QUIET: Continuous Query-driven Index Tuning[*]

Kai-Uwe Sattler Ingolf Geist Eike Schallehn

Department of Computer Science
University of Magdeburg
P.O. Box 4120, 39016 Magdeburg, Germany
{kus|geist|eike}@iti.cs.uni-magdeburg.de

Abstract

Index tuning as part of database tuning is the task of selecting and creating indexes with the goal of reducing query processing times. However, in dynamic environments with various ad-hoc queries it is difficult to identify potential useful indexes in advance. In this demonstration, we present our tool QUIET addressing this problem. This tool "intercepts" queries and – based on a cost model as well as runtime statistics about profits of index configurations – decides about index creation automatically at runtime. In this way, index tuning is driven by queries without explicit actions of the database users.

1 Introduction

Today's enterprise database applications are often characterized by a large volume of data and high demands with regard to query response time and transaction throughput. Beside investing in new powerful hardware, database tuning plays an important role for fulfilling the requirements. However, database tuning requires a thorough knowledge about system internals, data characteristics, the application and the query workload. Among others index tuning is a main tuning task. Here, the problem is to decide how queries can be supported by creating indexes on certain columns. This requires to choose between the benefit of an index and the loss caused by space consumption and maintenaince costs.

Though index design is not very complicated for small or medium-sized schemas and rather static query workloads, it can be quite difficult in scenarios with explorative analysis and many ad-hoc queries where the required indexes cannot be foreseen. An example of such a scenario is information fusion – the integration and interpretation of data from heterogeneous sources, where integrated data, intermediate results (e.g. cleaned or aggregated data, mining models etc.) are transparently materialized in order to speed up explorative/interactive analysis tasks.

A first step in providing support for such scenarios is already done with the most current releases of the major commercial DBMS such as Oracle9i, IBM DB2 Version 8, and SQL Server 2000. These systems include so-called index wizards which are able to analyze a workload (in terms of costs of previously performed queries) and – based on some heuristics – to derive recommondations for index creation, as described in [CN97, VZZ$^+$00]. This is mostly implemented using "virtual" indexes that are not physically created but only considered during query optimization in a "what if" manner. Though these tools utilize workload information collected at runtime they work still in design mode. That means, the DBA has to decide about index selection and index creation is completely separated from query processing.

In [Gra00] Graefe raised the question if one can exploit table scans in queries for building indexes on the fly which can be utilized by following operations in the same query or even by other queries. This approach would extend the idea of index wizards in two directions:

1. The database system automatically decides about index creation without user interaction.
2. Indexes can be built during query processing, i.e. full table scans are used to create indexes which are exploited for remaining parts of the query plan.

Both ideas are in principal orthogonal, i.e., if changes of the index configuration are performed automatically, the system may do this between queries, or schedule these changes to be performed during times of low system load. Building indexes during query processing would require more profound changes to currently existing systems, especially because index creation is considered to be done once during physical implementation of the database, and therefore up to now was not a major focus for applying optimization

[*]This research was partially supported by the DFG (FOR 345/1).

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

techniques. Nevertheless, a database system implementing these both strategies would support queries that are able to build indexes on demand and can better meet the requirements of dynamic explorative scenarios.

In this demonstration we present a middleware-based approach supporting such query-driven index tuning. This approach comprises

- a cost model taking into account costs for index creation and maintenance as well as benefits for the same and/or potentially future queries and
- decision strategies for choosing indexes which are to be created during query processing in a space-limited environment.

We have implemented this in a middleware called QUIET sitting between the query client(s) and the DBMS and acting as a DBMS proxy. Each query from the client is first sent to the QUIET system. This module analyzes the query and – based on statistics about profits of both existing (i.e., already materialized) and non-existing (i.e., virtual) indexes – decides about creating new indexes before executing the actual query. Thus, eventually chosen indexes are built and the query is forwarded to the DBMS which now can use these new indexes for processing the query. For the purpose of evaluating the benefit of virtual indexes we exploit features of the DBMS optimizer (in our case IBM DB2 Version 8.1), which is able to create query plans with virtual indexes and to derive index recommendations. In this way, we extend the static design-time approach of index wizards (for example available in DB2 as index advisor `db2advis`) towards a dynamic, contiously running tuning facility.

2 Cost-based Index Selection

The main objective of our approach is to improve the response time for a sequence of queries by dynamically creating additional indexes without explicit intervention of a user or DBA. Because creating indexes without limits could exhaust the available database space, we assume an index pool – an index space of limited size acting as persistent index cache. The size of this pool is configured by the DBA as a system parameter. Based on this assumption a query is processed as follows:

1. A given query Q is optimized assuming all potentially useful indexes are available. In addition to the query plan, this step returns a set of recommended indexes.
2. The index recommendation is used to update a global index configuration where cumulative profits of both materialized and virtual indexes are maintained.
3. Next, we have to decide about
   (a) creating indexes from the virtual index set
   (b) replacing other indexes from the index pool if there is not enough space for the newly created index.

