CS632 Project: Index Selection Tool for PostgreSQL

Mahendra Chavan
(Roll No: 08305043)

under the guidance of

Prof. S. Sudarshan

Index Selection Tool

The tool takes as input a workload of SQL queries along with their frequncies in the workload and suggests a set of suitable indexes for the workload.

An indexable column R.a for a query in the workload is a column such that there is a condition of the form *R.a operator Expression* in the WHERE clause. Columns in GROUP BY, ORDER BY are also considered to be indexable. An admissible index for a query is an index that is on one or more indexable columns on a query.

The tool finds out all the indexable columns from the workload queries. For each query, it finds out an best of the admissible indexes by calling PostgreSQL's optimizer to get the cost improvements with the admissibile indexes.

It does the index selection in four steps:

1. *Query specific index selection:* For each query, it creates a fake indexes on the indexable columns one at a tme and collects the query cost with each of the admissble indexes. A fake index is an index which is created only in the catalogs without actually building it. After the cost collection, it chooses the best indexes greedily. It uses the Greedy(m,k) algorithm to choose the best indexes.
   Greedy(m,k) – The problem of finding out the optimal subset of the given set of admissible indexes is NP hard. Greedy(m,k) algorithm enumerates all the subsets of size less than equal to m of the input set of admissible indexes. It chooses the subset which gives the best cost for the given query. After this, it adds indexes to this selected subset till the added index improves the query cost further. K is the upper bound on the number of indexes cosen in a solution. Greedy(2,k) algorithm has been found to work the best in terms of the execution time and the quality of the selected indexes.

2. *Index selection for the workload:* A workload contains a set of queries each with its frequency in the workload. The input to this step is the query specific selected indexes from step 1. It consolidates all these indexes and does a greedy selection over them using Greedy(m,k) algorithm. This time, for each index, it collects the cost of the entire workload. The workload cost is the weighted sum of the individual query costs where the weight is the frequency of a query in the workload.

3. *Multi-column index selection for the workload:* The input to this step is a set of two-column indexes. The columns whose admissible indexes were selected in the second step are used as the leading columns for the candidate multi-column indexes. The columns whose admissible indexes are not present in the output of the second step form the trailing columns of the candidate multi-column indexes. Once the input set of two-column indexes has been formed in this way, it selects the best two-column indexes for the workload using the Greedy(m,k) algorithm.

4. *Final index selection for the workoad:* The final step consolidates the outputs of the second and the third steps i.e. the best seingle column indexes for the workload and the best two-column indexes for the workload. From this consolidated set, it finds out the final set of the best indexes by Greedy(m,k) algorithm.

Implementation Details:

The tool has been implemented in C for PostgreSQL. A new command *propose_indexes* has been added in PostgreSQL which triggers the index selection process. The workload is specified in a log file called *top10.log* which is placed in the install/data/ directory of postgres. A sample input workload is as below:

4 *select count(distinct p_partkey) from part left outer join partsupp on part.p_partkey=partsupp.ps_partkey where part.p_partkey > 300;*

6 *select supplier.s_name, part.p_name from supplier inner join partsupp inner join part on part.p_partkey= partsupp.ps_partkey on partsupp.ps_suppkey=supplier.s_suppkey where partsupp.ps_suppkey > 600;*

The prefixed number to each query is the frequency of the query in the workload. The tool generates a log file called *index_tuner.log* in the install/data/ directory. The log file lists the outputs of all the four steps and also the greedy decisions taken by the algorithm at each step.

Code changes:

1. *Fake Index Creation*: Added a new function call to DefineIndex() function with the *skip_build* parameter set to **true** in *backend/tcop/utility.c* . This new function call is executed when the flag *is_fake_index* is set to **true**. The index selection module sets this flag before creating a fake index and resets it back to **false** immediately after creating the fake index.
2. *Command to invoke the tool*: Added a check on *query_string* in the function *exec_simple_query* in *backend/tcop/postgres.c* . If the query string is '*propose_indexes*', it calls the function *select_indexes.*
3. *Index Selection Tool:* The functions written in the file *backend/tcop/*select_indexes.*c* implement the entire index selection tool. It cntains separate functions for all the four steps. For finding out the admissible indexes it calls *pg_parse_query in backend/tcop/postgres.c* which returns the full query tree.
4. *Cost Collection*: The function *pg_get_cost* calls the optimizer to get the estimated query cost. The code from the function *exec_simple_query* in *backend/tcop/postgres.c* till the point it calls the planner to get the best plan for the query (which contains the estimated

query cost) has been reused to implement this function.

Experiments:

I used TPC-H dataset for testing the tool. The lineitem table had around 1,20,000 rows and the other tables had a proportional number of rows.

The results for a couple of workloads are as below:

1. **First Workload**
    1. *Workload*

       1 *select c_name, count(*) from customer where customer.c_custkey=89 group by customer.c_name order by customer.c_name;*

       10 *select * from nation order by nation.n_name;*
       6 *select count(*) from lineitem where lineitem.l_orderkey = 100;*

    2. *Proposed Indexes:*

       CREATE INDEX ind_lineitem_l_orderkey ON lineitem USING BTREE(l_orderkey);

       CREATE INDEX ind_customer_c_custkey_c_name ON customer USING BTREE(c_custkey,c_name);

    3. *Workload Cost before creating these indexes: 50 seconds*

    4. *Workload Cost after creating these indexes: 0.3 seconds*

2. **Seond Workload**

    1. *Workload*

       4 *select count(distinct p_partkey) from part left outer join partsupp on part.p_partkey=partsupp.ps_partkey where part.p_partkey > 300;*

       6 *select supplier.s_name, part.p_name from supplier inner join partsupp inner join part on part.p_partkey= partsupp.ps_partkey on partsupp.ps_suppkey=supplier.s_suppkey where partsupp.ps_suppkey > 600;*

    2. *Proposed Indexes:*

       CREATE INDEX ind_part_p_partkey ON part USING BTREE(p_partkey);

       CREATE INDEX ind_partsupp_ps_suppkey_ps_partkey ON partsupp USING BTREE(ps_suppkey,ps_partkey);

3. *Workload Cost before creating these indexes: 5.2 seconds*

4. *Workload Cost after creating these indexes: 2.5 seconds*

For almost all the workloads, it proposed meaningful indexes which after creating gave significant boost in the performance of the workload queries.

Current Issues:

The optimizer sometimes s returns NaN exception as the query cost. The problem occurs when a query scans large tables . The problem persists even with using the Postgres' memory management function palloc.

Possible Enhancements:

1. The tool does not consider subqueries. It can be extended further to consider the admissible indexes from subqueries.
2. The tool assumes that no indexes are already present in the database. It can be improved further to take into account the already existing indexes by excluing those admissible indexes which are already present.
3. Presently, all the proposed indexes are secondary. The proposed indexes set can be refined further by considering primary indexes.
4. Currently, All the proposed indexes are BTREE indexes. The tool can be extended further to analyze and propose RTREE, HASH and GIST type indexes.
5. To improve the accuracy of the cost values further, the tool may decide to build the real indexes instead of fake indexes on smaller relation (provided the index buildng does not take long time) during the index selection process.

Reference:

An efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server, Surajit Chaudhuri, Vivek Narasayya, Proceedings of the 23rd VLDB conference Athens, Greece, 1997