CS349 Project Report
Automatic Creation of Indices in PostgreSQL

Indian Institute of Technology, Bombay

Ananya Kulashreshtha : 22B0906
Navnit Kumar Thakur : 22B0936
Saksham Raj : 22B0964
Tanish Agarwal : 22B0978

May 4, 2025

GitHub Link

https://github.com/Tanish1e9/Auto_indexer

Introduction

Modern databases optimize query execution using indexes, but building all indexes preemptively can be inefficient. Our PostgreSQL extension, `auto_index`, automatically identifies promising candidates for indexing by analyzing sequential scans during query planning and creates an index on them.

Goals of the Project

- **Track Missing Indexes:** Modify PostgreSQL to note when an index could help a query.
- **Collect metrics:** Record these instances to gauge overall benefit.
- **Compare benefit to cost:** Check if saved query time multiplied by frequency outweighs index creation cost.
- **Index Creation:** Use a separate process to build the index when it makes sense.

Key Concepts

1. Planner Hooks

PostgreSQL provides **hooks** to allow developers to inject custom logic into internal processes. Our extension leverages the `planner_hook` to examine query execution plans. When a query is being planned, our custom hook intercepts the process, analyzes the plan tree, and identifies sequential scans that operate on user tables.
In this way we keep track of the tables with their attributes which are candidates for indexing. When the condition **(benefit $*$ num_queries $>$ cost of index creation)** gets true, we call a background worker which actually creates the index. (We need to do actual index creation in a separate process because index creation is expensive.)

```
static PlannedStmt *auto_index_planner_hook(Query *parse, const char *query_string,
                                            int cursorOptions, ParamListInfo boundParams);
```

2. Background Workers (On-Demand Invocation)

PostgreSQL supports **background workers** as auxiliary processes to run tasks independently from client sessions.

When our planner hook detects a sequential scan that meets our predefined cost-benefit criteria, it immediately triggers the background worker by calling the function `start_auto_index_worker`. The worker then connects to the database, executes the necessary SQL command via PostgreSQL's SPI, and creates the appropriate index. The worker's main function is:

```
PGDLLEXPORT void auto_index_worker_main(Datum main_arg);
```

This design minimizes unnecessary polling overhead, since the background worker is only invoked when there is a need to actually create the index.

Overall Execution Process

When a query is issued to the PostgreSQL database, the full execution cycle proceeds as outlined below:

1. **Query Reception and Parsing:** The query is parsed and rewritten into a parse tree.
2. **Query Planning:**
   - Custom planner hook `auto_index_planner_hook` is invoked during the planning phase.
   - It first calls the previous planner hook (or the standard planner) to produce a complete execution plan (a `PlannedStmt` structure) with all standard optimizations.
3. **Plan Analysis:**
   - After the standard plan is generated, our hook traverses the plan to detect sequential scans and extracts column information.
   - The gathered statistics are recorded, and if a column becomes a strong candidate for indexing, an asynchronous background worker is triggered.
4. **On-Demand Background Worker Invocation:**
   - The worker is launched on demand (via `start_auto_index_worker`) and runs in parallel.
   - Creates an index on the identified table-column pair using PostgreSQL's SPI.
5. **Query Execution:**
   - With the complete query plan in hand, the PostgreSQL executor initializes its state and begins executing the plan.
   - Each node in the plan (e.g., `SeqScan`, join nodes, etc.) is processed in sequence.
   - The results are produced and returned to the client.

Thus, the standard query is executed fully using the complete plan, while the background worker operates asynchronously to create an index if the accumulated statistics indicate a net performance benefit.

Running Instructions

The following steps describe how to integrate the `auto_index` extension into your PostgreSQL installation.

1. **Adding the extension:**

- Copy the `auto_index` folder into the contrib folder of postgres.
- Write auto_index in `shared_preload_libraries` in the `postgresql.conf` file of the database.

2. **Compilation and Installation**
   - Open a terminal and navigate to the `auto_index` folder.
   - Run `make` to compile the extension and `sudo make install` to install the extension.
3. **Initialize the Extension**
   - Everytime to initialize the extension, just run `CREATE EXTENSION auto_index` in the psql terminal which will set up the `aidx_queries` table.
4. **Cleanup**
   - To disable the extension, execute: `DROP EXTENSION auto_index`. This cleans up allocated resources and the aidx_queries table.

