Autonomic Physical Database Design - From Indexing to Multidimensional Clustering

Stephan Baumann, Kai-Uwe Sattler

Databases and Information Systems Group
Technische Universität Ilmenau, Ilmenau, Germany

Abstract—Database design is a well-studied and well-understood problem in developing database applications. However, modern application requirements in terms of query response time, query throughput, database sizes, database maintenance and database interactivity capabilities on the one hand, and the manifold features of modern DBMS for physical schema design such as numerous different index structures, materialized views, partitioning, and clustering schemes on the other hand make high demands on the database design process. With this growing number of impact factors on physical database design automatic solutions have been developed and are provided in most systems, while research in this area still continues. In this paper, we give a survey on the state of the art in autonomic physical design. Starting with the classic index tuning problem and possible solutions, we describe further design problems such as choosing materializations of aggregations for OLAP and multidimensional clustering schemes. Finally, we point out to most recent challenges resulting from a shift in requirements and give a direction on how to tackle these.

I. INTRODUCTION

Modern enterprise applications are typically characterized by high demands regarding data volume, query response time, and transaction throughput. In order to fulfill these requirements database design and tuning play important roles. However, the complexity of modern DBMS with hundreds of tuning knobs require deep knowledge about system internals, workload, and data characteristics as well as expertise and tremendous effort by the DBA. This problem is even aggravated in dynamic scenarios with changing workloads and requirements which would require a continuous adjustment of the system.

Studies from the year 2001 have shown [1] that approx. 80% of the total cost of ownership for database applications are spent on administration (training, deployment, DBA cost) – an observation that has stimulated the research and development of autonomic (also known as self-*) techniques for database systems. Techniques considered so far cover different levels ranging from the physical level with indexing, clustering, and partitioning over the conceptual schema level (with denormalization and materialization) and the system level (memory and bufferpool tuning, statistics and reorganization) up to the external application level. Features of these levels require different techniques, have different impact on the performance and come at different costs and decision intervals: whereas adjusting the sizes of the memory pools of the DBMS can be done at runtime in a couple of seconds with a direct impact on the following queries, changing the physical schema takes typically much more time and requires higher effort.

In addition, physical database design is a particular difficult problem: modern systems provide numerous storage and indexing options with many parameters and complex database schemas offer many opportunities, e.g. for creating indexes. Furthermore, there is often a tradeoff, e.g. between performance and space consumption or between query performance and update/maintenance costs.

Finally, more recent developments in the database field such as on column stores, in-memory processing, but also on NoSQL systems and large-scale parallel processing platforms seem to simplify physical database design, for example by making indexes obsolete for column stores. However, recent research works reintroduce access paths and storage options even in such systems: examples are min-max indexes for column stores [2], indexing for Hadoop [3] or considering other issues related to physical design such as memory layouts mitigating NUMA effects [4]. All this together raises the conclusion that the problem of physical database design is still an important problem requiring autonomous features.

In this paper we summarize contributions to autonomous database design In Section II we formulate the general problem of database design and sketch the search space. In Section III we first introduce the problem of index selection and point to important work in this field. Section IV looks at the problem of materialization in the context of database design. In Section V we cover the field of multidimensional clustering, in Section VI we take a look at benchmarking approaches for automatic design and in Section VII we summarize our observations and point to future directions.

II. THE PHYSICAL DATABASE DESIGN PROBLEM

Physical database design is the process of defining the physical part of a database schema. This includes storage structures, storage parameters, but also additional access paths such as indexes, partitions, summary tables, etc. Though, these structures and their parameters are mostly transparent to the application level, typically they have a significant impact on the performance of the applications and, therefore, are in fact an important subject of database tuning. The problem of physical database design can be formulated as follows. Given a database schema $D$ and a workload $W$ of queries/updates, find a configuration $C$ of physical database objects/parameters.
for $D$ which minimizes $\text{cost}(W)$ under some constraints, e.g. not exceeding a certain disk space.

However, the very big variety of possible data structures, techniques, and parameters – even in commercial systems – make this a complex and challenging task. Therefore, significant efforts have been made over the last years to automate physical design.

Physical database design tasks can be classified roughly according three dimensions:

- **What?** describes which database objects are considered. This includes indexes, table partitions, summary tables, etc. as well as combinations of them. Furthermore, interaction relationships exist between these objects which have to be taken into account.

