CS 632 Project Report

Semi Automatic Index Tuning

by
D V Deepankar Reddy (09005060)
Hasan Kumar Reddy A (09005065)

Abstract

DBA must choose a set of indices appropriate for the workload. The system can aid him providing him recommendations. The technique presented is a hybrid of online and offline techniques by taking feedback from the DBA.
This requires that the DBA should have the ability to propose hypothetical indexes and quantitatively analyze their impact on performance of workbench.

Salient Features of the project

- Read queries from the workbench and propose indexes on those queries which will improve the performance of the system
- Also take the feedback from the DBA into account while giving further suggestions

Outline

Directory Structure [6]

- postgresql-9.1.4/ - Postgresql 9.1.4 with a patch for Hypothetical Indexes[5]
- external_jars/
    - gsp.jar - General SQL Parser, for extractIndexes(query)
    - postgresql-jdbc-9.1-903.jdbc4.jar - JDBC postgresql driver
- dellstore2-normal-1.0 - Database dump
- PostgreSQL_index_tuning - Eclipse project containing all source files
    - src
        - AdminDB.java - Entry point
        - DBConnection.java - Database connection
        - DBTime.java - Class to store time - measured in #page_fetches
        - Index.java - Index class to store all its attributes
        - IndexExtraction.java - Extract interesting indexes form a query
        - IndexTuner.java - Contains necessary function for pruning candidate indexes. Also acts as slaves and runs the algortihm for each of the partition.
        - Master.java - Controls, Monitors and Serves the slaves with required statistics and aggregates the results in a central fashion.
        - Partitioner.java - Implements our own in house developed approximate partitioning algorithm. It is basically a modified version of kruskals algorithm but keeps in mind the constraints of computing an memory

Main Algorithm of the System:

Main Algorithm for the index tuning here is the WFIT (**W**ork **F**unction **I**ndex **T**uning) Algorithm. It is basically a dynamic programming algorithm which keeps track for the every possible subset of indices being considered the following $W_{q_n}(S)$ where S is the configuration (set of indices) being considered. The variable means that it has executed until the query q_n and the final configuration after the execution is the S. Using these variables we can easily infer that the optimal S is the one which has the least value $W_{q_n}(S)$. Suggestion is $argmin\ W_{q_n}(S)$.

Now for calculating the above variables step by step we use the following recurrence equation mentioned in the paper[1] referred below.

$$w_n(S) = \min_{X \subseteq \mathcal{C}} \{w_{n-1}(X) + cost(q_n, X) + \delta(X, S)\} \quad (4.1)$$
$$w_0(S) = \delta(S_0, S)$$

Now as we see the above equation we have the term $\delta(X, S)$ which is the cost for creating the indices. Now this is type of estimation was not available in postgresql. So we used a new heuristic for this enumeration. What we did is we run the selection query with projections on the attributes of the index and we order by the order specified in the index and then we also we run this same query after creating the hypothetical indices on the Admin Database which essentially contains only the hypothetical indices and then we used the difference between the two as the required approximation for the above enumeration. The complete algorithm can be seen in the reference.

Optimizations to the above algorithm

One thing to notice is that solving of the above recurrence equation needs the running time of Algorithm to be $\Theta(n^2)$. But this makes the Algorithm solve the problem for atmost 10 - 13 indices to find the optimal subset among them. But 10 is very low and can often lead to missing of large number of good index subsets. So to counter this problem they introduced the notion of divide and conquer to solve the problem. So what they did is to divide the whole set of indices in question into k partitions and solve the problem separately for each of the partition. Now this can give the opportunity to solve for large set of indices by choosing appropriate number of partitions.

Now one has to be careful while doing these partition, because randomly creating the partitions may make us lose some valuable subsets for consideration of indices. Now one has to choose those kind of partition where across the partitions the interaction between the indices is very low. Now to tackle this problem they designed this as a graph problem and then gave a metric to the edge between two indices which shows the degree of interaction between them. The Complete theory about the metric considered, justifications and algorithms for effectively computing the metrics for large number of indices is explained in the reference[2] in the end of

this report.

In the paper they have developed a randomized algorithm for choosing the new partitions to the system when some new set of indices get added or when the workbench of queries get upgraded. Now the algorithm is randomized is a little complicated and also heavily based on the paper reference[2]. So instead of implementing their random algorithm we implemented our own algorithm to do the partitioning. It is modified version of Kruskal's (MST) algorithm. But it is enoughly modified to satisfy our needs for smooth execution of the divide and conquer algorithm.