For dealing with costs and benefits of indexes as part of automatic index creation we have to distinguish between materialized and virtual (i.e. currently not materialized) indexes. Note, that we do not consider explicitly created indexes such as primary indexes defined by the schema designer. Furthermore, we assume statistics for both kind of indexes (virtual/materialized), possibly computed on demand: if a certain index is considered the first time, statistical information are obtained.

A set of indexes $I_1, \dots I_n$ which are used for processing a query Q is called *index set* and denoted by $\mathcal{I}$. The set of all virtual indexes of $\mathcal{I}$ is $\mathrm{virt}(\mathcal{I})$, the set of all materialized indexes is $\mathrm{mat}(\mathcal{I})$. Let be $\mathrm{cost}(Q)$ the cost for executing query Q using only existing indexes, and $\mathrm{cost}(Q, \mathcal{I})$ the cost of processing Q using in addition indexes from $\mathcal{I}$. Then, the *profit* of $\mathcal{I}$ for processing query Q is

$$\mathrm{profit}(Q, \mathcal{I}) = \mathrm{cost}(Q) - \mathrm{cost}(Q, \mathcal{I})$$

In order to evaluate the benefit of creating certain indexes for other queries or to choose among several possible indexes for materialization we have to maintain information about them. Thus, we collect the set of all materialized and virtual indexes considered so far in the *index catalog* $\mathcal{D} = \{I_1, \dots I_k\}$. Here, for each index I_i the following information is kept:

- $\mathrm{profit}(I_i)$ is the (cumulative) profit of the index,
- $\mathrm{type}(I_i) \in \{0, 1\}$ denotes the type of index, with $\mathrm{type}(I_i) = 1$ if I_i is materialized and 0 otherwise,
- $\mathrm{size}(I_i)$ is the size of the index.

The costs for maintaining indexes (updates, inserts, deletes) are considered in the form of negative profits.

The profit of an index set according to a query can be calculated in different ways. One way is the modification of the optimizer and the collection of additional statistics about “virtual indexes”. The modification concerns about holding of different plans during the optimization algorithm, which helps to compare different index configurations and their corresponding query costs. Here, techniques from the area of the adaptive query processing can be applied. The described functionality is included in some commercial systems, including DB2 and Oracle, though the availability of according interfaces varies.

The subset of $\mathcal{D}$ comprising all materialized indexes is called *index configuration* $\mathcal{C} = \mathrm{mat}(\mathcal{D})$. For such a configuration it holds

$$\sum_{I \in \mathcal{C}} \mathrm{size}(I) \leq \textit{MAX\_SIZE}$$

i.e., the size of the configuration is less or equal the maximum size of the index pool.

By maintaining cumulative profit and cost information about all possible indexes we are able to determine an index configuration optimal for a given (historical) query workload. Assuming this workload is also representative for the near future, the problem of index creation is basically the problem of maximizing the overall profit of an index configuration:

$$\max \sum_{I \in \mathcal{C}} \text{profit}(I)$$

This can be achieved by materializing virtual indexes (i.e. add them to the current configuration) and/or replace existing indexes. In order to avoid thrashing, a replacement is performed only if the difference between the profit of the new configuration $\mathcal{C}_{\text{new}}$ and the profit of the current configuration $\mathcal{C}_{\text{curr}}$ is above a given threshold:

$$\text{profit}(\mathcal{C}_{\text{new}}) - \text{profit}(\mathcal{C}_{\text{curr}}) > \mathit{MIN\_DIFF}$$

Considering the cumulative profit of an index as a criterion for decisions about a globally optimal index configuration raises an issue related to the historic aspects of the gathered statistics. Assuming that future queries are most similar to the most recent workload, because database usage changes in a medium or long term, the statistics have to represent the current workload as exactly as possible. Less recently gathered statistics should have less impact on building indexes for future use. Therefore, we applied an aging strategy for cumulative profit statistics based on an idea presented by O'Neil et al. in [OOW93] and refined by Scheuermann et al. in [SSV96]. If for one index I_{max} the cumulative profit exceeds a certain watermark

$$\text{profit}(I_{max}) \geq \mathit{MAX\_PROFIT}$$

the statistics for all indexes $I_i \in \mathcal{I}$ are reset to

$$\text{profit}(I_i) := s \cdot \text{profit}(I_i), 0 < s < 1$$

In order to globally decide about an index configuration optimal for future queries, statistics about possible profits has to be gathered, condensed and maintained to best represent the current workload of the system, and finally based on these information a decision has to be made if an index configuration can be changed at a certain point in time. During processing a query Q the statistics must be updated by adding profits of an index set $\mathcal{I}$ returned by the virtual optimization to the single indexes $I_i \in \mathcal{I}$. Because index profits are not independent, but on the other hand can not be quantified per index by currently existing optimizer, we used the approximation to add the average profit $\frac{\text{profit}(Q,\mathcal{I})}{|\mathcal{I}|}$ to the cumulative profit.

