Towards Indexing Schemes for Self-Tuning DBMS

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, based on previous research regarding automatic index creation at runtime we point out the need for new indexing schemes suitable for self-tuning. Based on problems with previous approaches we describe the key concepts, which are sparse and partial indexing, usage-balanced instead of data-balanced structures, and dynamic resource assignment. We illustrate the approach by a simple index structure, which provides adaptability as well as improved access characteristics. Furthermore, we will outline key tasks to introduce according concepts in future DBMS.

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. Besides 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.

While such considerations have been an integral part of physical database design for a long time and are well studied, most often this results in a very high complexity and tremendous efforts required to provide an efficient solution for a given application. During recent years the need for system support on this task became obvious. The paper is structured according to implications from our research and can be divided into the following three sections.

Static index management and recommendation (Section 2): the current state of the art within commercial DBMS includes index wizards or advisors, which help in analyzing single queries or complete workloads and deriving the recommendation of an index configuration for a given application. This is done by solving an optimization problem based on cost estimations for query processing with potential sets of index candidates. Nevertheless, this task remains a responsibility of the system designer or administrator and has to be carried out frequently to adjust to changing usage, data, and environment.

Dynamic index management and autonomous tuning (Section 3): in our previous research we investigated whether this task can be carried out autonomously and continuously during runtime. For this purpose, we implemented a component gathering and updating statistics based on which decisions about changing the current index configuration can be made. This approach was implemented and showed very encouraging results during evaluation. Despite the overall positive results, we received some critical feedback about the involved overhead and lack of control of the system behavior during query processing.

Dynamic self-tuning index structures (Section 4): as a possible consequence of the previous drawbacks, we currently consider moving the task of index tuning one step down, i.e. from the level of index configurations down to the indexes themselves. If the indexes could in some way adjust to their usage, e.g. grow with increasing or shrink with decreasing numbers of accesses, this would lead to a much more fine-grained control of self-tuning within the system.

We will conclude the paper by giving an overview of related work in Section 5 and, finally, describing conclusions in Section 6.

2. Static Index Management and Recommendation

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 [9].

The most current releases of the major commercial DBMS such as Oracle10g, IBM DB2 Version 8, and SQL Server 2000 provide (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 – 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 still work 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 mentioned 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. 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. Furthermore, a collected workload is very likely incomplete and biased by the impact of collecting the workload on a not fully optimized database. In addition, the necessity of index configuration changes may be triggered by changes of the hardware involved, like CPUs, main memory, and disks containing the database. All this renders the index tuning task as an ongoing process with a continuous need of DBA interaction. Index advisors do not change this, they just minimize the required human input of expert knowledge.

3. Dynamic Index Management and Autonomous Tuning

Our previous work regarding autonomous query-driven index tuning is based on the introduced concepts of static index recommendation, but in addition proposes the autonomous adjustment of index configurations during runtime. The overall approach and architecture is outlined in [13] while in [14] the full details and evaluation results are given.

Processing Index-building Queries

At first, we will sketch the overall process of executing index-building queries. The main objective of our 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.

(2) For query Q a cost-optimized query plan is derived.

(3) The query Q is re-optimized using the virtual indexes based on step (1).

(4) The profits of sets of virtual indexes are computed as the difference of the costs of the plans from step (2) and from step (3). The index set with the highest profit is called index recommendation and is used to update a global index configuration where cumulative profits of all indexes (both materialized and virtual) are maintained. From this index configuration we have finally to decide about: creating indexes from the virtual index set, and 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.

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).

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 $i_1, \dots, i_n$ 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 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* $\mathcal{D} = \{i_1, \dots, i_k\}$. 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 profit of an index set according to a query can be calculated in different ways, as outlined in the following. 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.

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 $\text{benefit}(\mathcal{C}) = \sum_{i \in \mathcal{C}} i.\textit{benefit}$. In addition, we have to take into account the cost building the new indexes $\text{cost}_{\text{build}}(i)$ which appear as negative profit:

$$\text{benefit}(\mathcal{C}_{\text{new}}) - \text{benefit}(\mathcal{C}_{\text{curr}}) - \sum_{i \in \text{virt}(\mathcal{C}_{\text{new}})} \text{cost}_{\text{build}}(i) > \textit{MIN_DIFF}$$

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 [11].

Criteria for Index Selection

As described before, 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.

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 dd 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 approximation of configuration benefits.

