Autonomous Query-driven Index Tuning

Kai-Uwe Sattler
Department of Computer Science and Automation
TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany
k.sattler@computer.org

Eike Schallehn Ingolf Geist
Department of Computer Science
University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg, Germany
{eike,geist}@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 potentially useful indexes in advance. In this paper, we investigate an approach addressing this problem by deciding about index creation automatically at runtime in order to speed up processing of subsequent queries. We present a cost model taking into account the benefits of indexes for an evolving query workload and discuss strategies for choosing indexes to be created in a space-limited environment.

1. Introduction

Today's enterprise database applications are often characterized by a large volume of data and high demand 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, applications and the query workload. Among others index selection 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 balance between the benefit of an index and the loss caused by space consumption and maintenance 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. A particular example scenario is OLAP where business intelligence and/or ROLAP tools produce queries as a result of an information request initiated by a user. These tools produce sequences of statements including creating and building tables, inserts and queries [12].

The most current releases of the major commercial DBMS such as Oracle9i, IBM DB2 Version 8, and SQL Server 2000 provide already (limited) support for this scenario. They 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 recommendations for index creation. This is 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 creation and index creation is completely separated from query processing. The drawback of this approach with respect to the above metioned scenarios is that queries are dynamically generated – eventually on temporary tables – and therefore an offline workload analysis is rather difficult. Another drawback of index advisors results from the approach of analyzing a workload once over a fixed period of time and create a static index configuration based on this observation. Despite this approach, in many applications database usage changes over time, e.g. over longer periods due to adjustments to changing work processes, with new applications working on the same data set, or frequent changes like seasonal usage or towards the end of business quarters. Actually, the existence of index advisors most of all raises the question: Why is this not done autonomously and continually by the DBMS itself?

Graefe discussed in [6] the idea of exploiting table scans in queries for building indexes on the fly which can be utilized by subsequent operations in the same query or even by other queries. This approach would extend the idea of index advisors in two directions: First, the database system automatically decides about index creation without user interaction. Second, 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 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 paper we present first results of our work towards such index-building query support. We present a cost model taking into account costs for index creation and maintenance as well as benefits for the same and/or potentially future queries. Furthermore, based on this model, we describe decision strategies for choosing indexes which are to be created during query execution in a space-limited environment. We will show how this approach can be implemented on top of a commercial DBMS by simulating the "on the fly" index building using the CREATE INDEX command right before executing the query. We also discuss extensions of the table scan operator for index building which we have implemented in PostgreSQL. However, we do not deal with mid-query reoptimizations for exploiting indexes built during previous query steps. Moreover, we consider only B+-tree indexes. We are aware these all are important issues and plan to address them in the future. For now, we focus on the problem of selecting and building indexes that could be created while a certain query is executed and could be beneficial for subsequent queries.

2. Processing Index-building Queries

Before discussing the cost model and the selection strategies in detail we will sketch the overall process of executing index-building queries. The main objective of this approach is to improve the execution times of (possible future) queries by creating useful indexes automatically. As creating indexes without limits could exhaust the available database space, we assume an index pool – an index space of limited size acting as a 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) For a given query Q the potentially useful indexes are determined by simple heuristics: if a column appears in a SELECT, WHERE, GROUP BY or ORDER BY clause and there exists no corresponding index for this column or combined columns so far, then the columns are marked as candidate indexes.

(2) Query Q is optimized in a conventional way, i.e., a cost-optimized query plan is derived.

(3) We obtain candidate indexes from step (1) for relations on which the table scan(s) is/are performed. These indexes are called virtual indexes and we create index sets for all possible combinations of them. This is necessary because in most cases we do not need indexes on all columns of a table. The query Q is reoptimized using the virtual indexes, i.e. we consider these indexes as existing indexes with all necessary statistic information (index size etc.).

(4) Next, the cost difference of the plan from step (2) and from step (3) is computed. This value represents the profit of a virtual index set. The index set(s) with the highest profit is/are called index recommendation and is/are used to update a global index configuration where cumulative profits of all indexes (both materialized and virtual) are maintained. Based on the information from this index configuration we have finally 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 indexes.

The above discussion about processing queries leaves out several important issues. First, it should be noted that step (2) and (3) can be merged. An optimizer based on the usual dynamic programming approach can consider relation access via virtual indexes in the first iteration. The only required modifications to the optimization algorithm are

- to generate access plans with virtual indexes if a table scan operator was chosen and
- to not prune a plan if only plans with virtual indexes are better.

Thus, the result of the optimization step consists of at least two plans: a plan without virtual indexes and one or more plans using virtual indexes.

A second issue is the "self-interest" of a query. If we consider only the best plan generated in step (3) we are able to find only an index set contributing to the current query Q because we try to maximize the benefit of this query (*local optimization*). If we would consider all index sets from step (2) that provide a positive profit or at least no high loss, we could create indexes that are possibly useful in the future (i.e. for other queries), too. However, this *global optimization* requires to consider more index sets.