If a locally optimal index set $\mathcal{I}$ can replace a subset $\mathcal{I}_{rep} \subseteq \text{mat}(\mathcal{D})$ of the currently materialized index configuration, such that

$$\begin{gathered}\text{profit}(\text{mat}(\mathcal{D}) \cup \mathcal{I} \setminus \mathcal{I}_{rep}) - \\ \text{profit}(\text{mat}(\mathcal{D})) > \mathit{MIN\_DIFF} \wedge \\ \text{size}(\text{mat}(\mathcal{D}) \cup \mathcal{I} \setminus \mathcal{I}_{rep}) < \mathit{MAX\_SIZE}\end{gathered}$$

an index configuration change can be triggered. These conditions allow only improvements of the index configurations according to the current workload and conforming to our requirements regarding index space, and the criterion to avoid thrashing. The replacement index set $\mathcal{I}_{rep}$ can be computed from the currently materialized index set $\text{mat}(\mathcal{D})$ applying a greedy approach. To do this, we sort $\text{mat}(\mathcal{D})$ ascending to cumulative profit and choose the least beneficial indexes, until our space requirements are fulfilled. Now, if the found replacement candidate is significantly less benefitial than the index set we investigate for a possible materialization, the index configuration can either be changed before, during, or after query execution or scheduled to be changed later on.

3 Tool Demonstration

To illustrate the ideas presented in the previous sections we implemented the QUIET system and tools to simulate and visualize workloads and their impact on the index configuration. The focus of interest for the visualization is on

- how changing workloads influence the statistics for index candidates,
- how the system adjusts to these changes by changing the currently materialized index configuration, and
- how the parameters of the self-tuning influence the overall process.

As such, the demonstration tools are also used as a basis for an ongoing thorough evaluation of the introduced concepts.

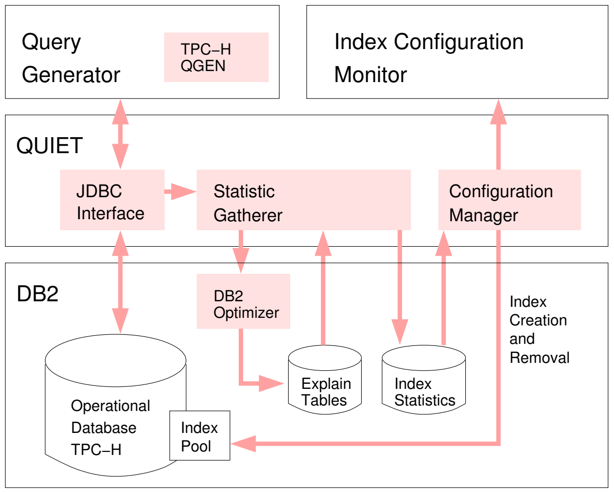
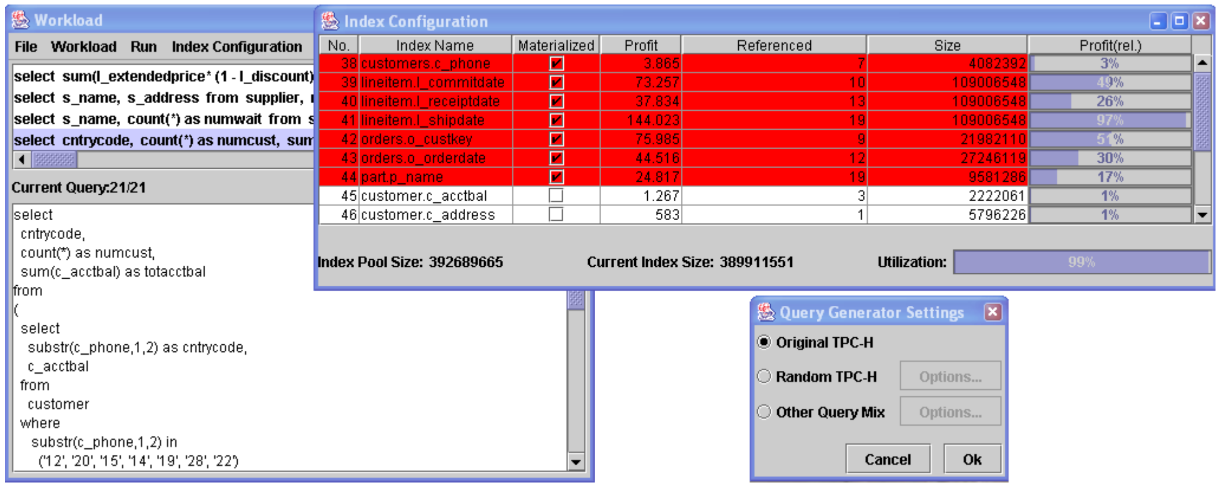