Difficulties we faced

We faced some challenges which were primarily because we were new to Postgresql internals.

- **Extension Initialization & cleanup**:- We got to know that PG_init does not always get called on CREATE and for DROP, even if we created a function for cleanup that would itself get deleted everytime. We fixed it using this article.
- **Setting transaction and Snapshot context**:- Refer this. Basically in case of workers, we have to set these manually as opposed to Postgres automatically doing it for backend processes.

Modifications to the PostgreSQL code

We have only made a folder auto_index in the contrib folder of postgres. It's recommended to clone the official mirror code from here to test the extension.

Implementation from user perspective

From an end-user standpoint, the extension provides:

- **Automatic Detection:** Monitors executed queries in real time and identifies candidates for indexing.
- **Self-service Indexing:** Automatically issues `CREATE INDEX` commands when thresholds are met, without interrupting active sessions.
- **Custom GUC variable:** User can set `auto_index.enable` on and off to temporarily enable and disable the extension.

Architecture of the Solution

1. **Planner Hook Integration**
   - Our extension leverages the `planner_hook` to examine query execution plans.
   - When a query is being planned, our custom hook intercepts the process, analyzes the plan tree, and identifies sequential scans that operate on user tables.
   - When the condition **(benefit $*$ num_queries $>$ cost of index creation)** gets true, we call a background worker which actually creates the index.
2. **Index Creation by Background Worker**
   - When our planner hook detects a sequential scan that meets our predefined cost-benefit criteria, it immediately triggers the background worker by calling the function `start_auto_index_worker`.
   - The worker then connects to the database, executes the necessary SQL command via PostgreSQL's SPI, and creates the appropriate index.

Frontend/Backend Functionality

- The user can use this extension by loading the *auto_index* folder into the *contrib* folder and then creating the index using `CREATE EXTENSION auto_index`.
- The user can also disable it by setting `auto_index.enable` to false.

Usage of tools available

We used AI to understand the syntax and different types of structures used in `postgres` and how they coordinate with each other in different files and functions. Also, some functionalities like detecting whether or not a table is a system table and figuring out whether a particular column of a particular table has already an index made upon it or not, were done taking help from AI.

Conclusion

The `auto_index` extension enhances PostgreSQL by introducing intelligent, cost-based index creation that operates seamlessly in the background. By analyzing query execution plans and identifying indexing opportunities dynamically, `auto_index` reduces manual tuning overhead and improves query performance over time.

Future Work

- Currently, we use approximate cost and benefit analysis to judge whether or not an index should be created. It can be improved by taking more accurate expressions and also the cost of **index maintenance** can be accounted for.

- The current implementation for column extraction only handles a limited set of expression nodes (e.g., `Var`, `OpExpr`, `BoolExpr`, `RelabelType`). Extending support to other expression types (such as `ScalarArrayOpExpr`, `CaseExpr`, and `FuncExpr`) could capture a broader range of potential candidate columns.
- Our code currently does not handle concurrency in its true sense. We have heuristically avoided the problem caused by concurrency since only a limited type of updates happen to the aidx_queries table. So, in future, this code can be made to handle concurrency effectively. `ON CONFLICT (tablename, colname) DO UPDATE SET num_queries = aidx_queries.num_queries + 1`
- For now, only simple queries were targeted and tested. The code might break on more complex queries which include subqueries. So, that has to be tested and improved.
- For now, we have only handled simple indices not composite indices. Even if two columns are used in a predicate, these are treated independently.

References

- **PostgreSQL Documentation:** https://www.postgresql.org/docs/current/ The official PostgreSQL documentation provides detailed information on PostgreSQL internals, extension development, configuration, and more.
- **Dropping Extension & Cleanup:** This article helped us in succesfully removing the extension and resetting the hook on DROP EXTENSION.
- **Cost Analysis: EXPLAIN command:** https://www.interdb.jp/pg/pgsql03/02.html This article helped us in understanding how costs are estimated using EXPLAIN command in case of Index Scan and Sequential Scan.
- **Database System Concepts:** https://www.db-book.com/online-chapters-dir/32.pdf Used this reference for understanding database concepts and theory especially ch-32 for understanding snapshots.