Due to the overhead of considering virtual indexes one could argue not to apply this approach to each query. Instead one could restrict this to certain queries only, e.g. queries with high costs and requiring table scans.

Finally, under the assumption of a space-limited index pool it can be necessary to replace existing indexes in the pool by other indexes if the new virtual indexes promise a higher benefit than the old one. For this purpose, different strategies are possible. Beside classical replacement strategies which have been developed over the past years (e.g. LRU, LFU), the profit of an index can be taken into account. However, this requires to maintain statistics about global profits, e.g. by monitoring and cumulating local profits of an index for different queries.

The query processing described above is to some limited extent supported by current database management systems. Virtual indexes, existing for instance in Oracle, allow the "as if"-runs of the optimizer as in step (3) to check the usefulness of indexes without materializing them. Relying on such a virtual optimization, we still have to find possibly applicable index sets. The DB2 optimizer goes one step further by providing index recommendations covering most of steps (1) to (3). Static design tools such as index wizards or advisors are built on top of the described functionality, which is also used in our approach. By building on the index recommendation facility and the virtual optimization of the DB2 system, our approach extends the existing approaches by providing self-tuning index maintenance at run time.

3. Cost Model

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 that statistics for both kind of indexes (virtual/materialized) are computed on demand: When a certain index is considered for the first time, statistical information about it is obtained.

A set of indexes $i_1, \ldots, i_n$ 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 simply

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

Obviously, if $\mathrm{virt}(\mathcal{I}) = \emptyset$ then $\mathrm{profit}(Q, \mathcal{I}) = 0$.

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* $\mathcal{D} = \{i_1, \ldots, i_k\}$. Here, for each index i the following information is kept:

- i.*benefit* is the benefit, i.e. the cumulative profit, of the index,
- i.*type* $\in \{0, 1\}$ denotes the type of index, with i.*type* $= 1$, if i is materialized and 0 otherwise,
- i.*size* is the size of the index, which is estimated based on the typical parameters available as databases statistics, e.g. the attribute size and the number of tuples in the relation.

The costs for maintaining indexes (updates, inserts, deletes) are considered in the form of negative profits. Because both building and maintenance costs are difficult to estimate without a deep knowledge of the DBMS' cost model, we derived some rules of thumbs experimentally (see Section 5).

The profit of an index set according to a query can be calculated in different ways (see Section 2). However, as we used the DB2 system for our evaluation, we could use the optimizer and recommended virtual indexes. For evaluation purposes we used the following technique to extract cost estimations of queries for different index configurations:

1. compute the costs for the query without any indexes except for primary key indexes via the EXPLAIN mode,
2. derive a recommended index set via the RECOMMEND INDEXES mode,
3. compute the cost for the recommended index set.

This way, we cannot only derive the potential profit of an index set, but the advisor mode of the DB2 optimizer also provides statistical information such as the cardinality and the number of leaf nodes that allow a precise estimation of the index size required for our strategies. Note that we use the cost model of the underlying DBMS. Thus, we can guarantee that our profit estimations are as accurate as the estimated query costs.

If we deal with multi-column indexes we have to consider cases where the optimizer proposes a (virtual) index where indexed attributes are a subset or a superset (more exactly a sublist/superlist) of the attributes of an already existing index. In such cases (and assuming the same order of attributes) the existing index could be exploited instead of creating a new index. However, only if the existing index is a superset we may add the full profit. As an example consider an existing index `R.+A-B+C` on relation `R` for the attributes `A` (ascending order – denoted by "+"), `B` (descending – denoted by "`-`"), and `C`. If the index `R.+A-B` is recommended we could use `R.+A-B+C` instead and therefore may add the profit. For the subset case, the existing index cannot be exploited fully because multi-column indexes are often also used for projections, i.e. avoiding additional page fetches for the base relations. This situation would occur, if we assume an existing index `R.+A` and a recommended index `R.+A-B`. Therefore, we may add only portions of the profit depending on the conformance of both indexes and the selectivity of their common attributes. In our approach we used simple heuristics for this problem.

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

$$\mathrm{size}(\mathcal{C}) = \sum_{i \in \mathcal{C}} i.\mathit{size} \leq \mathit{MAX\_SIZE}$$

holds, 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 finding an index configuration $\mathcal{C}_{\text{new}}$ which maximizes the overall benefit:

$$\max \sum_{i \in \mathcal{C}_{\text{new}}} i.\mathit{benefit}$$

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 benefit of the new configuration $\mathcal{C}_{\text{new}}$ and the benefit of the current configuration $\mathcal{C}_{\text{curr}}$ is above a given threshold. Here, the benefit of a configuration is computed by $\mathrm{benefit}(\mathcal{C}) = \sum_{i \in \mathcal{C}} i.\mathit{benefit}$. In addition, we have to take into account the cost building the new indexes $\mathrm{cost}_{\text{build}}(i)$ which appear as negative profit:

$$\mathrm{benefit}(\mathcal{C}_{\text{new}}) - \mathrm{benefit}(\mathcal{C}_{\text{curr}}) - \sum_{i \in \mathrm{virt}(\mathcal{C}_{\text{new}})} \mathrm{cost}_{\text{build}}(i) > \mathit{MIN\_DIFF}$$

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 [14].