- **When?** defines the time of creating or modifying a physical schema. Possible solutions are either static approaches which create a schema only once or after explicit invocation by the database administrator or dynamic approaches which adjust a schema continuously (or periodically) at runtime.

- **Why?** specifies the reason for a design decision. This can be based on heuristics or on cost models capturing the expected benefit wrt. to a given workload. Furthermore, the necessary information used for this decision can be obtained by analyzing the workload $W$ (or a sample), additional schema information (e.g. including integrity constraints) or only the database schema $D$.

Based on these dimensions, physical database design is often modeled as an optimization problem, either as static optimization, e.g. as a knapsack or integer programming problem, or an online optimization problem.

### III. INDEX TUNING

One of the most prominent and also well-studied physical design task is index tuning. Index tuning addresses the Index Selection Problem (ISP) \[?]\. In a simplified and typically used version it can be formulated as follows. Given a set of queries $Q_1, \ldots, Q_m$ and a set of possible indexes (index candidates) $I_1, \ldots, I_n$ where $\text{cost}(Q_k)$ denotes the execution costs for $Q_k$, $\text{cost}(Q_k, I_k)$ the execution costs for $Q_k$ when index $I_k$ is present, $\text{mcost}(I_k)$ denotes the maintenance costs for $I_k$ (e.g. for updates) and $\text{size}(I_k)$ the storage space occupied by $I_k$.

Furthermore, the profit of index $I_k$ for a given query $Q_k$ is the difference between the query execution costs with and without the index, i.e.

$$\text{profit}(Q_k, I_k) = \max\{0, \text{cost}(Q_k) - \text{cost}(Q_k, I_k)\}$$

Based on this, the ISP consists of finding a subset of all possible indexes $C \subseteq \{I_1, \ldots, I_n\}$ which is materialized (called an index configuration) and maximizes the profit for all given queries:

$$\sum_{i=1}^{m} \max_j \{\text{profit}(Q_i, I_j) : I_j \in C\} - \sum_{I_j \in C} \text{mcost}(I_j)$$

while still satisfying given index space constraints $S$:

$$\sum_{I_j \in C} \text{size}(I_j) \leq S$$

This problem can be seen as a special case of the knapsack problem \[?]\. However, calculating the profit only for a specific index per query is only an approximation of the real problem. Typically, there are dependencies between indexes that are used together in a certain query and, therefore the exact profit can only be defined in terms of index sets $I \subseteq \{I_1, \ldots, I_n\}$. These dependencies are called index interactions and should be taken into account, too.

Unfortunately, ISP is an NP problem \[?]\; the number of possible indexes for a table with $n$ columns is \[?]\;

$$\sum_{k=1}^{n} \frac{(2^n)n!}{(n-k)!}$$

Thus, solving the ISP manually is feasible only for very small database schemas.

However, various approaches to autonomously approximate the best configuration have been proposed. These autonomous approaches for index tuning can be classified into the following categories \[?]\:

- static index management and recommendation,
- online index management,
- self-tuning index structures.

Static index recommenders have been introduced around the year 2000 by the major commercial DBMS vendors, e.g. in Microsoft SQL Server \[?]\, \[?]\, IBM DB2 \[?]\, and Oracle. These add-on tools analyze a given query workload to derive recommendations for index creation together with the estimated cost benefit. This is typically implemented using “hypothetical” objects – in this case virtual indexes – which exist only in the catalog for the query optimizer but are not physically created. Possible virtual indexes or index candidates are identified from the relevant clauses of the SQL queries such as \textsc{where}, \textsc{order by}, \textsc{group by}, and \textsc{select}. For the relevant columns and their combinations, virtual indexes are created. The profit of these indexes is estimated as shown above by optimizing the query twice: with and without the index set. The profit is cumulated for all queries of the workload and the ISP is solved, e.g. using a modified knapsack solution, integer linear programming or approximative approaches. Several approaches have been proposed to improve the accuracy of the estimations, e.g. by taking index interactions \[?]\ and interactions with other physical structures \[?]\ into account, to reduce the number of optimizer calls \[?]\, or to consider further constraints in addition to disk space \[?]\.