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

$$\text{benefit}(\mathcal{C} \cup \mathcal{I} \setminus \mathcal{I}_{\text{repl}}) - \text{benefit}(\mathcal{C}) - \sum_{i \in \text{virt}(\mathcal{I})} \text{cost}_{\text{build}}(i) > \textit{MIN_DIFF} \wedge \text{size}(\mathcal{C} \cup \mathcal{I} \setminus \mathcal{I}_{\text{repl}}) < \textit{MAX_SIZE}$$

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, or check all beneficial index sets for a possibly **globally optimal** configuration.

The replacement index set $\mathcal{I}_{\text{repl}}$ is computed using the currently materialized 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, the **cumulative profit** of an index, and the **ratio of profit per query** or reference of an index. 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 or scheduled to be changed later on.

In our implementation and experiments we only considered strategies based on locally optimal index sets. 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.

4. Dynamic Self-tuning Index Structures

The approach for self-tuning index configurations described in the previous section provides a solution for continuous tuning on the level of index configurations, where configurations are a set of common index structures. In this section we will motivate our current research approach, that moves the solution of the problem at hand to the level of the index structures. Though the impact on DBMS implementations is quite severe in this case, we will illustrate that more

efficient and more organic solutions can be realized by rethinking the kind of indexes used in DBMS.

Motivation for new Index Structures

While indexing in databases using derivatives of B-Trees, R-Trees, Grid Files, Tries, and many others has been applied successfully for years, obviously these index structures were not conceived having self-tuning in mind. This is for the following reasons.

Data-balanced instead of access-balanced structure: with current DBMS index structures data access is optimized for all existing values of a certain, possibly multidimensional, range/domain. This is done to achieve $O(1)$ or $O(\log\ n)$ complexity for accessing a single data unit, i.e. a page, tuple, object, etc. Nevertheless, this is done not considering if the data unit is accessed very often, rarely, or not at all. Hence, the benefit of the index structure is greater for data accessed more frequently than for data accessed less often, though both come at the same price, i.e. used system resources.

Coarse granularity of tuning: with current index structures it is either all or nothing: an index can be created on an access path or not. Though the system may benefit for instance from partial indexes as proposed by Stonebraker in [16] or incomplete and sparse indexes, this is not implemented in most current DBMS. Furthermore, compared to the size the overall benefit of an incomplete index grows with $O(\log\ size)$, i.e. the bigger an index grows the more the gained profit $\Delta benefit/\Delta size$ declines. Considering the often tremendous disk space requirements, index structures suitable for self-tuning should be able to grow or even shrink on demand.

Unawareness of system resource usage: related to the previously mentioned aspects, currently used index structures are not suitable to consider space limitations during runtime. Though index sizes can be estimated very precisely before creation, the index itself cannot comply to space restrictions during runtime, which would be required to realize adaptive index configurations.

A Simple Adaptable Binary Tree

To illustrate a possible approach and get an overview of all the related issues we implemented a simple index structure based on a binary tree which adapts to its usage during runtime, by

1. having a **adjustable size**, i.e. number of nodes, that can be increased or decreased during runtime, which results in a sparse tree with page containers as leaves, and
2. being **access-balanced**, i.e. the tree is deeper for data regions which are accessed more often to minimize page accesses.

An example tree is sketched in Figure 1.

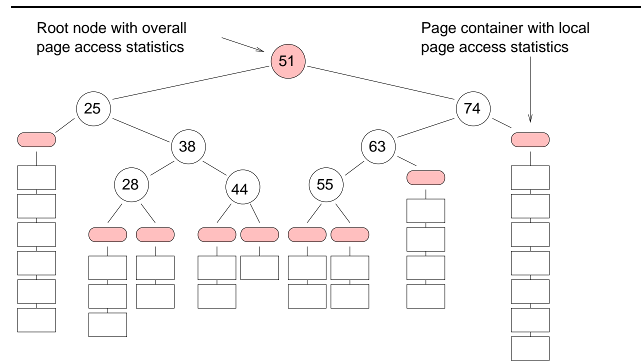

Figure 1. An access-balanced binary tree

The sparse index has page containers as leaf nodes, containing pages of tuples with keys of the value range specified by the tree. This way, currently data is organized by the index. The implications for secondary indexes and multidimensional indexing schemes are discussed later on. Nevertheless, the current approach allows assigning system resources to the tree by increasing or decreasing the allowed number of nodes within the tree. The lookup in the tree works as usual, whereas the lookup in the page containers requires a sequential scan through the pages.

