CS631: Implementation Techniques in DBMS

Automatic Index Creation for PostgreSQL Based on Relation Scans

Submitted BY

- Arnab Bhakta (23m0835)
- Pranav Shinde (23m0833)

Motivation

In PostgreSQL, full relation scans can be inefficient, especially when an index could significantly speed up query execution. However, creating indexes manually can be cumbersome and requires a deep understanding of the database schema and query patterns. Automating the index creation process can optimize query performance without requiring manual intervention, making the database more efficient and user-friendly.

Work Done

- [x] Track Full Relation Scans:
  - [x] Modify the code to monitor the number of full relation scans.
  - [x] Store this tracking information in an appropriate data structure.
- [x] Cost Calculation:
  - [x] Implement functions to calculate the cost of full scans and the cost of index creation.
  - [x] Initially use placeholder values for these costs, with the option to refine them later.
- [x] Threshold Check and Index Creation:
  - [x] Trigger the index creation process when the number of full scans exceeds a threshold (calculated as total cost of scans > cost of index creation).
  - [x] Automatically create indexes.

Design Choices

Existing Flow

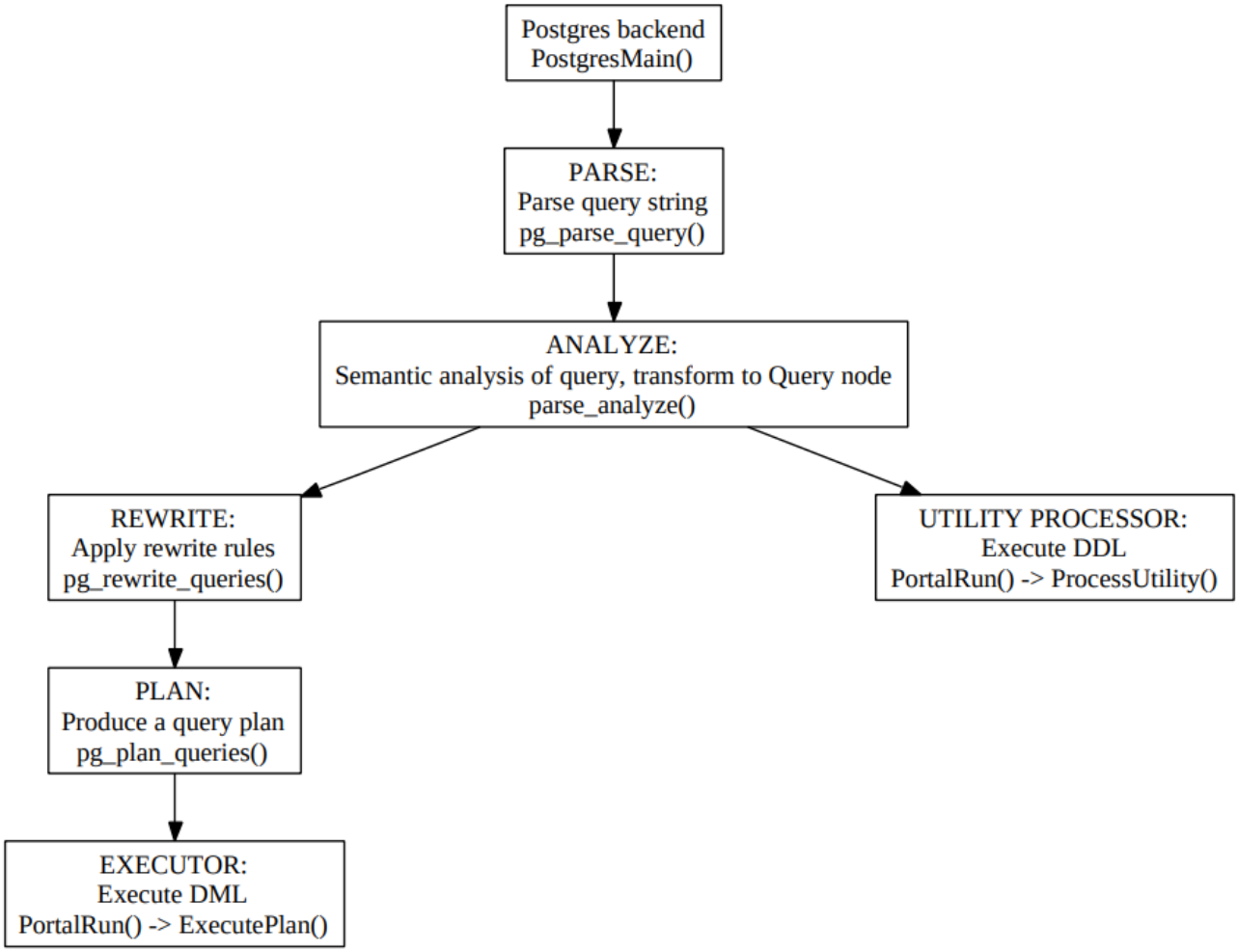

Figure 1: Architecture diagram of PostgreSQL

Here is a brief overview of the stages a query undergoes to produce a result in PostgreSQL:

This section explains the stages a query undergoes in PostgreSQL and how they correspond to specific functions in the PostgreSQL source code.

1. Connection Establishment

- **Function:** `PostgresMain()`
  - Handles the connection lifecycle, including receiving queries from the client and sending results back.
  - Acts as the main entry point for query processing and loops to process each query.

2. Parsing

- **Functions:** `pg_parse_query()` and `parse_analyze()`
  - `pg_parse_query()`:
    - Parses the query string into a raw parse tree structure.

        - Checks the query for correct syntax.
    - `parse_analyze()`:
        - Analyzes the raw parse tree.
        - Resolves object names and creates a fully analyzed query tree.

3. Rewrite System

- **Function:** `pg_rewrite_queries()`
    - Processes the query tree created by the parser.
    - Searches for applicable rules stored in system catalogs and applies transformations to the query tree.
    - Rewrites queries involving views into equivalent queries that operate on base tables.

4. Planning and Optimization