However, though index recommenders exploit runtime statistics of the given workload, they usually still work in design mode: the DBA has to decide whether and when the recommended indexes are created. Furthermore, the recommended index configuration is optimal only for the given workload. When database use changes over time – which is often the case, e.g. caused by seasonal effects, changing interest
Plan P can be formulated as follows. Given a workload $W$ of days for skiing is not known in advance, the question is configuration, and a configuration schedule $S$ such algorithms are the ski rental problem or the ice cream without knowing the entire input. Well-known examples for by online algorithms processing their input piece-by-piece optimization problem. Problems of this class can be solved increase the total profit.

For online index tuning, the problem described above is to maintain the virtual indexes and update their cumulative profit over time while processing the queries. Based on this information, decisions about changing the currently materialized index configuration $C$ can be made, e.g. if adding new index $I_{new} \not\in C$ to $C$ or replacing $I_{old} \in C$ by $I_{new} \not\in C$ would increase the total profit.

In this way, online tuning can be modeled as an online optimization problem. Problems of this class can be solved by online algorithms processing their input piece-by-piece without knowing the entire input. Well-known examples for such algorithms are the ski rental problem or the ice cream vendor problem [2]: under the assumption that the number of days for skiing is not known in advance, the question is whether to continue to pay a repeating cost for renting skis or to buy skis and just pay a one-time cost.

For online index tuning, the problem described above can be formulated as follows. Given a workload $W = \langle Q_1, Q_2, \ldots, Q_m \rangle$ as a sequence of queries, $C_i$ an index configuration, and a configuration schedule $S = \langle C_0, C_1, \ldots, C_m \rangle$ denoting that $Q_i$ is processed on $C_i$ at cost $cost(Q_i, C_i)$. Now, starting with an initial configuration $C_0$, the goal is to find an optimal schedule $S^*$ that minimizes the costs:

$$S^* = \arg \min_S cost(W, S)$$

where $cost(W, S)$ represents the total execution cost for $W$ in $S$ as well as the transition costs between two configurations (i.e., materializing and/or remove indexes):

$$cost(W, S) = \sum_{i=1}^{m} cost(Q_i, C_i) + transition(C_{i-1}, C_i)$$

However, estimating costs for alternative plans, maintaining performance statistics for virtual indexes, and finding the optimal configuration cause a significant overhead and it is not suitable to perform these steps for every single query. The work demonstrated in [7] addresses this issue by gathering statistics at several levels of detail and using heuristics to regulate the overhead, e.g. re-tune more frequently in case of shifting workloads or reduce activities for stable workloads. Furthermore, issues like throttling and oscillation have to carefully be taken into account [7].

Another problem is related to acceptance: transition to a new configuration means to create new indexes which could result in unexpected delay of queries triggering the transition. Also, DBAs are typically very careful with changing configurations of a production system and, therefore, accept fully autonomous techniques only slowly [7].

One step further to address these problems is to push down the task of index tuning from the level of index configurations to the index structures themselves. This could mean to piggyback index building with queries or even a more fine-grained control by adjusting the index structure to their usage, e.g. grow with increasing number of accesses.

In [2] we have presented the concept of index-building queries. If a necessary change of the current index configuration was detected which pays off even by taking the transition cost into account, the recommended indexes are created in a deferred mode. Now, as soon as a query is executed that performs a scan on the table underlying a deferred index, the query execution plan is modified to populate the index while scanning the table and make it available for usage. However at this point, the index-building query does not benefit from the index. For this purpose, we have introduced in [2] an approach that allows to switch execution plan after the index is populated.

While this approach still treats indexes as atomic objects which can be constructed and used only as a whole, the work on adaptive indexing described in [7] breaks this down to incrementally build and maintain indexes. Adaptive indexing is based on the idea of database cracking [7] combined with adaptive merging. As our index-building queries, database cracking aims at making index creation and maintenance a byproduct of query processing. To achieve this, queries are interpreted also as an advice to “crack” the physical database into smaller pieces. These pieces are then assembled in a cracker index to speedup subsequent queries. For adaptive indexing, a new cracker index is initialized when a column is queried by a predicate for the first time. Then, driven by subsequent queries with predicates on the index column, the cracker index is refined by range partitioning - leading to a form of incremental quicksort. The work described in
combines this idea with adaptive merging where the first query produces already sorted runs which are later merged by following queries. In this way, indexes become self-tuned and self-optimized data structures which could help to lower the burden for index maintenance and eliminate the need of explicit tuning.

