DB2 Design Advisor: Integrated Automatic Physical Database Design

Daniel C. Zilio[1], Jun Rao[2], Sam Lightstone[1], Guy Lohman[2]
Adam Storm[1], Christian Garcia-Arellano[1], Scott Fadden[3]
[1]IBM Toronto Laboratory
{zilio,light,ajstorm,cmgarcia}@ca.ibm.com

[2]IBM Almaden Research Center
{junrao,lohman}@almaden.ibm.com

[3]IBM Portland
sfadden@us.ibm.com

Abstract

The DB2 Design Advisor in IBM® DB2® Universal Database™ (DB2 UDB) Version 8.2 for Linux®, UNIX® and Windows® is a tool that, for a given workload, automatically recommends physical design features that are any subset of indexes, materialized query tables (also called materialized views), shared-nothing database partitionings, and multidimensional clustering of tables. Our work is the very first industrial-strength tool that covers the design of as many as four different features, a significant advance to existing tools, which support no more than just indexes and materialized views. Building such a tool is challenging, because of not only the large search space introduced by the interactions among features, but also the extensibility needed by the tool to support additional features in the future. We adopt a novel "hybrid" approach in the Design Advisor that allows us to take important interdependencies into account as well as to encapsulate design features as separate components to lower the reengineering cost. The Design Advisor also features a built-in module that automatically reduces the given workload, and therefore provides great scalability for the tool. Our experimental results demonstrate that our tool can quickly provide good physical design recommendations that satisfy users' requirements.

1 Introduction

Technology advances and competition continue to significantly reduce the cost and increase the capacity of database systems, making large, complex database applications commonplace. For example, popular database applications such as SAP [19] typically contain over 30,000 database objects, which include tables and indexes. Concurrently, the cost of skilled database administrators (DBAs) to manage those increasingly complex systems has relentlessly increased. These economic trends have driven the total cost of ownership of systems today to be dominated by the cost of people, not hardware or software. This new reality has sparked recent interest in developing self-managing, or *autonomic* [18], systems that can relegate many of the DBAs' more mundane and time-consuming tasks to automated tools [6,13].

Perhaps the best candidate to date for such automation is physical database design. DBAs have, for years, systematically tuned applications by time-consuming trial and error – methodically creating each index, collecting statistics on it so that the query optimizer knew its properties, recompiling every query that might benefit, and then evaluating whether the index was, in fact, exploited to improve performance for that workload. Each iteration of this painstaking process could take

---

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

minutes or even hours on today's terabyte-sized databases, and there was no way to assure convergence to something considered "optimal". Recent research has produced powerful utilities that use the query optimizer as a "What if?" tool to automate this process for indexes, drastically reducing the manual effort while increasing the number of promising solutions [5,21].

New database features to enhance performance -- such as materialized views, parallelism, and different ways to cluster data – have only compounded this task by providing more options that the beleaguered DBA must consider when designing and tuning an application. To make matters worse, these features interact in complex ways. For example, materialized views, being stored tables, themselves require indexes. And one partitioning option is to replicate smaller tables among all nodes, resembling materialized views.

This paper describes the DB2 Design Advisor in DB2 Version 8.2 [9], the first integrated commercial tool to automatically determine all aspects of physical database design, including not only indexes and materialized views (called materialized query tables in DB2 UDB), but also partitioning and multi-dimensional clustering of tables. As in the individual advisors that this tool integrates, the Design Advisor exploits the DB2 Optimizer to recommend promising candidate solutions and to evaluate alternative solutions. We adopt a novel "hybrid" searching approach that not only takes into account important interdependencies among different features, but also makes the tool easy to extend for supporting new features in the future. To improve the scalability of the tool, we add a built-in workload compression module that automatically reduces the workload size while not sacrificing the quality of the design recommendations.

The rest of the paper is organized as follows. Section 2 explains the terminology specific to DB2 UDB. We summarize related work in Section 3. The overview of our design is given in Section 4 and the implementation details are provided in Section 5. Section 6 describes the workload support, including the workload compression module in the Design Advisor. We present our experimental results in Section 7. The future work is discussed in Section 8 and we conclude in Section 9.

2 Terminology

In this section, we define some of the terminology used in DB2 UDB.