We have added the following changes to the Kruskal's algorithm. We maintained a union find data structure and then have the edge weights which are the degree of interactions between the indices. Now we start merging the partitions which have high degree of interaction between them, in kind of edges between them. but while merging we keep in mind that to prevent hot spotting, if we observe that a partition is getting hot spotted we then ignore any edges between that partition and any other partition. Like this we are able to achieve our goal deterministically by keeping check on number of indices in a partition (approximately 10) and also getting the requisite number of partitions to suit our availability of computing power.

Feedback mechanism

Now in the above paper they have considered to implement the feedback mechanism from the Database Admin. Thus making the Semi automatical in the name justified. What they do that they take two sets of indices F^+ and F^- , which are the good indices and bad indices respectively. Now when the above WFIT generates a suggestion, they take the suggestion and intersect it with F^+ and then they subtract this from F^- and this gives the final set of suggestions. In addition to that they spike the value of the subsets which contains the F^- and this can stop these subsets from appearing again soon in the suggestions.

WFIT Algorithm Overview (Source: [1])

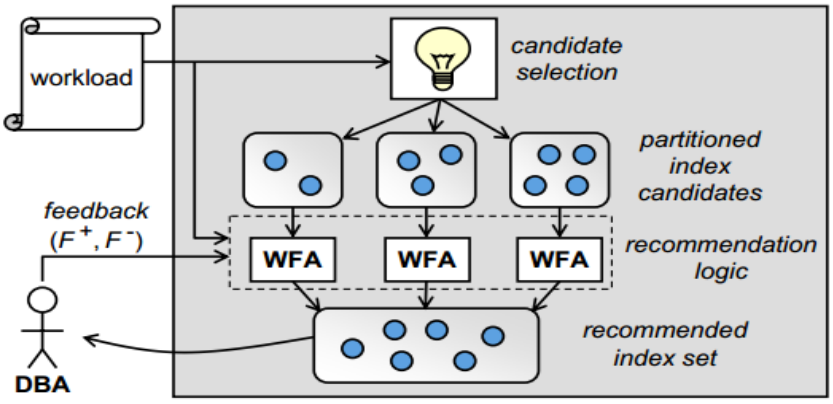

Figure 1: Components of the WFIT Algorithm.

Candidate Index Selection

For every additional new query, we generate a set of interesting indexes. The idea is to determine the best configuration for each query independently and consider all indexes that belong to one or more of these best configurations as candidate index set.

Candidate Enumeration (*optional step*)

We use greedy(m,k) alogorithm to pick k indexes from n candidate indexes, where first we pick an optimal configuration of size m (m<=k) and this seed is expanded greedily until size k.

What-If

Given a query and an index configuration, what-if calculates the cost of executing that query if all the index configurations are materialized. It does it with the help of postgresql patch for hypothetical indexes and explain hypothetical by actually creating hypothetical indexes.

Analyze

Executes the main WFIT algorithm. It updates the stats we kept for each possible subset of indices with next set of stats we get from the existance of new query in the work bench.

Feedback

This executes the feedback step by considering the two subsets F^+ and F^- , it updates the suggestions accordingly and also do a spike up for bad index containing subsets.

Deploy

This Function deploys the algorithm on the several slaves here. The slaves here need not be threads, inherent speed up happens due to the divide and conquer nature of the algorithm in consideration

Master

Master is the guy which aggregates the statistics that are available from different slaves, Also it is the program that calls the partitioner for reshuffling or including the new candidate index views. And then when such reshuffle happens what this program does uses the stats from the previous steps and then calculates the new stats for the current partitions. The procedure is elaborately described in the ***update*** algorithm in the reference[1] ,given below.

Partitioner

This implements the partition algorithm described in the above part of the report. Its just a modified version of the Kruskal's algorithm. But also considers our computing resources and hot spotting conditions.

References:

1. Karl Schnaitter, Neoklis Polyzotis. Semi-Automatic Index Tuning: Keeping DBAs in the Loop http://users.soe.ucsc.edu/~alkis/papers/wfit.pdf
2. Karl Schnaitter, Neoklis Polyzotis, Lise Getoor. Index Interactions in Physical Design Tuning http://users.soe.ucsc.edu/~alkis/papers/PAPER582.pdf
3. Surajit Chaudhuri, Vivek Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server http://www.vldb.org/conf/1997/P146.PDF
4. Surajit Chaudhuri, Vivek Narasayya. AutoAdmin "What-If" Index Analysis Utility http://research.microsoft.com/pubs/68480/autoadmin_conf_version.ps
5. Postgresql with Hypothetical Indexes http://www.inf.puc-rio.br/~postgresql/index.php?acao=projeto1/agente_intrusivo_teoria
6. Project Source https://github.com/mintuhouse/postgresql-tuning