4. Index Selection

The basic idea of our approach is to optimize the indexes of a database system at run time according to the current workload. As described in the previous sections, it is easy to decide whether a query can locally benefit from a certain index configuration by quantifying the profit of feasible index combinations using virtual optimization. In order to globally decide about an optimal index configuration for future queries, the information 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.

As local profits are computed for index sets as described in Section 2, a natural but for reasons described below rather theoretical approach would be to maintain the cumulative profit for all possible index configurations. During the statistics update for the locally optimal index set $\mathcal{I}_i$ of query Q the local profit $\mathrm{profit}(Q, \mathcal{I}_i)$ would be added to each configuration $\mathcal{C}_j$ where $\mathcal{I}_i \subset \mathcal{C}_j$ and aging applied if required. Though we obviously only have to maintain statistics for configurations that have $\mathrm{size}(\mathcal{C}_j) < \mathit{MAX\_SIZE}$ and $\forall \mathcal{C}_i, \mathcal{C}_j : \mathcal{C}_i \not\subseteq \mathcal{C}_j$ we would still face the problem of combinatorial explosion. As an illustration, assume that the index size of all indexes is fixed, such that we only have to consider a fixed number k of indexes on n indexable attributes in each configuration. These possible configurations represent mathematical combinations, so their number would be $\binom{n}{k}$. As an example, in the TPC-H benchmark we used for evaluation purposes, according to the heuristics data type, relation size, and data distribution there were $n = 17$ indexable attributes, and given the reasonable space assumption of $k = 10$ we would have to maintain statistics on 19448 configurations, without even considering multi-column indexes. This would cause an impractical overhead, especially considering that n would tend to be much greater in most realistic scenarios.

To avoid the problems related to the combinatorial explosion of possible index configurations we instead implemented strategies based on per-index statistics. These include approximations for assigning profits to single indexes instead of index configurations and a greedy strategy for replacing indexes from the set of materialized indexes.

While processing a query Q the statistics must be updated by adding profits to each involved index. At this point we considered various strategies for assigning profits to each index involved. One alternative relates to the fact, that there may be various combinations of indexable attributes yielding a profit during virtual optimization. In this case, we can either

- add profits for **all** minimal index sets $\mathcal{I}_i$ yielding a profit, or
- add only profits for the minimal index set that is locally **optimal**, i.e. yields the most profit,

where an index set $\mathcal{I}_i$ is minimal, if there is no index set $\mathcal{I}_j \subset \mathcal{I}_i$ yielding the same profit. While the former yields a more complete picture of possible gains of certain index configurations, the latter introduces less overhead while over large workloads still providing a reasonable approxi-

Algorithm 1 Find locally beneficial index sets

```
Input: Query $Q$
Output:
Locally optimal index set $\mathcal{I}_{\text{opt}}$

procedure findBestIndexSet($Q$)
$\mathcal{I}_{opt}$ := {}
profit := 0
$\mathcal{I}_{\text{Attr}}$ := getIndexableAttributes($Q$)
forall $\mathcal{I} \in$ buildCombinations($\mathcal{I}_{\text{Attr}}$) do
    if $\text{profit}(Q, \mathcal{I}) >$ profit then
        $\mathcal{I}_{\text{opt}}$ := $\mathcal{I}$
        profit := $\text{profit}(Q, \mathcal{I})$
    end if
done
return $\mathcal{I}_{\text{opt}}$
```

Algorithm 2 Find all beneficial index sets

```
Input: Query $Q$
Output:
Set of all minimal beneficial index sets $\mathcal{M}$

procedure findAllIndexSets($Q$)
$\mathcal{M}$ := {}
$\mathcal{I}_{\text{Attr}}$ := getIndexableAttributes($Q$)
forall $\mathcal{I} \in$ buildCombinations($\mathcal{I}_{\text{Attr}}$) do
    if $\mathcal{I} \in \mathcal{M} \vee \text{profit}(Q, \mathcal{I}) = 0$ then continue
    if $\exists \mathcal{I}' \in \mathcal{M}$ :
        $\mathcal{I} \subset \mathcal{I}' \wedge \text{profit}(Q, \mathcal{I}) \leq \text{profit}(Q, \mathcal{I}')$ then
            $\mathcal{M} := \mathcal{M} \cup \mathcal{I} \setminus \mathcal{I}'$
        end if
done
return $\mathcal{M}$
```

mation of configuration benefits. To compute either the locally optimal or all beneficial index sets, Algorithms 1 and 2 respectively avoiding checks of unbeneficial configurations can be used.

