|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Database Management System Laboratory** | **Semester:** | **IV** |
| **Date of Performance:** | **18 / 04 / 2023** | **Batch No:** | **A – 2** |
| **Faculty Name:** | **Prof. Shila Dhande** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade / Marks:** | **\_\_\_ / 25** |

**Experiment No.: 9**

**Title: Indexing and Query Processing**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| **Objective:**   * Implementing indexing and query processing. * To understand Query Processing and implement indexing to improve query execution plans. |

|  |
| --- |
| **COs to be achieved:** |
| **CO3:** Use SQL for Relational database creation, maintenance and query processing. |

|  |
| --- |
| **Books / Journals / Websites Referred:** |
| 1. G. K. Gupta:” Database Management Systems”, McGraw – Hill 2. Korth, Slberchatz, Sudarshan: “Database Systems Concept”, 6th Edition, McGraw Hill 3. Elmasri and Navathe, “Fundamentals of Database Systems”, 5th Edition, PEARSON Education. |

|  |
| --- |
| **Tools Required:** |
| * Postgresql Software |

|  |
| --- |
| **Theory:** |
| A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.  While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.  To add an index for a column or a set of columns, you use the CREATE INDEX statement as follows:  CREATE INDEX index\_name ON table\_name (column\_list)  **Query Life Cycle:**  https://miro.medium.com/max/341/1*8alwVGt6Ph2CB4bzxFX9eA.png  **Planner and Executor:**  The planner receives a query tree from the rewriter and generates a (query) plan tree that can be processed by the executor most effectively.  The planner in Database is based on pure cost-based optimization.  **Explain Command:**  This command displays the execution plan that the PostgreSQL/MySQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.  As in the other RDBMS, the EXPLAIN command in Database displays the plan tree itself. A specific example is shown below –  Database: testdb=#   1. EXPLAIN SELECT \* FROM tbl\_a WHERE id < 300 ORDER BY data; 2. QUERY PLAN 3. --------------------------------------------------------------- 4. Sort (cost=182.34..183.09 rows=300 width=8) 5. Sort Key: data    * Seq Scan on tbl\_a (cost=0.00..170.00 rows=300 width=8) 6. Filter: (id < 300) 7. (4 rows)   A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command in PostgreSQL –  Fig. 3.5. A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command.  **Nodes:**  The first thing to understand is that each indented block with a preceding “->” (along with the top line) is called a node. A node is a logical unit of work (a “step” if you will) with an associated cost and execution time. The costs and times presented at each node are cumulative and roll up all child nodes.  **Cost:**  It is not the time but a concept designed to estimate the cost of an operation. The first number is start-up cost (cost to retrieve first record) and the second number is the cost incurred to process entire node (total cost from start to finish).  Cost is a combination of 5 work components used to estimate the work required: sequential fetch, non-sequential (random) fetch, processing of row, processing operator (function), and processing index entry.  **Rows:**  Rows are the approximate number of rows returned when a specified operation is performed.  (In the case of select with where clause rows returned is  Rows = cardinality of relation \* selectivity )  Width is an average size of one row in bytes.  **Explain Analyze Command:**  The EXPLAIN ANALYZE option causes the statement to be actually executed, not only planned. Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality.  e.g. EXPLAIN (ANALYZE) SELECT \* FROM foo;  http://codingsight.com/wp-content/uploads/2017/10/query-plan2.png  The command displays the following additional parameters:   * **actual time** is the actual time in milliseconds spent to get the first row and all rows, respectively. * **rows** is the actual number of rows received with Seq Scan. * **loops** is the number of times the Seq Scan operation had to be performed. * **total runtime** is the total time of query execution.   Query plans for select with where clause can be sequential scan, Index Scan, Index only Scan, Bitmap Index Scan etc.  Query plans for joins are Nested loop join, Hash join, Merge join etc. |