However, as we will see in the following sections, this addresses only one aspect of the problem space of physical database design.

IV. MATERIALIZATION

OLAP queries typically access large chunks of data and perform complex joins and aggregations. Because of that it may become feasible to materialize calculations on the raw fact data in order to speed up (later) query execution. Different names have been given to this idea: materialized views, aggregation tables, materialized query tables or indexed views. The idea behind it is simple: Invest some (extra) work, that commonly needs to be done for many queries and benefit from that during query execution. The additional cost of the investment is then amortized over the execution of multiple queries. That means in this case it is important to have queries with at least partially shared workload, which is typical for OLAP scenarios.

Similar to the index selection problem, a view selection problem (VSP) can be defined. Let the view candidates be \( V_1, \ldots, V_n \) instead of index candidates \( I_1, \ldots, I_n \) and the VSP can be defined analogous. Again, the profit for the selected views is to be maximized while still holding the space constraint.

Research on view materialization has evolved from static to dynamic approaches, where the lifetime of a materialized view has shrunk during this evolvement. In the static case a typically cost-based decision based on workload analysis is made on which views to materialize and in the dynamic case a starting set of materialized views is chosen and is maintained and evolved over time based on the analysis of the current load. The latest approaches have become more fine-grained, where instead of full views only potentially interesting intermediate results of query operators (that typically are already materialized anyway) are kept in a buffer to accelerate queries in a nearer future.

Research on static approaches led to some form of recommendation tools found in the various commercial products \([7], [8], [9]\). The original recommendation tools worked on a representative workload and the process was for all of them similar: generate candidates from the representative workload, prune the candidate set, rate the candidates based on a cost model and recommend the most valuable candidates.

For example \([2]\), where syntactically relevant candidates of views are generated, based on analyzing the SQL queries. From these candidates an interesting subset is chosen as too many candidates are generated otherwise. This choice is based on finding interesting table subsets in the candidates, i.e. candidates that cover a tables subset and thus can potentially reduce costs when this subset needs to be accessed. After this pruning step a detailed optimizer-based cost analysis over the representative workload is performed for the remaining candidates, where potential gain of each view across a representative set of queries is calculated. A problem of materialized views aside from the materializing costs itself is space consumption. For this reason \([7]\) in addition tries to merge candidate views with the goal of reducing space consumption but keeping query performance at about the same level. In the end the top k views are chosen as recommendations and the DBA has to choose which ones to deploy. With the variety of database objects, and restrictions on space these advisors typically take many different objects into account and recommend a set database objects of different kinds.

As the static approaches could not deal with changing workloads, more dynamic approaches became relevant, leading to design alerters or design recommenders that are always on. In addition to an initial representative workload, the actual workload is monitored and the recommendations are adapted according to this. Some solutions treat the workload as a set \([7], [8]\) or others argue to exploit sequence information \([7]\) in order to generate recommendations with more precision. However, in order to exploit sequence information, the query sequence needs to be known in advance to a certain extent. Otherwise no reasoning about when and how to change a view configuration during the sequence would be possible. In both cases the general process is similar: find an initial setup based on an initial workload, monitor the current workload and re-evaluate the current design, suggest design changes if the current workload requires it. Commonly these optimizations are done in parallel in case of indexing. However, with the advantage of having views for the current workload comes the disadvantage of paying for creation parallel to the workload.

All previously explained approaches still lead to recommendations and required offline invocations of either the design tool or at least a reaction to the alerters’ output. To overcome these limitations, approaches to continuously monitor workloads and adapt the design on-the-fly were developed. \([7]\) however focuses only on index creation. In \([7]\) we have

![Fig. 2. Slice of an example lattice with customer and time dimension. Solid lines are direct relationships, dotted lines represent indirect relationships.](image)
presented strategies based on the analysis of MDX queries and are applied in the Mondrian [?] system. Here, based on dimension hierarchies, an aggregation lattice for a cube is created, where the lattice edges represent dimensions with dimension levels and each crossing point (node) in the lattice represents a possible aggregation table. Figure 2 shows an example slice of such a lattice for two dimensions, customer with a hierarchy from region to street and date with a hierarchy year to day. The solid line arrows denote which node can be derived from which node, i.e. (4,1) can be calculated from (3,1). For node (3,2) we show dotted line arrows, that denote indirect relationships, i.e. indirect ways to derive the node (3,2). Based on workload monitoring nodes are added to the lattice, if a corresponding query has been issued or if a node is a common base node for two others, i.e. the two other nodes’ aggregation tables can be derived from the common base node. Workload cost is monitored and estimated for materialized and non-materialized nodes of the lattice and based on heuristics configuration changes are invoked.

