 'Scrooge'::text)"

"QUERY PLAN"

"Index Only Scan using movie\_title\_idx on movie m (cost=0.28..4.29 rows=1 width=16)"

" Index Cond: (title = 'Scrooge'::text)"

**Analysis:**

1. **Index Only Scan:**
   * Both query plans employ an Index Only Scan, which means PostgreSQL retrieves the necessary data directly from the index without accessing the table itself.
   * This scan is more efficient than a full table scan because it only accesses the index, which typically contains fewer pages and requires less I/O operations.
2. **Query Plan Similarity:**
   * Both queries generate the same query plan, indicating that PostgreSQL's optimizer has chosen the same efficient execution path for both queries despite the differences in query syntax (/\*+ FULL(m)\*/ hint in S10).

**Conclusion:**

* Despite any differences in query syntax or hints, the PostgreSQL optimizer has determined that an Index Only Scan is the most efficient method to retrieve rows where the title is 'Scrooge' from the movie table.
* The similarity in query plans suggests that the presence of the /\*+ FULL(m)\*/ hint in S10 did not influence the optimizer's decision, as it still chose the most efficient access method based on the available indexes.