A materialized query table (MQT) is a stored and maintained query result, more commonly known in the literature as a materialized view. MQTs were known as Automatic Summary Tables [22] before Version 8.1 of DB2, when support for join-only views was added.

A multi-dimensional clustering (MDC) table [15] organizes a table in a multi-dimensional cube. Each unique combination of dimension attribute values is associated with one or more physical regions called blocks. A block is a basic unit of clustering and typically contains tens of pages. Indexes are created at the block level for fast access. Since data is clustered in an MDC table, range queries (especially with more than one dimension) can be answered much more efficiently than secondary indexes. The design of an MDC table involves choosing the dimensions as well as the granularity (i.e., the number of distinct values) of each dimension.

The DB2 Enterprise Server Edition has a data partitioning feature (DPF) that enables a shared-nothing parallel architecture [3], where independent processors are interconnected via high-speed networks. Each processor stores a horizontally hash-partitioned (referred to simply as partitioning in the rest of the paper) portion of the database locally on its disk. The design of a partitioning includes selecting a set of columns as the partition key, as well as a set of nodes to which the data will be distributed. A good partitioning design minimizes the movement of data by allowing operations such as joins and aggregations to be done at each node locally and thus are much cheaper to execute. In DB2 UDB, an index does not have its own partitioning, but rather shares the partitioning with the table on which it is defined.

3 Related Work in Database Design

The design of many individual physical features has been well studied in the literature. We cannot possibly list all the related references here. Instead, we simply point out that many works on the selection of indexes, materialized views and partitionings are referenced in [5], [1,23], and [17], respectively.

Work in self-managing databases started as early as 1988 [7], in which the authors proposed to use the optimizer to evaluate the goodness of index structures.

Microsoft Research's AutoAdmin project [1,5] has developed wizards that automatically select indexes and materialized views for a given workload. Their tools also exploit the cost model used by the optimizer to estimate the benefit of suggested indexes and materialized views. More recently, their tools have been extended to support both vertical and horizontal partitions in a research prototype [2].

DB2 UDB has had an Index Advisor [21] since Version 6. A tool [17] that recommends the partitionings in a shared-nothing database system has also been prototyped in DB2 UDB. More details on the architecture of these tools is given in Section 1.1.

Other commercial database vendors such as Informix (bought by IBM in 2001) [11] and Oracle 8i [14] have made similar efforts to build such design tools.

However, most of those existing tools only handle the design of one or two features. The only exception is the tool in [2] where as many as four features can be recommended. [2] uses integrated search among all candidates in order to take into consideration the interactions among features. In comparison, we employ a hybrid approach for both the quality of design recommendations and the extensibility of the tool. The Design Advisor also has a built-in workload compression module for scalability. Finally, the DB2 Design Advisor is the first product to recommend a total of four features so far. In the rest of this paper, we address the challenges that we faced when building this tool.

4 Overview

We first formally defined the problem that we were trying to solve: Given a workload W (a set of SQL statements that may include queries, inserts, updates and deletes), a set of selected features F, and a disk space constraint D, find a set of recommendations for F that reduces the total cost of W the most, while using no more space than D. Our Design Advisor currently supports an F that is any subset of {index, MQT, partitioning, MDC}. Notice that while indexes and MQTs are auxiliary data structures, partitionings and MDCs are modifications to existing structures.

It was clear from the beginning that the Design Advisor would face a huge design search space. Suppose that for a given workload, the number of possible indexes, MQTs, partitionings, and MDCs is NI, NM, NP, and NC, respectively. The combined search space could be as large as $2^{NI+NM+NP+NC}$, because different features could potentially interact with one another. Therefore, we needed a novel approach to solving this problem. In Section 4.1, we discuss two potential approaches to this problem, and compare the pros and cons of each. We then discuss in detail the dependencies among the four features in Section 4.2. Finally, in Section 4.3, we introduce a "hybrid" approach that combines the advantages of the two previous ones.