- **Function:** `pg_plan_queries()`
    - Generates a query plan from the rewritten query tree.
    - Evaluates different execution paths (e.g., sequential scan vs. index scan).
    - Estimates the cost of each path and selects the cheapest one.
    - Expands the selected path into a detailed plan structure for execution.

5. Execution

- **Function:** `portal_run()->execute_plan`
    - `portal_run()->execute_plan`:
        - Executes the plan tree generated during the planning stage.
        - Iteratively processes plan nodes to perform tasks like scanning, sorting, joining, and evaluating conditions.
        - Retrieves rows and sends the results back to the client.

The planner/optimizer in PostgreSQL begins by generating plans for scanning each relation (table) involved in a query. The potential plans are influenced by the available indexes on each relation. A sequential scan plan is always created since it is a fundamental method for accessing data. When an index, such as a B-tree index, is defined on a relation and the query includes a condition like `relation.attribute OPR constant`, the optimizer checks if `relation.attribute` matches the key of the B-tree index and if `OPR` corresponds to one of the operators defined in the index's operator class. If both conditions are met, an additional plan is generated that utilizes the B-tree index for scanning the relation. If more indexes exist that align with the query's restrictions, further plans will be considered. Additionally, index scan plans are formulated for indexes that can match the sort order specified in an ORDER BY clause or that may facilitate merge joins.

Change of Flow

```
main(int argc, char ** argv)
PostgresSingleUserMain(int argc, char ** argv, const char * username)
PostgresMain(const char * dbname, const char * username)
exec_simple_query(const char * query_string)
PortalRun(Portal portal, long count, _Bool isTopLevel, _Bool run_once,
DestReceiver * dest, DestReceiver * altdest, QueryCompletion * qc)
PortalRunSelect(Portal portal, _Bool forward, long count, DestReceiver *
dest)
ExecutorRun(QueryDesc * queryDesc, ScanDirection direction, uint64 count,
_Bool execute_once)
standard_ExecutorRun(QueryDesc * queryDesc, ScanDirection direction, uint64
count, _Bool execute_once)
ExecutePlan(EState * estate, PlanState * planstate, _Bool
use_parallel_mode, CmdType operation, _Bool sendTuples, uint64
numberTuples, ScanDirection direction, DestReceiver * dest, _Bool
execute_once)
ExecProcNode(PlanState * node)
ExecProcNodeFirst(PlanState * node)
```