The function *getIndexableAttributes(Q)* of Algorithms 1 and 2 returns for query Q the possible indexes. A possible implementation of this function is the index enumeration algorithm presented in [18]. Actually, we used the DB2 index advisor for finding locally beneficial index sets as described in Section 5.

Another question is, how to assign the profit of an index set $\mathcal{I}$ returned by the virtual optimization to the single indexes $i \in \mathcal{I}$ in order to collect the benefit. We considered the following alternatives: for each index

- add a **constant** value (equivalent to reference counting) (**CONST**),
- add the **full** profit $\text{profit}(Q, \mathcal{I})$ of the index set (**FULL**),
- add the **average** profit $\frac{\text{profit}(Q,\mathcal{I})}{|\mathcal{I}|}$ (**AVG**), or
- add the **weighted** profit according to the profit of the single index i $\text{profit}(Q, \mathcal{I}) \frac{\text{profit}(Q,\{i\})}{\sum_{j \in \mathcal{I}} \text{profit}(Q,\{j\})}$ (**WEIGHT**).

All of them are approximations with an increasing degree of accuracy, but even for the last alternative we have to assume that

$$\text{profit}(Q, \mathcal{I}) \neq \sum_{j \in \mathcal{I}} \text{profit}(Q, \{j\})$$

As an example consider a merge join, where the benefit of two indexes on the join attributes in two relations can be greater than the sum of the profits having either one of the indexes alone.

So far we have focused on the analysis of a given query and how to maintain statistics of data gathered from virtual optimization. Now the question arises: is it necessary to update the materialized index configuration? If an index set $\mathcal{I}$ can replace a subset $\mathcal{I}_{\text{repl}} \subseteq \mathcal{C} = \text{mat}(\mathcal{D})$ of the currently materialized index configuration, such that

$$\begin{gathered}\text{benefit}(\mathcal{C} \cup \mathcal{I} \setminus \mathcal{I}_{\text{repl}}) - \text{benefit}(\mathcal{C}) - \\ \sum_{i \in \text{virt}(\mathcal{I})} \text{cost}_{\text{build}}(i) > \textit{MIN\_DIFF} \wedge \\ \text{size}(\mathcal{C} \cup \mathcal{I} \setminus \mathcal{I}_{\text{repl}}) < \textit{MAX\_SIZE}\end{gathered}$$

holds, indexes in $\mathcal{I}_{\text{repl}}$ can be dropped and those in $\mathcal{I}$ can be created. 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. For choosing $\mathcal{I}$ from locally beneficial index sets we considered two strategies:

- from the beneficial index sets choose only the **locally optimal** index set (**LOC**), or
- check all beneficial index sets for a possibly **globally optimal** configuration (**GLOB**).

These strategies are illustrated in Algorithm 3 and 4.

The locally optimal strategy is in some sense egoistic and more beneficial for the current query, especially considering a scenario where indexes are build on the fly during query processing as described before. The globally optimal strategy is more altruistic and could be more flexible regarding global requirements but requires a much higher effort. Using the function from Algorithm 1 and 2 we first derive the beneficial index sets. Next, the estimated costs are used to update the profits of all indexes of $\mathcal{D}$ according to one of the above described alternatives (function *updateProfits*).

The replacement index set $\mathcal{I}_{\text{repl}}$ is computed using the function *findReplacement* from the currently materialized

Algorithm 3 LOC strategy

```
Input: Query $Q$

procedure localOptStrategy($Q$)
$\mathcal{I}_{\text{opt}}$ := findBestIndexSet($Q$)
updateProfits($\mathcal{D}$, $\mathcal{I}_{\text{opt}}$)
$\mathcal{I}_{\text{repl}}$ := findReplacement($\mathcal{C}$, size($\mathcal{I}_{\text{opt}}$))
if $\text{benefit}(\mathcal{C} \cup \mathcal{I}_{\text{opt}} \setminus \mathcal{I}_{\text{repl}}) - \text{benefit}(\mathcal{C}) -$
    $\sum_{i \in \text{virt}(\mathcal{I}_{\text{opt}})} \text{cost}_{\text{build}}(i) > MIN\_DIFF$ then
    /* update configuration */
    forall $i \in \mathcal{I}_{\text{repl}}$ do $i.type$ := 0 done
    forall $i \in \mathcal{I}_{\text{opt}}$ do $i.type$ := 1 done
end if
perform or schedule updates on $\mathcal{D}$
```