4.1 Iterative vs. Integrated Approach

A relatively straightforward approach to our problem is to use an iterative approach, which selects each feature one at a time. However, the problem with this approach is that it ignores the interactions among different design features. For example, as explained in [1,23], indexes and MQTs are closely dependent on each other. An MQT, like a regular table, normally needs indexes defined on itself in order to be attractive to a query. The selection of an MQT, on the other hand, can also make an index useless, and vice versa. As another example, in a DPF-enabled database, an MQT can be partitioned. The selection of the partitioning can make an MQT more or less useful to a given query. It is such a dependency among features that significantly complicates our problem. The iterative approach does have an advantage, though. It can treat the selection of each feature as a black box, and does not need to know the implementation details inside a feature, which makes future extension much easier. To support a new feature f, we can just plug in a new component implementing the selection of f. It also gives flexibility to the implementation of each feature selection, since a different searching algorithm can be used for each feature.

An alternative to the iterative approach is an integrated one, in which joint searching is performed directly in the combined search space, and heuristic rules are applied to limit the candidate sets being considered. The advantage of the integrated approach is that it can better handle the interdependencies among different features. For example, by jointly enumerating indexes and MQTs together, it is very likely to identify the optimal index and MQT combination. For this particular reason, The Microsoft Tuning Wizard [1,2] uses an integrated approach to recommend indexes and materialized views.

The main drawback of the integrated approach is its extensibility. While it may be suitable for selecting a couple of features, it will not scale with the addition of new features since the search space grows combinatorially with respect to the number of new search points. Also, to support an additional feature, a large portion of the code needs to be changed to support joint searching with the new feature, which makes the reengineering cost high and can lead to a higher cost of ownership for customers. As a result, neither the iterative nor the integrated approach alone solves our problem well.

4.2 Feature Dependency

We recognize that although interdependencies often exist, the degree of interdependencies among different pairs of features is not always the same. We say that feature A "strongly" depends on feature B, if a change in selection of B often results in a change in that of A. Otherwise, we say A "weakly" depends on B. We argue that weak dependencies are likely to exist because a new physical design feature is normally introduced to help the areas where existing features do not apply or do not perform well. It's unlikely for a system to support two features that duplicate each other on functionalities.

We categorize the degree of dependencies among the four features that we currently support in Table 1, where an S represents strong and a W represents weak. We explain how the dependency of each feature pair is decided by sweeping through the table diagonally from the upper left.

A \ B	Index	MQT	Partitioning	MDC
Index		S	W	W
MQT	S		W	S
Partitioning	W	S		W
MDC	W	S	W	

Table 1. Classification of Dependencies of A on B

As we described earlier in Section 4.1, indexes and MQTs mutually depend on each other a lot, and thus it's clear that their interdependencies should be classified as strong. One of the complexities comes from the fact that

MQTs can have indexes which competes with indexes on base tables.

The interaction between indexes and partitionings is different. First of all, indexes for local predicates are relatively insensitive to how data is partitioned. Partitioning keys are usually determined by joins and aggregations. Next, consider those indexes selected for nested loop joins. It is true that changing which indexes are available can possibly change the join methods in the execution plan, and therefore may affect the selection of partitionings. However, good partitionings are more influenced by intermediate result sizes, which depend only on cardinalities and predicate selectivities (independent of the existence of indexes or not). Conversely, although the selection of a table's partitioning can potentially influence the selection of join methods and consequently the selection of indexes, such influence is not as strong as that of predicate selectivities. Consider the following SQL query on a TPC-H[20] database as an example.

```
Q1.  SELECT L_ORDERKEY, O_ORDERKEY,
                  P_PARTKEY
      FROM LINEITEM, ORDERS, PART
      WHERE L_ORDERKEY = O_ORDERKEY
         AND L_PARTKEY = P_PARTKEY
         AND P_NAME = 'SOME PART'
```

Notice that a good set of indexes for Q1 probably should include I_1=(p_name, p_partkey) on PART, I_2=(l_partkey, l_orderkey) on LINEITEM and I_3 = (o_orderkey) on ORDERS, where I_1 helps the evaluation of the local predicate and I_2 and I_3 can be used in nested loop joins. Such a choice is mostly based on the fact that the local predicate on PART is very selective, not much based on how tables are partitioned. We observe that whether I_2 and I_3 exist or not affects the join methods and thus can affect the partitioning selection. For example, LINEITEM can be chosen to be partitioned on l_orderkey in one case and on l_partkey in another. However, since all intermediate results are small in both cases, the performance difference on two partitioning selections will also be small, if any. Based on the above analysis and the finding in [8] that complex queries tend to use hash joins more often, we classify that indexes and partitionings weakly depend on each other.