|  |
| --- |
| **Implementation Details (Problem Statement, Query and Screenshots of Results):** |
| **Comprehend how indexes improves the performance of query applied for your database. Demonstrate for the following types of queries on your database**   1. **Simple select query** 2. **Select query with where clause** 3. **Select query with order by query** 4. **Select query with JOIN** 5. **Select query with aggregation**   Creating test table –    Inserting random values in table –    Query plan for SELECT query using EXPLAIN –      Inserting 50 values –    Query plan for SELECT query using EXPLAIN after inserting value –    Analyzing query performance along with execution time using EXPLAIN (ANALYZE) with the SELECT query –    Creating index –    Query plan for a SELECT query with a WHERE clause using the indexed column c1 –    Analyzing query performance along with execution time for the SELECT query using the indexed column c1 –    Disable sequential scan to force the usage of indexes –      Re-enable sequential scan –    Query plan for WHERE query –    Query plan for WHERE, AND, LIKE query –    Query plan when selecting only the indexed fields –    **Query with ORDER BY**  Drop the existing index on c1, then explain the query plan for a SELECT query with ORDER BY clause –    Recreating index and explaining –    **Query with JOIN**  Creating new table and inserting values –    Analyzing table and explaining –    **Multicolumn Indexing**  Creating table, inserting values and analyzing items –    Query plan for simple query without indexes –    Adding multicolumn index and explaining the query plan –    Create an index on the product\_id column only –    Adding price into the index –    **SQL Query Code**  CREATE TABLE TestTable (  c1 integer,  c2 text  );  INSERT INTO TestTable  SELECT i, md5(random()::text)  FROM generate\_series(1, 120120) AS i;  EXPLAIN SELECT \* FROM TestTable;  INSERT INTO TestTable  SELECT i, md5(random()::text)  FROM generate\_series(1, 50) AS i;  EXPLAIN SELECT \* FROM TestTable;  EXPLAIN (ANALYZE) SELECT \* FROM TestTable;  CREATE INDEX ON TestTable(c1);  EXPLAIN SELECT \* FROM TestTable WHERE c1 > 500;  EXPLAIN (ANALYZE) SELECT \* FROM TestTable WHERE c1 > 500;  SET enable\_seqscan TO off;  EXPLAIN (ANALYZE) SELECT \* FROM TestTable WHERE c1 > 500;  RESET enable\_seqscan;  EXPLAIN SELECT \* FROM TestTable WHERE c1 < 500;  EXPLAIN SELECT \* FROM TestTable WHERE c1 < 500 AND c2 LIKE 'abcd%';  EXPLAIN SELECT c1 FROM TestTable WHERE c1 < 500;  DROP INDEX TestTable\_c1\_idx;  EXPLAIN (ANALYZE) SELECT \* FROM TestTable ORDER BY c1;  CREATE INDEX ON TestTable(c1);  EXPLAIN (ANALYZE) SELECT \* FROM TestTable ORDER BY c1;  CREATE TABLE bar (  c1 integer,  c2 boolean  );  INSERT INTO bar  SELECT i, i%2=1  FROM generate\_series(1, 500000) AS i;  ANALYZE bar;  EXPLAIN (ANALYZE, BUFFERS, VERBOSE)  SELECT \* FROM TestTable JOIN bar ON TestTable.c1 = bar.c1;  CREATE TABLE line\_items (  item\_id serial PRIMARY KEY,  product\_id integer,  price numeric  );  INSERT INTO line\_items (product\_id, price)  SELECT i, random() \* 100  FROM generate\_series(1, 100000) AS i;  ANALYZE line\_items;  EXPLAIN ANALYZE SELECT \* FROM line\_items WHERE product\_id > 80 GROUP BY 1;  CREATE INDEX items\_product\_id\_price ON line\_items(product\_id, price);  EXPLAIN (ANALYZE) SELECT \* FROM line\_items WHERE product\_id > 80 GROUP BY 1;  CREATE INDEX items\_product\_id ON line\_items(product\_id);  EXPLAIN (ANALYZE) SELECT \* FROM line\_items WHERE product\_id > 80 GROUP BY 1;  EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT \* FROM line\_items WHERE product\_id > 80 GROUP BY 1; |

|  |
| --- |
| **Post Lab Subjective / Objective Type Questions:** |
| 1. **Illustrate with an example Heuristic based query optimization with suitable example.**   Heuristic-based query optimization is a technique used in database management systems to improve the performance of query processing by making educated guesses or using rules of thumb to optimize query execution plans. Instead of exhaustively searching through all possible query plans, which can be computationally expensive, heuristic-based optimization relies on heuristics or guidelines to quickly find a satisfactory plan.  Consider a scenario where we have a database table called "Orders" containing information about orders placed by customers. The table has the following structure:   * OrderID (Primary Key) * CustomerID * OrderDate * TotalAmount   Creating the table –    Now, let's say we have a query to retrieve orders placed by a specific customer within a certain date range, sorted by the order date:    To optimize this query using heuristic-based optimization, we can consider the following heuristics:   1. **Selectivity Estimation**: Estimate the selectivity of each predicate in the WHERE clause. In this case, estimating the number of orders placed by customer '123' within the date range. 2. **Index Selection**: Determine if indexes exist on the columns involved in the query predicates (CustomerID, OrderDate). If indexes exist, consider using them to speed up data retrieval. 3. **Join Order Optimization**: Since there's no join in this query, this heuristic is not applicable. 4. **Cost-Based Estimation**: Estimate the cost of different execution plans based on factors like data distribution, index selectivity, and disk I/O. Choose the plan with the lowest estimated cost.   Based on these heuristics, we can optimize the query as follows:    In this optimized version:   * We create indexes on the CustomerID and OrderDate columns to speed up data retrieval. * The query plan will likely utilize these indexes to efficiently locate orders for the specified customer within the date range. * By leveraging heuristics, we improve query performance without the need for exhaustive cost calculations. * This example demonstrates how heuristic-based query optimization can enhance the efficiency of database queries by making informed decisions based on rules of thumb and common sense rather than exhaustive analysis. |

|  |
| --- |
| **Conclusion:** |
| In this experiment, we analyzed query performance using EXPLAIN and EXPLAIN ANALYZE to optimize execution plans. By leveraging indexing and heuristic-based optimization, we enhanced query efficiency and observed the impact of various optimization techniques on database performance. |

**Signature of faculty in-charge with Date:**