Algorithm 4 GLOB strategy

```
Input: Query $Q$

procedure globalOptStrategy($Q$)
$\mathcal{I}_{\text{opt}} := \{\}$; $\mathcal{I}_{\text{repl}} := \{\}$
$maxProfit$ := $\text{benefit}(\mathcal{C}) + MIN\_DIFF$
$\mathcal{M}$ := findAllIndexSets($Q$)
forall $\mathcal{I} \in \mathcal{M}$ do updateProfits($\mathcal{D}$, $\mathcal{I}$) done
forall $\mathcal{I} \in \mathcal{M}$ do
    $\mathcal{I}_{\text{r}}$ := findReplacement($\mathcal{C}$, size($\mathcal{I}$))
    $newProfit$ := $\text{benefit}(\mathcal{C} \cup \mathcal{I} \setminus \mathcal{I}_{\text{repl}}) -$
        $\sum_{i \in \text{virt}(\mathcal{I})} \text{cost}_{\text{build}}(i)$
    if $newProfit > maxProfit$ then
        $\mathcal{I}_{\text{opt}} := \mathcal{I}$; $\mathcal{I}_{\text{repl}} := \mathcal{I}_r$
        $maxProfit$ := $newProfit$
    end if
done
/* update configuration */
forall $i \in \mathcal{I}_{\text{repl}}$ do $i.type$ := 0 done
forall $i \in \mathcal{I}_{\text{opt}}$ do $i.type$ := 1 done
perform or schedule updates on $\mathcal{D}$
```

index set $\mathcal{C} = \text{mat}(\mathcal{D})$ applying a greedy approach. To do this, we sort $\mathcal{C}$ ascending to a replacement criterion and choose the least beneficial indexes, until our space requirements are fulfilled. As replacement criteria we considered

- the **number of references** for an index (**REF**),
- the **cumulative profit** of an index (**PROF**), and
- the **ratio of profit per query** or reference of an index (**PQR**).

Now, if the found replacement candidate is significantly less beneficial than the index set we investigate for a possible materialization, the index configuration can either be changed during query execution as described in Section 2 or scheduled to be changed later on.

Based on these different alternatives for choosing indexes, assigning profits and replacing indexes we can define the following strategies as illustrated in Fig. 1.

	REF	PROF	PQR
LOC	**CONST**	**FULL**	**FULL**
		AVG	**AVG**
		WEIGHT	**WEIGHT**
GLOB	**CONST**	**FULL**	**FULL**
		AVG	**AVG**
		WEIGHT	**WEIGHT**

Figure 1. Classification of possible strategies

Here, **LOC-REF** combines the local index choosing approach with the reference counting strategy and uses only the **CONST** approach for updating profits, whereas **GLOB-REF** is based on the global index choosing strategy. In contrast, the other combinations **PROF** and **PQR** can be used together with detailed profit measures, i.e. **FULL**, **AVG**, and **WEIGHT**.

However, in our implementation and experiments described in the following sections we consider only **LOC**-based strategies. First of all, the number of indexes which have to be taken into account during processing is much smaller and therefore this approach has a significant lower overhead. Secondly, this strategy can be easily implemented on top of index recommendation facilities provided by commercial DBMS such as the index advisor of DB2, which recommends only the best index set for a given query or workload respectively.

After choosing a single index or a set of indexes for materialization these indexes have to be created in order to speed up the processing of the subsequent queries. Here, several approaches are possible:

(1) In the simplest case, the beneficial indexes are collected and the DBA is informed later on about these. In fact, this is the same approach as supported by the index advisors/wizards currently available for commercial DBMS.

(2) The selected indexes are created offline, i.e. during maintenance times of the database system. This avoids delaying the query answer by building the indexes before. But queries cannot be sped up until the indexes are created.

(3) Indexes are built by invoking the CREATE INDEX command explicitly right before the query is processed. We have chosen this approach for evaluating our strategies using a commercial DBMS. However, the draw-

back is that the relation has to be scanned at least twice: the first time for building the index (by executing the CREATE INDEX command) and the second time as part of the query.

(4) One could exploit full table scans of the query in order to build previously chosen beneficial indexes on these tables. The query itself cannot benefit from the index but subsequent queries can be speed up. Furthermore, the additional scan of the CREATE INDEX command can be avoided.

(5) Finally, an index created as result of a table scan could be used directly in this query. This is possible for example in nested iteration joins where in the first iteration the index is built which can be used in the following iterations. This requires to insert a "choose-plan" node into the query plan [4]. Basically, this leads to a reoptimization approach.

The approaches (4) and (5) can be implemented by *deferred* indexes which work as follows: Either the user/DBA or the system indexes creates these indexes by invoking the command

CREATE DEFERRED INDEX *idx_name*
  ON *rel* (*columns*)

Such an index is registered in the data directory like an ordinary index but marked with a special flag. In addition, only an empty index is created on disk. If now a query is executed containing a full table scan on relation *rel*, all deferred indexes for this relation are built during the scan.

Deferred indexes can be used in our approach by invoking CREATE DEFERRED INDEX for all indexes chosen as result of our decision strategy instead of ordinary indexes. In this way, we can avoid the additional effort of explicit index building. However, in cases where the subsequent query profits from this index, it is better to explicitly create the index right beforehand.

5. Implementation and Evaluation

To evaluate and illustrate the ideas presented in the previous sections we developed the QUIET tool using DB2 and Java. The QUIET system consists of a middleware on top of the DB2 database system implementing the proposed functionality, a query generator based on the TPC-H benchmark as well as a monitor for index configuration tracking. The overall system is described in detail in [16].