The reorganization and access-based balancing can only be sketched roughly due to space limitations. In principle it works as follows: the root node keeps track of all page accesses and the current size of the tree. Accordingly it computes a balance

$$balance = \frac{all\ page\ accesses}{number\ of\ page\ containers}$$

for all page containers. For the page containers pc the number of all $pageAccesses(pc)$ is stored. If a leaf node with two page containers pc_1 and pc_2 fulfills

$$pageAccesses(pc_1) + pageAccesses(pc_2) < balance$$

it is removed and added to a pool of free nodes, which is maintained as a number of free nodes in the tree root. The two page containers are merged and linked to the predecessor in the tree. If the number of page accesses of a single page container fulfills

$$pageAccesses(pc_1) > 2 * balance$$

and, furthermore, there are free nodes in the node pool available, a new node is inserted with the median key from the container, and the container is split into two according new containers. Growing the tree works simply by increasing the free node pool variable and relying on the previously sketched reorganization algorithms. Shrinking can be done the same way, but also may be forced to immediately regain resources by removing nodes and merging their page containers having the least page accesses.

Currently, these reorganization as well as statistic updates are performed during lookup operations. For reasons discussed later on they may also be deferred to avoid concurrency problems.

Based on this reorganization schema, the tree can adjust to size restrictions as well as data usage, e.g. the example tree in Figure 1 may index data with keys uniformly distributed in the range of 1 to 100, while accesses are normally distributed with a mean key value of 50, i.e. there are far more accesses in the middle of the range. Therefore, the tree is deeper towards this range and has smaller page containers to grant efficient lookup of often accessed data.

For experimental evaluation we tested a similar scenario of uniformly distributed data (100000 tuples, approx. 100 tuples per page, tree with 1000 nodes) and normally distributed exact match lookups.

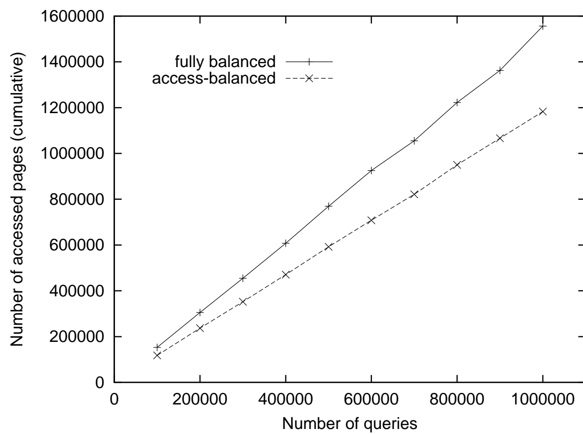

Figure 2. Comparison of page accesses

In Figure 2 the results for page accesses are shown. Because it would not make sense to compare our approach to a dense index, we instead compare to an alternative implementation of a fixed size tree without the automatic access-based balancing, which is fully balanced from the beginning. Though in both cases the complexity of a single lookup remains in the order of $O(log\ n)$, the access-balanced tree outperforms the fully balanced tree with a linear factor because access to more often used data is more efficient.

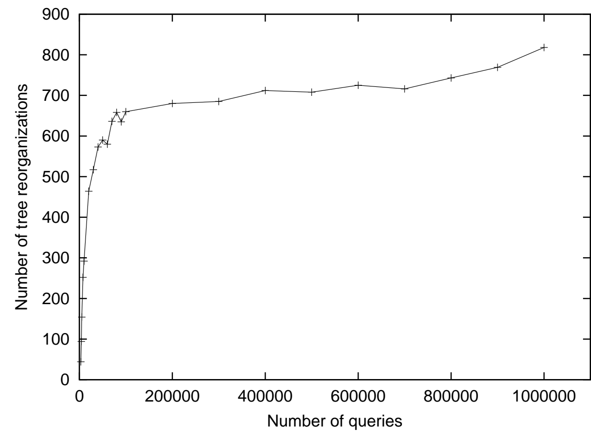

Figure 3. Number of tree reorganizations

In Figure 3 we illustrate the number of tree reorganizations triggered by the number of exact match lookups. After ca 100000 lookups the tree reaches a relatively stable structure with very few reorganizations required afterwards.

Nevertheless, we could not yet quantify the major advantage of the proposed approach: instead of choosing to create indexes within a configuration or not, we now can assign space to indexes where needed within a more complex indexing schema. And, as discussed before, even small indexes may provide a huge benefit. This way, having a great number of those small indexes is preferable to having a small number of very huge indexes. An evaluation of these aspects, apart from solving the problems described in the following, requires a real life scenario with multiple tables, columns, etc., e.g. the TPC benchmarks and a full implementation of the global indexing schema. Because we cannot build our test environment based on available DBMS technology due to the strong difference of our approach, this is a very complex task we are dealing with in our current research.