A similar argument can be made on the dependencies between MQTs and partitionings. The only difference is that MQTs themselves can have partitionings, which makes partitionings strongly dependent on MQTs, but not vice versa. This is an interesting example where the degree of dependencies is not necessarily symmetric.

We classify the dependencies between indexes and MDCs as weak, which seems somewhat controversial. Indeed, MDC enables and requires a special kind of index. However, MDC was actually developed to serve a different class of queries than traditional indexes serve. For example, a secondary index is typically useful when the number of matching records is relatively small. On the other hand, an MDC organization is especially beneficial for OLAP types of queries, taking slices of the multi-dimensional cubes, for which there are typically many matches, since the matching records have all been pre-clustered together. There is, in fact, a strong interaction (similarity) between a one-dimensional MDC and a conventional clustered index. We address this issue in Section 5.3.

Finally, we do find that MDCs are very similar to indexes in their relationship to MQTs and partitionings, which means that MDCs and MQTs are strongly coupled while MDCs and partitionings are weakly coupled. Similar to partitioning, an MQT can be further clustered through an MDC organization.

4.3 The Hybrid Approach

We observe that mutual strong dependencies are difficult to break and thus are better handled using an integrated approach. We decouple other dependencies (unilateral strong and weak) and apply the iterative approach. To minimize the impact of doing so, we carefully choose the ordering within an iteration and add special cases within a component whenever necessary. We are now ready to outline our "hybrid" approach.

In general, for a pair of features A and B, if A and B are mutually strongly dependent on each other, we will create a component that jointly searches both A and B. If only B strongly depends on A, we will iteratively search A and B, but make sure that A is searched before B so that B is properly influenced by A. Finally, if A and B are weakly coupled, we will again separate them into different components, but can iterate through them in any order. Furthermore, we try not to lose the weak dependencies completely; we allow each component to optionally implement a quick and simple search of a feature in another component to account for the weak relationship. Our hybrid approach enables us to break the implementation of different features into smaller components while capturing the most important interdependencies among them. Such an approach makes it possible for us to build a tool that can handle the design of all four features and be able to extend in the future. In the next section, we describe in detail how we developed the hybrid approach in the Design Advisor.

5 The Implementation of the Hybrid Approach

We developed the hybrid approach by extending the infrastructure of the existing Index Advisor in DB2 UDB. In Section 1.1, we revisit the architecture of the Index Advisor. We then describe the necessary extensions in Section 5.2. In Section 5.3, we describe the main algorithm used in the Design Advisor. Finally, we discuss some of the issues concerning MQT selection in Section 5.4 and unused structures in Section 5.5.

5.1 Index Advisor Revisited

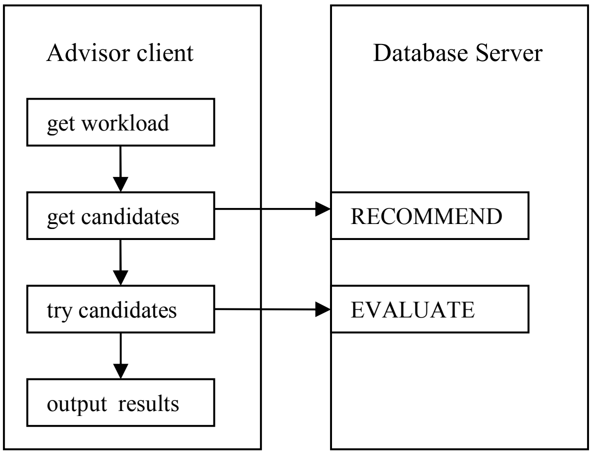

Figure 1. Architecture of an Index Advisor

The DB2 Index Advisor [21] uses an architecture as depicted in Figure 1. The database server is augmented with two special "EXPLAIN" modes—RECOMMEND INDEXES and EVALUATE INDEXES. Under a special explain mode, a given statement is compiled but not executed. In the RECOMMEND mode, the optimizer is extended such that it will generate promising "virtual" indexes for a given statement on the fly. Virtual indexes are then considered during optimization as if they are physically present. Those indexes that are part of the final best execution plan are treated as the best index candidates for the given SQL statement. They are collected by the optimizer and are written to a special advise_index table. In the EVALUATE mode, the optimizer obtains from the advise_index table those candidates marked as "in_use" and generates the corresponding virtual indexes before optimization starts. In other words, the EVALUATE mode causes the virtual indexes in the advise_index table to act as a temporary extension to the DB catalog information. The optimization then continues to compute the best execution plan assuming those virtual indexes are physically present.

