

# 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`.
- `./images`: Contains the results and sample runs.
- `./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.

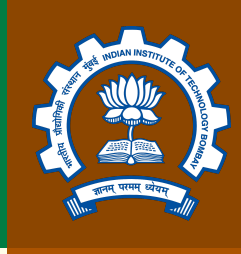
# When to use indices?

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 executed, 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.

# 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.
- All other details can be found in the next sections.

# Goals of the project



- 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 [fNN23].
- **Another Goal** was to understand and implement the paper *"An Auto-Indexing Technique for Databases Based on Clustering"* [ZSG04].

# What We Implemented From User Perspective


- We implemented an interface that can take and submit queries from users as usual as well as automatically create (and remove) indices appropriately, thereby improving performance without any user intervention.

# What All Functionalities We Implemented



- Developed a standalone C++ tool that takes SQL queries as input and performs real-time analysis.
- Implemented policy from the paper *"An Auto-Indexing Technique for Databases Based on Clustering"* [ZSG04].
- The tool tracks attribute access frequencies and cost, and forks a background process to decide on index creation.
- Index creation is not based on fixed thresholds alone:
  - It also invokes the PostgreSQL query planner to compare costs of executing the current query for different candidate indices.
  - Index is created only if the cost savings are significant.
- Also integrated removal of indices using 2 policies.

# How We Implemented It – Part 1

- Used the `pqxx` C++ library to connect and interact with PostgreSQL databases.
- Integrated the `sqlparse` Python library to parse complex SQL queries and extract attribute-level access details (we ignore the attribiutes in the `SELECT` clause).
- Designed custom data structures to:
  - Maintain per-query attribute access statistics.
  - Track information about the existing (our tool made) indices.
- Queries are handled online (i.e., one at a time), so query clustering was not required, unlike in batch-based approaches.

# How We Implemented It – Part 2

- **Candidate attribute selection** is based on the following condition:

$$\texttt{Freq} > \texttt{threshold}_1 \quad \textbf{OR} \quad \texttt{Freq} \times T > \texttt{threshold}_2$$

where:

  - `Freq` = weighted frequency of attribute usage in past queries.
  - `T` = number of rows in the table containing that attribute.

We fixed $\texttt{threshold}_1$ to be $10$. We took $\texttt{threshold}_2$ as the average of the size of all tables in the database (we change it every $50^{th}$ iteration).

- **Weighted frequency computation**:
  - Weight = 3 if attribute is in `WHERE` clause.
  - Weight = 2 if in `GROUP BY` or `ORDER BY`.
  - Weight = 1 if used in an aggregate function (e.g., `SUM`, `COUNT`).

This helps prioritize attributes more critical to query performance.

# How We Implemented It – Part 3



- Once candidate indexable attributes are identified (via frequency and weights), we use the `hypopg` extension for final selection.
- `hypopg` allows us to:
  - Create **hypothetical indexes** without modifying the database.
  - Run the SQL query planner with these indexes as if they existed.
  - Retrieve the estimated query execution cost from the planner.
- This approach enables us to:
  - Leverage PostgreSQL's internal **statistics and heuristics**.
  - Avoid wasting resources on ineffective indexes.
- The top 50% lowest cost of the candidate indices become the final indices and index is created for them in a child process (if not already).
- Apart from this, we have also added an `up arrow` feature which helps user to naviage to the previous queries.

# How We Implemented It - Index Eviction



We have implemented two policies for index eviction:

- **Policy P1: Time-based Eviction**
  Indices that are older than a threshold (5 discrete events) are removed from the list and dropped from the database. This ensures that short-lived, potentially less useful indices are cleaned up promptly.
- **Policy P2: Usage-based Eviction**
  Indices are evicted if their age (current_timestamp - create_time) is more than four times the number of accesses. This removes infrequently used indices that have become stale, balancing age and usage.

# Results

```
mknined@expectnothing:~/Desktop/Database/CS349-Project/code$ ./run
pgshell# explain analyse SELECT movie_id, avg_rating FROM ratings WHERE avg_rating = 8.5;
ratings
2
QUERY PLAN
--------------
Seq Scan on ratings  (cost=0.00..158.96 rows=19 width=16) (actual time=0.027..1.340 rows=19 loops=1)
  Filter: (avg_rating = 8.5)
  Rows Removed by Filter: 7978
Planning Time: 0.093 ms
Execution Time: 1.353 ms
pgshell# explain analyse SELECT movie_id, avg_rating FROM ratings WHERE avg_rating = 8.5;
ratings
2
QUERY PLAN
--------------
Seq Scan on ratings  (cost=0.00..158.96 rows=19 width=16) (actual time=0.032..1.174 rows=19 loops=1)
  Filter: (avg_rating = 8.5)
  Rows Removed by Filter: 7978
Planning Time: 0.069 ms
Execution Time: 1.191 ms
pgshell# Query executed successfully. No results to display.
Index (ratings2) created for ratings(avg_rating)
explain analyse SELECT movie_id, avg_rating FROM ratings WHERE avg_rating = 8.5;
ratings
2
QUERY PLAN
--------------
Bitmap Heap Scan on ratings  (cost=4.43..45.29 rows=19 width=16) (actual time=0.031..0.050 rows=19 loops=1)
  Recheck Cond: (avg_rating = 8.5)
  Heap Blocks: exact=16
  ->  Bitmap Index Scan on ratings2  (cost=0.00..4.42 rows=19 width=0) (actual time=0.022..0.022 rows=19 loops=1)
        Index Cond: (avg_rating = 8.5)
Planning Time: 0.202 ms
Execution Time: 0.068 ms
pgshell#
```