Using the QUIET system we evaluated our different strategies and parameters and compared the results to the DB2 index advisor as well as to a static strategy without indexes. We used for evaluation IBM's DB2 Universal Database Server V8.1 running on a Sun workstation (SunBlade 2000 UltraSPARC III+, 900 MHz) with 1GB main memory. The experiments ran against TPC-H databases of scale factor 0.1 (100MB) and 1 (1GB). As we conducted many test runs, most of them were executed against the smaller database, but several key runs were also evaluated on the large database. We observed similar results for both database sizes, but because we did a more thoroughly testing for the smaller one, we present here mainly results for scale factor 0.1.

We used two workloads based on the TPC-H benchmark. The first workload **W1** consisted of the original 22 TPC-H queries, which ran four times. Workload **W2** comprised 100 randomly selected TPC-H queries in a random sequence. In order to be able to deal with index building costs, we derived a simple approximation function between execution time for CREATE INDEX commands, index size and DB2's cost measure called timerons and added an experimentally chosen build cost factor for weighting the build costs. Using this function, we could estimate the build costs (in timerons) of an index based on its estimated size with an acceptable error. Index maintenance costs were not considered in our experiments because we focused on OLAP-like queries. However, these costs could be taken into account simply by estimating the costs of the update operation and treat them as negative profit.

First, we wanted to investigate, how good the different strategies perform. The following strategies were selected for comparison:

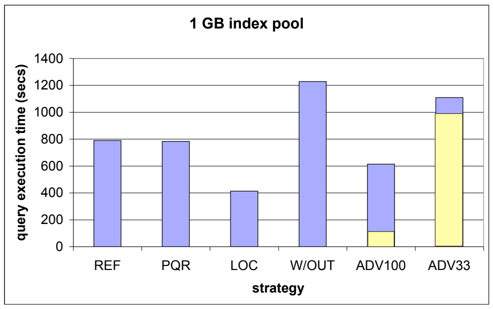
W/OUT: The **W/OUT** strategy executes the queries of workload without additional index creation, i.e. only the primary key indexes exist in the database.

ADV: Here, a workload is executed against a database containing indexes recommended by the DB2 index advisor `db2advis`. As a time constraint we used a 10 minute limit for the brute force approach of the tool for finding an optimal index configuration, because after this time we had a relatively stable recommendation. We distinguish two scenarios: **ADV33**, where we present only 33% of the executed workload to the advisor and **ADV100**, where the complete workload is known to the advisor. The **ADV33** scenario simulates the situation where we know only a fraction of the workload at tuning time.

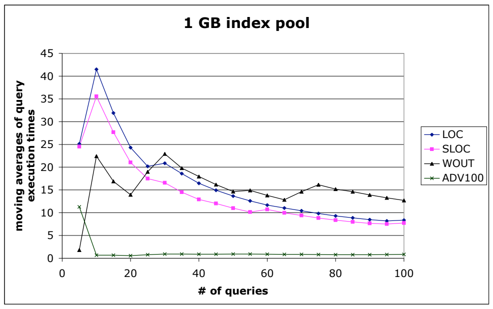
REF: The following strategies are different versions of the **LOC** strategy defined in Section 4. The **REF** strategy corresponds to the reference counting strategy.

PQR: The PQR strategy combines the local strategy with the ratio profit per query approach. The average profit is added to the index statistics. It is the **LOC-PQR-AVG** strategy according to our categorization.

LOC: The last strategy corresponds to the **LOC-PROF** technique with average index profits per query.



(a) Overall times



(b) Moving averages of query execution times

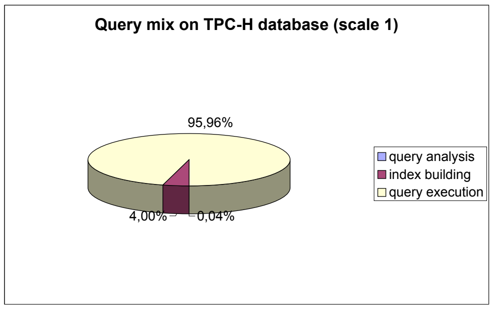
Figure 2. Query execution times

Fig. 2(a) shows the results of overall execution times for workload **W1** on a 100 MB database with an index pool of 1 GB. For comparison the index creation times for **ADV100** and **ADV33** were included into the result, but the pure query execution times of these strategies are shown as light-gray boxes, too. As the diagrams indicate all index creating strategies are faster than the **W/OUT** strategy without indexes. Furthermore, as expected the **ADV100** results in the best execution time but with higher index building costs. However, as soon as the workload is not known to the advisor completely (**ADV33**), the results become worse. Considering both execution and building times our **LOC** strategy produces the best results because it helps to select the right indexes before execution of a query and to replace unnecessary indexes. In addition and in contrast to the advisor it takes the index building costs into account. Experiments with different sizes of the index pool produced basically similar results.