The client side of the Index Advisor first collects a workload. It then compiles each statement in the RECOMMEND mode and collects the best index candidates for each. An enumeration algorithm then combines those candidates in various ways, and for each combination compiles the workload in the EVALUATE mode to get a corresponding cost estimation for each statement in the workload. Finally, the combination with the lowest total cost is returned as the best solution for the workload. One of the advantages of this architecture is that it uses the RECOMMEND mode to suggest index candidates. Since the RECOMMEND mode is integrated inside the database engine, it is able to find candidates quicker and more accurately.

5.2 Explain Mode Extension

We extend the methodology used for the Index Advisor to the three other features as well. For each additional feature we intend to support, we add two special EXPLAIN modes, one for recommending the feature and the other for evaluating the feature. We also add a corresponding "advise" table to store the candidates. Most importantly, we extend the EXPLAIN register from a single value to a bit set such that multiple EXPLAIN modes can be set at the same time. Such an extension is very powerful, since it provides the capability of conducting both joint searching and iterative searching. For example, by setting RECOMMEND INDEXES and RECOMMEND MDC together, the optimizer is able to suggest RID index and MDC dimension candidates together. In another example, by setting EVALUATE INDEXES and RECOMMEND PARTITIONINGS mode together, the optimizer can try to generate the best partitionings for a statement while assuming the existence of indexes suggested from a previous iteration. Note that although the infrastructure appears to be the same for each feature, it does not prevent each feature (if in a separate component) from using a different searching method in the try candidates phase. Finally, the bit set representation can easily support additional features in the future.

5.3 The Hybrid Algorithm

We divide the Design Advisor into three components, IM, P, and C, where IM is responsible for index and MQT recommendation, and P and C are responsible for partitionings and MDCs, respectively. Each component F recommends candidates within its own disk constraint D_F. The components are iterated in the order of IM, P, and C. This closely reflects the degree of dependencies as described in Table 1, except for the relationship between MQTs and MDCs. We chose not to integrate the selection of MDCs with indexes and MQTs for several practical reasons. First, we want to limit the search space within a single component. Second, we feel that MQTs are more closely coupled with indexes than with MDCs. Finally, MDCs do not compete for as much space as both indexes do since they use block indexes instead of RID indexes, and we wanted to integrate only the resource-intensive features together.

The implementation of each of the three components has been described in detail in [12,17,23] and is not the focus of this paper. We just want to point out that the searching algorithm used for each component is customized. For example, a knapsack algorithm followed by a random-swapping phase was used for the IM component [23], and a rank-based search was used for the P component [17]. While components are relatively independent of each other, they are aware of the presence of others whenever needed. For example, we support using sampling in each component to obtain more accurate statistics. When this occurs, components share as much sampling as possible. Each component can be disabled. This is useful when a feature is not available on a particular platform (e.g., partitioning is only available when DPF is enabled) or the user does not select all the features.

```
get the workload information, database and system characteristics and the disk constraint D
get initial cost of the workload
while (stop criteria met) {
  for each enabled component F of IM,P,C {
    invoke F with $D_F$
    (solutions for F now stored in advise_F)
    if (current_cost < best_cost)
      best_cost =current_cost
    else
      unmark candidates in advise_F
    turn on EVALUATE F in explain register
  }
}
output solutions
```

Figure 2. Main Algorithm

Our main hybrid algorithm used in the Design Advisor is presented in Figure 2. As usual, the advisor first obtains the workload, information on the database (e.g., DB name, user ID, and password) and a disk constraint D. If D is not provided, the Advisor automatically recommends a D based on such information as the amount of available space and used space in a database. Each statement in the workload is then compiled to get an optimizer-estimated initial workload cost. After that, the algorithm starts to iterate through each enabled component. The total disk constraint is divided among all components. More space is given to indexes and MQTs than partitionings and MDCs because the latter two tend to use less space. (The only type of partitioning that takes extra space beyond the base data is replication.) After invoking a component with an allocated constraint D_F, the suggested candidates for F are recorded in the corresponding advise table (with the "in_use" field marked). Note that if a component does not use all the allocated space, unused space is passed to the next component and can be consumed by other features. The current workload cost (with the solutions for F) is used to compare against the best workload cost. If the current cost is smaller, the new solution for F is accepted and the best cost is updated. Otherwise, the new solution is discarded by unmarking the "in_use" field in the advise table. We then turn on the EVALUATE F mode in the special explain register so that the solutions for F become visible by subsequent components. As a special case, we let the C component handle all clustered indexes. If an MDC solution has only one dimension, the C component will decide whether to use an MDC organization or simply create a conventional clustered index.