ExecInitSeqScan(SeqScan *node, EState *estate, int eflags):

```
AttrNumber attnum = qual->steps[0].d.var.attnum;
Oid relid = scanstate->ss.ss_currentRelation->rd_id;
printf("attrid: %d relid %d\n", attnum, relid);
int freq = update_seq_attr_file(attnum, relid);
if(should_create_index(relid,attnum, freq)){
    elog(WARNING, "Creating index on attribute: %d rel: %d",attnum, relid);
    create_index(relid, attnum);
}
```

```
ExecSeqScan(PlanState * pstate)
ExecScan(ScanState * node, ExecScanAccessMtd accessMtd, ExecScanRecheckMtd
recheckMtd)
ExecScanFetch(ScanState * node, ExecScanAccessMtd accessMtd,
ExecScanRecheckMtd recheckMtd)
```

Files Changed

Auto Create index on threshold breach

```
src/backend/executor/nodeSeqscan.c | 181
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
1 files changed, 181 insertions(+)
```

Create index using BackGroundWorker

```
src/backend/executor/nodeSeqscan.c             | 35
++++++++++++++++++++++++++----------
src/backend/postmaster/Makefile                |  3 ++-
src/backend/postmaster/bgworker_create_index.c | 60
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
src/include/executor/nodeSeqscan.h             |  2 ++
src/include/postmaster/bgworker_create_index.h | 13 +++++++++++++
5 files changed, 102 insertions(+), 11 deletions(-)
```

Indepth function designs

`int update_seq_attr_file(int attrid, int relid)`

This function updates a file named `seq_attr.txt` with the frequency of access for a specific attribute (`attrid`) in a relation (`relid`). It performs the following steps:

1. Opens the file `seq_attr.txt` for reading and writing. If the file does not exist, it creates a new one.
2. Reads the file contents into an array of `Entry` structures, each containing `attrid`, `relid`, and `counter`.
3. Checks if the combination of `attrid` and `relid` exists in the array. If found, increments the counter for that entry.
4. If the combination is not found, adds a new entry with the given `attrid` and `relid`, and sets the counter to 1.
5. Writes the updated data back to the file.
6. Returns the frequency of access for the given `attrid` and `relid`.

`bool should_create_index(int relid, int attrid, int freq)`

This function determines whether an index should be created for a specific attribute (`attrid`) in a relation (`relid`) based on the frequency of access (`freq`). It performs the following steps:

1. Opens the relation with `relid` and retrieves its statistics.
2. Calculates the cost of a sequential scan (`seq_scan_cost`), the cost of creating an index (`index_creation_cost`), and the cost of an index scan (`index_scan_cost`).
3. Compares the frequency of access (`freq`) with one-third of the index creation cost.
4. Returns `true` if the frequency is greater than one-third of the index creation cost, indicating that an index should be created. Otherwise, returns `false`.

`void create_index(Oid relationOid, int attributeId)`

This function creates an index for a specific attribute (`attributeId`) in a relation (`relationOid`). It performs the following steps:

1. Constructs an index name based on the relation OID and attribute ID.
2. Opens the relation with `relationOid`.
3. Initializes an `IndexInfo` structure with the necessary index information.
4. Sets the index column names, class object ID, column options, and relation options.
5. Calls the `index_create` function to create the index with the specified parameters.
6. Closes the relation.

References

- An Introduction to Hacking PostgreSQL by Neil Conway and Gavin Sherry (https://www.cse.iitb.ac.in/infolab/Data/Courses/CS631/PostgreSQL-Resources/pg_hack_slides.pdf)
- Overview of PostgreSQL Internals (https://www.postgresql.org/docs/current/internals.html)