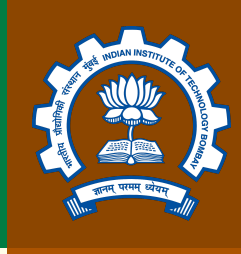

Automatic Index Creation

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

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.
- 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).

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.

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.