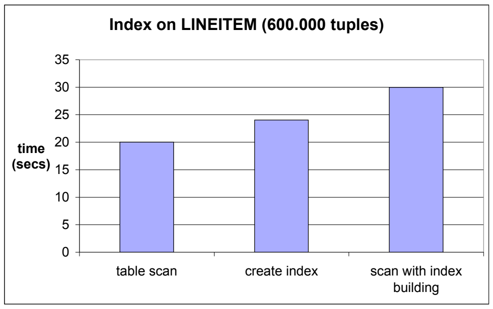
In our next experiments we depict the development of the query execution times in a detailed way. In order to answer this question, we used workload **W2** and evaluated the strategies **LOC**, **ADV100** and **W/OUT**. Thereby, we collected the average query processing time after each processed query. The index creation times of strategy **ADV100** were not included in these times. The results are illustrated in Fig. 2(b) for an index pool of 1 GB. For the first queries the average answer time is relatively high for the **LOC** strategy, because several indexes were created. As the workload represents a randomized TPC-H workload, a relatively stable index configuration should be possible. As the figures indicate, the **LOC** strategy reaches a good configuration, because the average query execution times decrease slightly faster during processing the workload. The results indicate that the adaption of index configuration works well.

Finally, we investigated the overhead caused by index recommendation and building selected indexes. So, we ran the workload **W2** on both the 1.0 scale factor database and the 0.1 scale factor database and measured the times for query analysis (including the index recommendation step provided by DB2), index creation and query execution (Fig. 3(a)). As expected, the time for analysis is constant: in our setting it took approx. 1 second for each query. For long running queries (on the 1 GB database) this overhead is nearly negligible (0.04% in Fig. 3(a)), but for fast queries (in particular on the 100 MB database) the fraction of time required for query analysis is up to 28% (not shown here). Thus, considering only sample queries or expensive queries could reduce the overall time. Good candidates are queries where the costs exceed a certain limit or simply queries where the plan contains full table scans on large tables. In Fig. 2(b) the results of such a strategy (called **SLOC**) are shown, where only the top 10% of queries (regarding the costs) were considered.

The second issue is to reduce the effort for index building because as mentioned above up to 28% of the overall time is spent on creating indexes. Because we had no access to the DB2 source code, we were not able to implement the deferred index feature. However, in order to evaluate this in a practical environment we added deferred indexes to PostgreSQL. Unfortunately, PostgreSQL provides no index recommendation feature, so we have not ported the whole QUIET system. But we can study at least the overhead of index building during query evaluation. Our PostgreSQL implementation supports the CREATE DEFERRED INDEX command as introduced in the previous section. Beside the optimizer modification we extended the scan operator `SeqScan` to deal with index building. The additional source code for all extensions consists of less than 500 lines of code. Using this implementation we measured the times for a full table scan query with and without index building as well as for the corresponding CREATE INDEX command (Fig. 3(b)). This times were measured on the `LINEITEM` table of the TPC-H data set (scale factor 0.1, i.e. approx. 600.000 tuples) for an index on an inte-



(a) Overhead of query analysis and index creation



(b) Times for index building on the fly

Figure 3. Overhead

ger column. Further experiments with other table sizes and indexed columns showed similar results. There is a small overhead of building indexes during a scan, but it performs much better than explicitly creating indexes and execute table scan separately. Thus, in cases were a query cannot profit from an index building deferred indexes is very helpful.

In summary, our experiments have shown the following results: *(1)* Our approach cannot beat the index advisor if the whole workload is known in advance. However, in dynamic environments with evolving and changing workloads the query-driven approach produces better results. *(2)* It is not necessary to analyze each query. Instead, considering only expensive queries and query plans containing full table scans can reduce the overhead of our approach and in this way the overall query execution times. *(3)* We can avoid a separate index creation by using deferred indexes which are created while the table is scanned as part of processing a query. Here, the overhead resulting from the additional "on the fly" index building is rather low.

6. Related Work

Query-driven index building comprises two questions: the selection of an index configuration and index replacement strategies of the index pool.

The selection of an index configuration is an important task of physical database design. However, this problem is considered only during design time in literature and in current practice, for overviews see [19, 10]. Our approach of building indexes during query processing has some different characteristics and challenges: early adaption to a current workload, iterative update of the index statistics, index replacement strategies, and possible usage of table scans of a query for index building. Therefore, there are many similarities to other self-tuning features of a database system, for instance the cache and buffer management.

There are several academic approaches as well as database products for advising index selections [5, 3, 15, 2, 17, 7]. A common approach is the analysis of a workload given by the database administrator or by former queries from a log file. Using this approach several techniques were developed which use either a separate cost model or rely on the optimizer estimates.