The main algorithm is capable of iterating through each component F more than once, and therefore solutions for F can change after the design of other features is exposed. Such iterations continue until some stopping criteria are met, which can either be that solutions do not improve the workload any more, or a user-specified time limit is reached. In a prototype we implemented that repeats the iteration between component IM and P, we found that the second invocation of IM (with the solutions for partitioning) does not change the previous index and MQT recommendations significantly. This actually verifies our assumption that indexes and MQTs only weakly depend on partitionings. Therefore, we currently only iterate through each component exactly once.

We add a special support in the IM component to address MQTs' weak dependency on partitionings. Observe that although the influence of partitionings on MQTs is relatively weak, a terrible partitioning key for an MQT can still reduce its potential benefit, especially when maintenance cost needs to be considered. Hence, we extend IM by adding a module that quickly selects a reasonable initial partitioning for each MQT to prevent a good MQT from being pruned. Every incrementally maintainable MQT in DB2 UDB has an implied unique key [22]. For instance, the implied key is the grouping columns for an aggregate MQT. For a join MQT, the implied key is the concatenation of the key on each joined table. We choose an arbitrary column from the implied key to serve the initial partitioning key of the MQT. During incremental maintenance, such a partitioning key (a subset of the implied key) allows the join between the MQT and the "delta" to be performed locally at each node, and thus reduces the maintenance cost. Subsequently, the partitioning of the MQTs will be further tuned by the P component.

5.4 MQT Selection

The goal of the Design Advisor is to allow users to select any subset of the supported features. If a feature is not requested by the user, normally, we can simply bypass the corresponding component during the iteration. However, when the user only asks for MQTs, the semantic is a little bit tricky. If we faithfully follow the request and suggest only MQTs and nothing else, such MQTs may not be usable because they typically need some indexes on them. On the other hand, since the users probably have some confidence in existing indexes, we probably should not voluntarily perform a full index search. As a solution, we decide that if MQT is the only feature requested by the user, the Design Advisor will automatically recommend indexes and partitionings (if DPF is enabled) on suggested

MQTs. We choose not to cluster suggested MQTs using MDC since it is a more advanced feature.

In order to support this, we introduce another EXPLAIN mode VIRTUAL_MQT. When this mode is enabled in the EXPLAIN register bit set, indexes and partitionings are only recommended and evaluated for newly recommended (virtual) MQTs.

5.5 Unused Structures

While the Design Advisor recommends new design structures, would it make sense for it to remove structures that are not used at all? House-cleaning often has lower priority than adding new designs, so it's common to have indexes and MQTs that are out-of-date. However, the danger is that the Design Advisor may not see the complete workload. Although more built-in workload supports have been added to the Design Advisor (Section 6.1 describes the details), infrequent queries are still hard to collect. Therefore, we may delete an index that seems useless but is very important for a CEO query that runs only every quarter. Because of this, the Design Advisor does not recommend any deletion of existing structures. Instead, it reports a list of existing indexes and MQTs that are useful to the given workload. The set of unused indexes and MQTs can be inferred from this list.

6 Workload Support

Since the Design Advisor is a workload-driven tool, we pay a lot of attention to workload-related issues. In Section 6.1, we describe additional ways in the Design Advisor for users to conveniently obtain a workload. In Section 6.2, we introduce the built-in workload compression method for scalability.

6.1 Obtaining a Workload

The DB2 Index Advisor accepts a workload from the command line (a single statement), a file, or an advise_workload table. In the Design Advisor, we add two additional workload sources, one from the dynamic statement cache and the other from the Query Patroller.