Figure 1: Architecture of the QUIET Demo

The QUIET Demonstration architecture shown in Figure 1 conists of the described functionality implemented as middleware on top of the DB2 system, a query generator, and a monitor for index configuration statistics and changes. Offering a *JDBC interface*, the QUIET system on the one hand just passes through queries to and results from the database. On the other hand queries are passed on to the *Statistic Gatherer*, which uses virtual optimization and the index advisor facility of DB2 to compute benefits of possible index usage for a single query. Furthermore, it



```
select sum(l_extendedprice* (1 - l_discount)
select s_name, s_address from supplier,
select s_name, count(*) as numwait from s
select cntrycode, count(*) as numcust, sum
```

Current Query:21/21

```
select
 cntrycode,
 count(*) as numcust,
 sum(c_acctbal) as totacctbal
from
(
 select
  substr(c_phone,1,2) as cntrycode,
  c_acctbal
 from
  customer
 where
  substr(c_phone,1,2) in
   ('12', '20', '15', '14', '19', '28', '22')
```

No.	Index Name	Materialized	Profit	Referenced	Size	Profit(rel.)
38	customers.c_phone	✓	3.865	7	4082392	3%
39	lineitem.l_commitdate	✓	73.257	10	109006548	49%
40	lineitem.l_receiptdate	✓	37.834	13	109006548	26%
41	lineitem.l_shipdate	✓	144.023	19	109006548	97%
42	orders.o_custkey	✓	75.985	9	21982110	51%
43	orders.o_orderdate	✓	44.516	12	27246119	30%
44	part.p_name	✓	24.817	19	9581286	17%
45	customer.c_acctbal	☐	1.267	3	2222061	1%
46	customer.c_address	☐	583	1	5796226	1%

Index Pool Size: 392689665 Current Index Size: 389911551 Utilization: 99%

[✓ Original TPC-H] Random TPC-H Other Query Mix

Figure 2: Screenshot of the QUIET Demo

updates the cumulative *Index Statistics* and applies aging if required. The index statistics are evaluated by the *Configuration Manager*, which checks if a promising index set can replace a subset of the currently materialized indexes in the index pool as described in the previous section. If this is the case, the current index configuration is changed automatically. The index statistics and the work of the configuration manager are visualized by the *Index Configuration Monitor*. Finally, the whole demonstration uses data and queries from the TPC-H benchmark. Our *Query Generator* incorporates the TPC-H query generator, but provides additional queries and ways to influence the current query mix.

A screenshot of the demo is given in Figure 2. The currently generated workload can be controlled and monitored using the query generator part of the tool. For this purpose the user can specify and change the query mix generated by the TPC-H query generator. As the default queries provided by the benchmark are focused on analytical processing, we provide an additional set of query templates that presents a typical business workload, and these queries in general involve less relations, less grouping, and have smaller result sets. For the current query indexes proposed by the DB2 index advisor facility and the related benefit are monitored. The query generator also controls the execution and possible log creation. The generator can run indefinitly until it is stopped for demonstration purposes, in which case the user can interactively change parameters and the query mix. In another execution mode, the generator runs for a fixed length of time or number of queries for gathering test results.

The current index configuration and the statistics for index candidates are monitored in a seperate application window. The statistics hold information on the usage frequency, cumulative profit, and the index size. One column is a graphical representation of the cumulative profit relative to the global watermark *MAX_PROFIT*, so by sorting the statistics according to the profit, the aging mechanism can be observed. The currently materialized index configuration can be monitored by sorting the table on the *Materialized* column. As the decision about the inclusion of an index in the materialized configuration is a a trade-off between its cumulative profit and its space requirement, toggling between sorting by profit and sorting by size illustrates the index replacement strategy. The overall size of the currently used index configuration and its required space in the index pool are monitored below the table representation of the index statistics.

References

[CN97] S. Chaudhuri and V.R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. of Int. Conf. on VLDB 1997*, pages 146–155, 1997.

[Gra00] G. Graefe. Dynamic Query Evaluation Plans: Some Course Corrections? *Bulletin of the Technical Committee on Data Engineering*, 23(2), June 2000.

[OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. of ACM SIGMOD 1993*, pages 297–306, 1993.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *Proc. of Int. Conf. on VLDB 1996*, pages 51–62, 1996.

[VZZ$^+$00] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. of ICDE 2000*, pages 101–110, 2000.