The works [3, 8] belong to the class of techniques which make use of a separate cost model. Relying on a stand-alone cost model has the important disadvantage, that the tool cannot exactly estimate the real system behavior. In contrast, an optimizer-based approach works directly with the system's estimations. The work of [8] also deals with adaption to changing workloads at run time, but it is based on its own cost model, in contrast our approach is based on optimizer estimations.

An early realization of the optimizer-based approach is described in [5]. This work describes the design tool DBDSGN, which relies on the System R optimizer and computes for a given workload an optimal index configuration. This approach inspired the index wizards and advisors of current database management systems [2, 18, 1].

The work described in [13] deals with view and index selection in a data warehouse environment. It combines a cost-based method with a set of rules of thumb. The cost-based technique uses an A* algorithm, but it does not sufficiently reduce the search space for real world scenarios, which leads to the rule set. The authors of [7, 9] propose another technique for index selection for OLAP. Here, indexes are considered during the selection of materialized views. The cost model is based on the estimated number of returned rows of a query and is independent from the optimizer. As an optimization algorithm the authors used a greedy algorithm. The greedy behavior prevents the discovery of index interaction, e.g. in a merge-join. Therefore, Chaudhuri and Narasayya included in their index selection an exhaustive search for the best configurations [2]. Furthermore, the algorithm in [2] consists of two phases: enumerating possible configurations from every single query of the workload and subsequently, selecting the final configuration by using the mentioned combined greedy approach with an optimizer-based cost model. The first step is similar to our approach, but the second step of our method is an iterative replacement algorithm instead of an one step operation. The costs are computed by using the systems optimizer and virtual or "what-if" indexes, respectively. The DB2 approach [18] uses another approach than Chaudhuri and Narasayya. In their work the authors include the index selection into the optimizer to utilize its plan selection and enumeration facilities.

Graefe mentioned the vision of index building queries in the context of adaptive query processing [6]. He raised the questions how to build an index during query processing without disturbing concurrent queries. This question was not addressed in detail in our paper, but it will be a research focus in the future. The last point of Graefe's paper issues the question about how concurrent and succeeding queries can benefit from a possible new index. These ideas lead to the field of adaptive query processing (e.g. [4, 11]). Techniques of adaptive query processing could be very useful in combination with our approach.

7. Conclusion

Index tuning is – among others – an important task for fulfilling the query execution time requirements of modern database applications. Today's commercial database systems support this task with so-called index wizards or advisors recommending indexes based on a given workload. In this paper, we argue that this support can be further improved by a tight integration of query monitoring, index selection and building with the actual query processing. Based on a cost model we have presented different strategies for identifying potentially beneficial indexes, maintaining statistics on index profits and deciding about indexes for creating and/or dropping. Furthermore, we have discussed how this approach can be implemented on top of a commercial DBMS providing basic facilities for index recommendation. Though in our implementation we exploit some special features of DB2 the approach basically can be ported to other systems providing similar support for index recommendation. Finally, we have shown evaluation results demonstrating that the proposed techniques and strategies after a further refinement and adjustment to system specifics are well applicable in commercial database management systems. Though we did not expect to beat the performance of index advisors for workloads known in advance, we did exactly this in a number of test runs and came close in most of the others.

References

[1] *Oracle9i Database Online Documentation, Release 2 (9.2)*, 2002.

[2] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'1997*, pages 146–155, 1997.

[3] S. Choenni, H. M. Blanken, and T. Chang. On the Selection of Secondary Indices in Relational Databases. *DKE*, 11(3):207 – 234, December 1993.

[4] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *SIGMOD'1994*, pages 150–160, 1994.

[5] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *TODS*, 13(1):91–128, 1988.

[6] G. Graefe. Dynamic Query Evaluation Plans: Some Course Corrections? *Bulletin of the Technical Committee on Data Engineering*, 23(2):3 – 6, June 2000.

[7] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *ICDE'1997*, pages 208–219, 1997.

[8] M. Hammer and A. Chan. Index Selection in a Self-Adaptive Data Base Management System. In *SIGMOD'1976*, pages 1–8, 1976.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD'1996*, pages 205–216, 1996.

[10] IEEE. *Bulletin of the Technical Committee on Data Engineering*, volume 22, June 1999.

[11] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD'1998*, pages 106–117, 1998.

[12] T. Kraft, H. Schwarz, R. Rantzau, and B. Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *VLDB'2003*, pages 488–499, 2003.

[13] W. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehouses. In *ICDE'1997*, pages 277–288, 1997.

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

[15] S. Rozen and D. Shasha. A Framework for Automating Physical Database Design. In *VLDB'1991*, pages 401–411, 1991.

[16] K. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-driven Index Tuning (Software Demonstration). In *VLDB'2003*, pages 1129–1132, 2003.

[17] B. Schiefer and G. Valentin. DB2 Universal Database Performance Tuning. *Bulletin of the Technical Committee on Data Engineering*, 22(2):12–19, June 1999.

[18] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE'2000*, pages 101–110, Mar. 2000.

[19] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering. In *VLDB'2002*, pages 20 – 31, 2002.