The dynamic statement cache stores the plans for all dynamic SQL queries submitted to a database engine to avoid recompilation. As a side effect, each dynamic SQL statement itself is cached, together with the frequency of execution. Therefore, the statement cache serves as a good source for a typical workload. With this new option, users can run their favourite applications for a while and then invoke the Design Advisor, which will then collect the SQL statements and associated frequencies from the cache automatically.

Query Patroller [16] is a powerful query management tool included in DB2 Data Warehouse Enterprise Edition. It provides the capability of classifying queries into classes, prioritizing queries, and tracking runaway queries. We add an option in the Design Advisor so that we can fetch all statements passed through Query Patroller.

6.2 Built-in Workload Compression

A key factor that affects the scalability of the Design Advisor is the size of the workload. Since each statement in the workload needs to be compiled by the optimizer (most likely more than once) in order to obtain the estimated cost, the larger the workload, the longer it takes the advisor to run. As a matter of fact, the time to run design tools such as Microsoft Tuning Wizard and DB2 Index Advisor typically grows exponentially with a linear increase of the workload size (verified through experiments). Thus, workload compression is imperative for the scalability of these design tools.

A simple workload compression technique is to merge statements that are exactly the same, but with different parameter bindings. In fact, for workloads that are obtained from the dynamic package cache, such compression has already been done. One study [4] proposes a more sophisticated workload compression technique that employs mining-like methods to summarize the workload. The authors demonstrate that using the reduced workload, both the Microsoft Tuning Wizard and DB2 Index Advisor can provide design recommendations very close to those based on the full workload. However, to get the reduced workload, the technique requires relatively intensive computation such as calculating the distance between pairs of statements in the original workload.

We see a benefit in adding workload compression as a built-in module for the Design Advisor. The Design Advisor will invoke the workload compression module if it feels that the workload is too large and the analysis cannot finish in a reasonable amount of time. One important requirement for the compression module is efficiency. We want to spend a relatively small fraction of the total amount of time on compression the workload. Therefore, we take an approach that only keeps the top K most expensive queries, whose total cost is no more than X% of the original workload cost. The Design Advisor already compiles each statement in the workload to obtain an estimated cost (line 2 in Figure 2). We then sort the statements in descending cost order and keep selecting statements from the top into a reduced workload until the cost of the reduced workload is less than or equal to X% of the original workload. Our approach, although simple, is quite effective in reducing the workload size, especially when the distribution of statement cost is skewed. The reduced workload includes the most time-consuming statements, which typically need tuning.

We can control the compression ratio by scaling the percentage X. Instead of burdening the users to come up with an appropriate value for X, we expose only three compression levels: low, medium and high, with X set to 60, 25, and 5, respectively. By default, medium compression level will be used. Once compression is done, the hybrid algorithm simply works on the reduced workload and gives recommendations accordingly. It makes one final pass over the original workload at the end to obtain the cost for the whole workload with the recommendations based on the reduced workload. Finally, we allow the user to turn off workload compression completely for more accurate design tuning.

7 Experimental Results

In this section, we select two sets of experiments to present. All experiments were conducted on a regular build for DB2 UDB Version 8.2. In Section 7.1, we test the Design Advisor by selecting all four features. In Section 7.2, we focus on the selection of indexes and MQTs only, and demonstrate the benefit of our built-in workload compression module. We summarize our experimental results in Section 7.3.

7.1 TPCH Results

The first experiment was to demonstrate how well the Design Advisor recommends all design features. This was done using a 1 GB TPCH [20] database stored on an 8 CPU AIX® 5.2 system with 4 logical partitions. The workload contains all the 22 TPCH queries.

We started with a baseline design that stored the tables across all 4 partitions and used the primary key as the partition key for all tables except for LINEITEM, which was partitioned on L_PARTKEY, part of the primary key. The LINEITEM partitioning was chosen based on the fact that L_PARTKEY is used in quite a few of the 22 queries. The rest of the baseline physical DB design is derived from a TPCH benchmark.