But even in the online case someone has to pay for the materialization of an aggregation table, either the query responsible for triggering the materialization or subsequent queries as the table is materialized in the background. In addition, maintenance costs for materialized views need to be taken into account as they can explode with the number and size of the views.

A last approach in this category that is able to ignore materialization and maintenance costs is recycling intermediate results. One may argue that this is not strictly database design, but we argue that it is worth viewing intermediate results as database design objects. The idea behind it is simple: keep interesting intermediate results in a buffer, which can be memory or disk, and reuse these for subsequent similar queries. In [?] an architecture for an operator-at-a-time column store is presented and in [?] the idea is taken further to pipelined query execution. Both approaches rely on keeping intermediate results that are materialized anyway during query execution, in the operator-at-a-time model this happens by default. In the pipelined approach recycling is based on matching subtrees of the query DAG with corresponding intermediates in the recycler graph. The challenge of course is to select the subtrees for which an intermediate should be kept. Also, taking the recycler cache to a hierarchical structure so it matches the memory structure is not considered. And thus, these techniques can only partially be viewed as part of database design.

V. MULTIDIMENSIONAL CLUSTERING

In addition to traditional indexing and materialization strategies, multidimensional indexing or clustering techniques have been developed, e.g. [?], [?], [?]. These schemes index data according to multiple keys and actually cluster tuples with common characteristics on disk. By indexing according to multiple keys, beneficial access is achieved across a broader range of queries, when compared to a single clustered index, especially when multiple indexing keys are involved in executing a query. However, if only a single index key is relevant, a specialized clustered index on this key is likely to perform better.

An automated design approach for Multi-Dimensional Clustering in DB2 (short MDC) is described in [?]. Here, potential dimensions, or clustering keys, are identified by the optimizer analyzing the SQL queries. Next, a “what-if” analysis is performed in the optimizer based on simulating the query for the highest possible benefit per dimension according to the predicates of the SQL query. As dimensions can be used at different granularities, the benefit of each dimension at each granularity level is calculated based on a logarithmic function model. Next, a set of candidate clustering keys is generated based on the dimensions with their various degrees of coarseness. The keys in this set are ranked and in a greedy approach the highest ranked key that satisfies the storage constraints is picked. However, this analysis is done table by table, though considering interdependent dimensions, and leads to a set of clustering keys per table from which a multidimensional clustering, again per table, is derived in the end.

We have taken the idea further when designing a schema for our approach of co-clustered tables [?]. Rather than considering a multidimensional clustering table by table, we generate a multidimensional clustering for a whole database schema. Of importance here is, that we take the foreign key references in the original schema into account when creating the co-clustered schema. This way we provide not only an access path optimized database design but also a query execution optimized database design by explicitly considering joins to be important for query execution. In [?] we show that query processing (more precisely, operator execution for aggregation, grouping, sort and join) can greatly benefit from database design in such a multidimensional setup.

Figure 3 sketches the idea of co-clustering for a setup with three different fact tables and three dimensions. Ideally all dimensions are used along all foreign key join paths in such a setup. This allows for efficient selection pushdown but also for efficient processing of joins, sorts and aggregations, for details see [?]. For example, the time dimension is used to co-cluster tables A and B. The decision on using this dimension in both tables is based on the foreign key relationship of A and B (FK_A_B). As a result, time selections can be pushed down to both tables and in case of a join the co-clustered layout from the time dimension contributes to the sandwiched execution of the join of A and B. Sandwiched execution here is similar to a partitioned execution of the join and matching partitions stem from the co-clustering. Also, aggregations/groupings based on a time attribute or an attribute functionally determining a time attribute can be executed in a partitioned fashion, for more details we refer to [?].