Open Issues

It is quite obvious that there are a great number of open problems and questions with the approach, which we are currently trying to resolve. For our future work we will focus on the following aspects:

An overall indexing schema: the basic ideas outlined before mostly only refer to new indexes, which must be the building blocks of an indexing schema covering the task of index tuning for a full database, for instance by deciding about whether a certain index makes sense at all or assigning space resources to indexes. A simplistic approach would be to support all indexable columns with indexes, and control the tuning only via resource assignment, e.g. the previously presented binary tree

also allows trees with size 0, i.e. no nodes and only a list of pages.

Data structures suitable for DBMS storage: of course binary trees are not the optimal choice for a DBMS index structure. Because we balance based on access characteristics and not on the data distribution, the only interesting property of B-Trees is the fixed node size that fits well with the storage management of a DBMS. If we want to support multiple (or even all possible) access paths, the best solution are multidimensional index structures, such as Grid Files or kdB-Trees. Nevertheless, these structures will have to be adjusted to be access-balanced and to serve as sparse indexes to support resource assignment.

Index-organized data vs. secondary indexes: to support multiple access paths in current DBMS most often secondary indexes are used. These can cause a tremendous space overhead for sparse indexes, because possibly very long lists of tuple identifiers are required even for small indexes. To solve this issue, our current research again focuses on multi-dimensional index structures which are used to organize data on secondary storage similar to Grid Files. As an alternative we consider Bitmap indexes, but for these the well-known problems regarding the update granularity would have to be resolved.

Concurrency issues: concurrency issues can result from keeping often updated statistics within the index structures. For instance, the points where statistics are kept within the binary tree are marked grey in Figure 1. Our goal is to isolate such hot spots as much as possible. On the other hand, the statistics do not necessarily underly strong transaction requirements and therefore can be managed by the DBMS separately. Further concurrency problems relate to the reorganization of data and index structures, which may also be triggered by read-only accesses. A possible solution would be deferred reorganizations.

Operations based on access-balanced index structures: as a consequence of new index structures, index based operations and their implementation have to be investigated. Simple examples would be range scans, sorting and ordered scans, merge joins, etc.

Possibilities of extending commercial DBMS: because the basic principles of the proposed approach are quite oppositional to index management and usage in current DBMS, the integration of according concepts in commercial solutions will not be an easy task. Therefore, we investigate the transfer of partial results, for instance based on partial or multi-layered indexes, into existing systems.

Evaluation: as pointed out before, the introduced concepts require a thorough evaluation with a complex test environment. On the one hand, index structures must be evaluated regarding the advantages of access-based balancing as well as the overhead and speed of reorganization. Finally, the concept of sparse indexing must be evaluated separately to quantify the performance of the overall indexing schema.

5. Related Work

Index tuning in general – whether carried out by experts, supported by tools, or even accomplished autonomously by the DBMS – always comprises the selection of an index configuration providing the highest possible system efficiency. Autonomous index tuning furthermore requires support for index replacement strategies.

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 [18, 8]. 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 [4, 3, 12, 2, 15, 5]. 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, 6] 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 [6] 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 [4]. 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, 17, 1].

The work described in [10] 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 [5, 7] 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.

6. 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 our previous research we argued 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 presented different strategies for identifying potentially beneficial indexes, maintaining statistics on index profits and deciding about indexes for creating and/or dropping. Furthermore, we discussed how this approach can be implemented on top of a commercial DBMS providing basic facilities for index recommendation. Though, in our implementation we exploited some special features of DB2 the approach basically can be ported to other systems providing similar support for index recommendation.

Finally, to overcome some problems with the previous approach and provide a more natural solution we proposed moving the index tuning problem from the configuration layer to the index layer itself. For this purpose, we illustrated some advantages of the new approach based on a simple, flexible, and access-balanced binary tree index. Because this is research work that started only recently, we gave an overview of the issues to be considered for self-tuning index structures.

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

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

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

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

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

[9] 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.

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

[11] 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.

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

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

[14] K. Sattler, E. Schallehn, and I. Geist. Autonomous query-driven index tuning. In *Proc. Int. Database Engineering and Applications Symposium (IDEAS 2004), Coimbra, Portugal*, pages 439–448, July 2004.

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

[16] M. Stonebraker. The case for partial indexes. *Sigmod Record*, 18(4):4–11, Dec. 1989.

[17] 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.

[18] 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.