Figure: A sample run

# Results

```
mknined@expectnothing:~/Desktop/Database/CS349-Project/code$ ./run
pgshell# explain analyse SELECT title FROM movie WHERE country = 'USA';
movie
2
QUERY PLAN
--------------
Seq Scan on movie  (cost=0.00..213.96 rows=2260 width=15) (actual time=0.005..1.252 rows=2260 loops=1)
  Filter: ((country)::text = 'USA'::text)
  Rows Removed by Filter: 5737
Planning Time: 0.118 ms
Execution Time: 1.355 ms
pgshell# explain analyse SELECT title FROM movie WHERE country = 'USA';
movie
2
QUERY PLAN
--------------
Seq Scan on movie  (cost=0.00..213.96 rows=2260 width=15) (actual time=0.014..1.249 rows=2260 loops=1)
  Filter: ((country)::text = 'USA'::text)
  Rows Removed by Filter: 5737
Planning Time: 0.084 ms
Execution Time: 1.364 ms
pgshell# Query executed successfully. No results to display.
Index (movie2) created for movie(country)
explain analyse SELECT title FROM movie WHERE country = 'USA';
movie
2
QUERY PLAN
--------------
Bitmap Heap Scan on movie  (cost=33.80..176.05 rows=2260 width=15) (actual time=0.114..0.592 rows=2260 loops=1)
  Recheck Cond: ((country)::text = 'USA'::text)
  Heap Blocks: exact=114
  ->  Bitmap Index Scan on movie2  (cost=0.00..33.23 rows=2260 width=0) (actual time=0.088..0.088 rows=2260 loops=1)
        Index Cond: ((country)::text = 'USA'::text)
Planning Time: 0.194 ms
Execution Time: 0.688 ms
pgshell#
```

Figure: A sample run

# Results

```
mknined@expectnothing:~/Desktop/Database/CS349-Project/code$ ./run
pgshell# explain analyse SELECT production_company FROM movie WHERE id = 'tt9876543';
movie
2
QUERY PLAN
--------------
Index Scan using movie_pkey on movie  (cost=0.28..8.30 rows=1 width=18) (actual time=0.026..0.026 rows=0 loops=1)
  Index Cond: ((id)::text = 'tt9876543'::text)
Planning Time: 0.102 ms
Execution Time: 0.037 ms
pgshell# explain analyse SELECT production_company FROM movie WHERE id = 'tt9876543';
movie
2
QUERY PLAN
--------------
Index Scan using movie_pkey on movie  (cost=0.28..8.30 rows=1 width=18) (actual time=0.020..0.020 rows=0 loops=1)
  Index Cond: ((id)::text = 'tt9876543'::text)
Planning Time: 0.081 ms
Execution Time: 0.036 ms
pgshell# Query executed successfully. No results to display.
Index (movie2) created for movie(id)
explain analyse SELECT production_company FROM movie WHERE id = 'tt9876543';
movie
2
QUERY PLAN
--------------
Index Scan using movie2 on movie  (cost=0.28..8.30 rows=1 width=18) (actual time=0.022..0.022 rows=0 loops=1)
  Index Cond: ((id)::text = 'tt9876543'::text)
Planning Time: 0.181 ms
Execution Time: 0.036 ms
pgshell#
```

Figure: A sample run

# Information about the Dataset



## ImDB Database Overview

| Table Name | Approx. Row Count | Columns |
|---|---|---|
| names | 25735 | id, name, height, date_of_birth, known_for_movies |
| role_mapping | 15615 | movie_id, name_id, category |
| genre | 14662 | movie_id, genre |
| movie | 7997 | id, title, year, date_published, duration, country, worlwide_gross_income, languages, production_company |
| ratings | 7997 | movie_id, avg_rating, total_votes, median_rating |
| director_mapping | 3867 | movie_id, name_id |

# Conclusion and Future Work

- We developed a real-time auto-indexing system that:
  - Tracks attribute access patterns and frequencies.
  - Applies a cost-aware filtering mechanism using PostgreSQL's planner via `hypopg`.
  - Automatically creates and removes indexes using adaptive policies.

- **Future Work:**
  - Extend the system to handle batch workloads.
  - Incorporate clustering of similar queries to identify shared indexable patterns.
  - Explore reinforcement learning or predictive models for smarter index management.
  - Enhance the parser for better weighing mechanisms.

# References


Nagarjun Nagesh.
When and how to create indexes in a sql database.
https://medium.com/@nagarjun_nagesh/when-and-how-to-create-indexes-in-a-sql-database-445d8fc59b09, 2023.

M. Zaman, J. Surabattula, and L. Gruenwald.
An auto-indexing technique for databases based on clustering.
In *Proceedings. 15th International Workshop on Database and Expert Systems Applications, 2004.*, pages 776–780, 2004.

# Code and Report



The code and the report can be found at the following link:

`https://github.com/sakshamrathi21/CS349-Project`




# Thank You