A database design is then created in three steps. First dimensions need to be identified. Here, we currently do not rely on query analysis for finding dimensions but rather just take DBA hints, formulated as CREATE INDEX statements. Second, these dimensions are propagated through the original schema graph according to the defined foreign key relationships as illustrated in Figure 3. And third, each table is created autonomously
based on the assigned dimensions from the second step, where the granularity of the clustering is auto-tuned according to the data distribution and the disk access block size. In the third step a round robin assignment of the dimension bits insures that all dimensions contribute to the clustering, which later provides the benefits for data access and query processing.

By integrating database design and query processing, or better support query processing by the database design, we achieve a design that we call workload agnostic. From our perspective this means that it is able to serve a broad range of queries, not always to the optimum, but with a significant contribution for very many different queries. In addition we view a co-clustered design as the primary physical design of a database that introduces only a marginal space overhead and does not introduce additional maintenance as extra indexes or materialized views would do.

VI. BENCHMARKING THE PHYSICAL DESIGN

In this paper we have looked at various autonomous database design approaches, from traditional and adaptive indexing over static and dynamic materialization strategies to multidimensional clustering. Each of the discussed approaches has its particular use case and this way its right to exist. However, comparing the different approaches is quite difficult. In [2] ideas on how to benchmark design recommenders are proposed and the authors of [7] have taken these ideas further defining a benchmark for online database design tools. In contrast to traditional database benchmarking where the final system performance is of interest, a design benchmark needs to make a statement about effectiveness or robustness of a recommended or automatic design.

[7] argue that the benchmark needs to be aware of the performance goals of recommenders, as these can range from total execution time of a workload over an improvement rate between old and newly recommended design to executing a certain number/fraction of all queries within a certain time. The benchmark uses real world (NREF) and synthetic (TPC-H) data for each of which a large set of queries if defined of which a sample set is used during the benchmark. The recommenders are then compared against two additional setups, one using primary key indexes and one using single column indexes for all possibly relevant columns.

Benchmarking online index tuning algorithms is quite different to benchmarking recommenders. The characteristic difference described in [7] is that shifting workloads are required in order to trigger design changes by the online algorithms. As a metric here the total execution time and the optimization objective are used. The latter provides insight on how the system performs during query execution and online adaption. Other than [7] no further reasoning about performance goals of the tuning tools are made.

To understand and assess more fine-granular tuning as in adaptive indexing, [7] proposed benchmarking guidelines suited to these scenarios. An adaptive indexing approach typically requires a startup phase where indexes are build piece by piece and query execution is actually slowed down before it reaches a stable phase of good performance. The authors actually split the time span in between in two phases, the nursing phase, that starts, when execution time drops below the execution time without any index and the growing phase that starts when the first queries arrive that no longer contribute to the adaptive index, i.e. the relevant part of the index is already present and these queries are executed with optimal performance. Not only the stable performance in the end or the overall execution time but also the duration of the different phases are of interest. As adaptive indexing techniques are designed for shifting workloads in particular, the benchmark should account for this. Each shift in a workload will typically require re-tuning the system and imply an extra cycle of the above phases.

VII. CONCLUSIONS AND OUTLOOK

Physical database design is one of the fundamental problems of database design and development which is subject of research since the seventies, e.g. [7]. Over the last two decades significant effort has been made to relieve the burden of physical design tuning from the DBA by developing autonomous techniques adjusting physical schema configurations.
automatically or at least give recommendation for tuning steps. In this paper, we have discussed some of these basic design problems as well as autonomous techniques to tackle them.

Though, numerous approaches and techniques have been proposed and also found their way into commercial database systems, there are still some challenging research problems. First of all, physical design tuning aims at improving performance and scalability of a database – however, in the daily business of database operation peak performance is not the main goal. Instead, autonomous techniques should improve the robustness of the database system, both in terms of predictability and good performance [?].

In [?] we already argued for adaptive index structures which have received attention over the past years, namely database cracking and index merging techniques [?], [?]. In addition to that adaptive materialization strategies [?], [?] have been proposed. However, to our knowledge, these techniques have not yet found their way into the major database systems. Combined with a workload agnostic setup [?], to minimize the penalty when a query does not hit the highly specialized adaptive index, systems should be one step closer to providing robust and high performance.