Automatic Index Creation

Saksham Rathi, Kavya Gupta, Shravan S, Mayank Kumar
(22B1003) (22B1053) (22B1054) (22B0933)

CS349: DataBase and Information Systems
Under Prof. Sudarshan and Prof. Suraj

Indian Institute of Technology Bombay
Spring 2024-25

Contents

Introduction to the Problem Statement

- Indexes are crucial for efficient query execution in relational databases.
- However, developers sometimes forget to create indexes for frequently queried columns.
- This can lead to repeated full relation scans, significantly degrading performance.
- **Goal:** Modify the application layer of PostgreSQL to detect such patterns and automatically create indexes when beneficial.
- Approach:
  - Track full relation scans with equality predicates.
  - Estimate the potential benefit of an index.
  - Automatically trigger index creation if estimated benefit outweighs the cost.
  - Rejecting low selectivity columns, such as gender, which has low number of distinct values.

Directory Structure

Here is the directory structure of the submission:

- `./code`: Contains the header and C++ files for the implementation, along with the Makefile.
- `./theory`: Contains some relevant paper and slides.
- `./documentation`: Contains the report as `readme.pdf`.
- `./README.md`: Contains the instructions to run the code.

About Indices

An index in SQL is a database object that improves the speed of data retrieval operations on a database table.

When a query is executed, the database can use the index to quickly find the relevant rows.

Without an index, the database might need to scan every row to find the data, which is much slower.

When to use indices?

- **Frequent searches on specific columns:** Columns that are often used in WHERE clauses, JOIN conditions or as part of a SELECT query.
- **Large Tables with Heavy Read Operations:** Tables with a vast number of records where read operations are more common than write operations.
- **Columns used in JOINs:** Indexing these columns can speed up the join process.
- **Unique or Primary Key Constraints:** Indices improve lookup efficiency, so easy to impose such constraints.
- **Composite Indices:** When queries often filter on multiple columns, a composite index can be beneficial, rather than creating separate indices for each column.

When to use indices?

There are also cases, where we should refrain from using indices, such as tables with heavy write operations, because indices slow down INSERT, UPDATE, and DELETE operations (index needs to be updated too). Similarly, in case of small tables, or columns with low selectivity (many duplicate values).

Indices, overall lead to improved query performance, slower write opterations, and increased storage requirements.

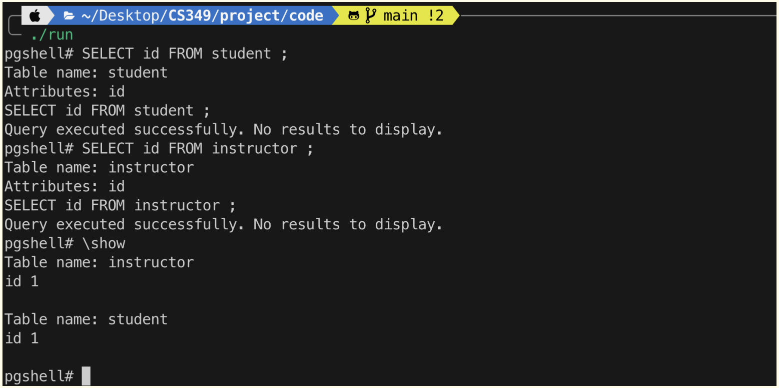
We can analyze how a query is execueted, and whether an index is effectively used or not by using the `EXPLAIN` command in PostgreSQL. Moreover, to maintain performance, expecially in databases with frequent data modifications, we need to regularly rebuild and reorganize indices.

Calculating the frequency of relations and their attributes

For Stage 1 of this project, we made a simple parser, which gives us the list of relations, and the attributes involved in some query. This helps us know the number of times, a particular attribute of a relation is accessed, this in turn can be used to make the decision on whether we should construct an index on that attribute.
For a sample run, the `\show` command can be used to display the frequency results.

Results

Here is the sample output:



```
 ~/Desktop/CS349/project/code   main !2
  ./run
pgshell# SELECT id FROM student ;
Table name: student
Attributes: id
SELECT id FROM student ;
Query executed successfully. No results to display.
pgshell# SELECT id FROM instructor ;
Table name: instructor
Attributes: id
SELECT id FROM instructor ;
Query executed successfully. No results to display.
pgshell# \show
Table name: instructor
id 1

Table name: student
id 1

pgshell#
```

Figure: Displaying the output of show command

An Auto-Indexing Technique for Databases Based on Clustering

- Automate the physical design so that the task of the database administrator (DBA) is minimized.
- The first category is external tools which use linear programming optimization techniques and other cost minimization techniques to solve the Index Selection Problem.
- The second category is the tools that utilize the query optimizer to give cost estimates for various index configurations and suggest a configuration with the least cost estimation.
- In this technique the optimizer is invoked only once for each query in the workload to choose the final set of indexes from a set of externally determined index configurations.
-

Identifying Candidate Indexes

- A query attribute matrix is created.
- The presence of an indexable attribute is created by 1 and absence by a 0.
- The condition applied is:
  `Freq > threshold1 OR Freq * T > threshold2`
- `Freq` is the frequency of each indexable attribute in the workload and `T` is proportional to the size of the table in rows to which the column belongs.
- Weights of 3, 2, 1 are given to the columns occurring in a `WHERE` clause, `GROUP BY` or `ORDER BY` clauses and aggregate functions, respectively.
- During the clustering phase queries that are similar based on common and frequently occurring attributes are clustered together.

Candidate index suggestion

- During this phase, those candidate indexable attributes which are common to all the queries clustered together during the clustering phase are suggested as indexes.
- The optimizer uses its statistics and cost estimates to choose indexes for each query.
- Those indexes not being picked up by the optimizer are dropped because the presence of these unused indexes will cause an overhead of space and maintenance in the database.

Thank You