Design Feature	Number Recommended
Indexes	20
MDC dimensions	6
Partitioning Changes	4
Materialized Views	2

Table 2. Design Advisor Recommendations for a TPCH 1GB Database

The Design Advisor was able to finish the design of all features in about 10 minutes. Table 2 shows how many recommendations the advisor made for each design feature. For example, we show below one of the MQTs recommended (MQT2) that contains the subquery in Q18 using LINEITEM. In this MQT, the partitioning key was also properly selected on C1, because the MQT sometimes needs to be further joined with the ORDERS table. An index IDX3 was also recommended on MQT2. Notice that the index key is ordered in (C0,C1). This is because the subquery result in Q18 was subsequently filtered though a range predicate on L_QUANTITY.

```
CREATE SUMMARY TABLE MQT2 AS (
    SELECT SUM(L_QUANTITY) AS C0,
              L_ORDERKEY AS C1
    FROM TPCD.LINEITEM
    GROUP BY L_ORDERKEY)
DATA INITIALLY DEFERRED
REFRESH DEFERRED
PARTITIONING KEY (C1)
IN TPCDLDAT

CREATE INDEX IDX3 ON MQT2
(C0 DESC,
 C1 DESC)
ALLOW REVERSE SCANS
```

An MDC recommendation from this experiment is also given below. The recommendation was for the PARTSUPP table to be MDC clustered (as shown by the ORGANIZE BY clause) based on a newly generated column. This column groups the values of PS_PARTKEY such that each group falls into a clustered block of the MDC. In this particular case, a single-dimensional MDC is better than a conventional clustered index because the generated column condenses the value domain.

```
CREATE TABLE TPCD.PARTSUPP (
    PS_PARTKEY INTEGER NOT NULL ,
    PS_SUPPKEY INTEGER NOT NULL ,
    PS_AVAILQTY INTEGER NOT NULL ,
    PS_SUPPLYCOST DOUBLE NOT NULL ,
    PS_COMMENT VARCHAR(199) NOT NULL,
    MDC040303204738000 GENERATED
        ALWAYS AS (
           INT((PS_PARTKEY-11)/(792))) )
PARTITIONING KEY (PS_PARTKEY)
IN TPCDTDAT
ORGANIZE BY (MDC040303204738000 )
```

Finally, we note that besides the partitionings recommended for MQTs, the partitioning key of LINEITEM is also changed from L_PARTKEY to L_ORDERKEY. The latter is useful for fewer, but much more expensive, queries.

The DB2 Design Advisor makes its recommendations based on estimated response times for workloads using the cost model in the optimizer. In this experiment, we obtained an estimated response time improvement over the baseline (without recommendations from the Design Advisor) of 88.01%. We implemented all the recommendations made by the Design Advisor and measured the actual cost of the workload. Figure 3 shows the real performance improvement as 84.54% (the baseline is normalized to 100%), which is very close to the optimizer's estimation.

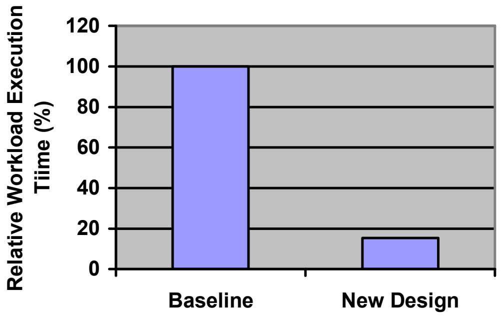

Figure 3. 1 GB TPCH Real Workload Performance Improvement

7.2 MOLAP Results

In the next set of experiments, a DB was set up as a classic MOLAP schema. Both the data and the workload are synthetic, but represent similar characteristics seen in various customer MOLAP schemas. The DB has a fact table with 8 measures, and there are 16 hierarchical dimensions. Table 3 shows the number of levels in each hierarchy and the cardinality of each dimension.

Experiments were run on a Windows® 2000 Server SP4 with four 400 MHz CPUs. We set up the database to have four logical partitions. Note that these experiments only selected indexes and materialized views.

Dimension	Number of Levels in Hierarchy	Cardinality
1	2	4
2	2	3
3	2	2
4	4	52
5	3	3000
6	2	7
7	2	4
8	3	300
9	2	331
10	2	2
11	4	189
12	2	11
13	3	3000
14	4	372
15	2	2
16	2	2

Table 3. MOLAP Schema Characteristics

We demonstrate the usefulness of our workload compression with respect to reducing the Design Advisor execution time. Medium compression was compared to no compression with workloads of varying numbers of queries. All queries include range predicates, join one or two dimensional tables with the fact table, and aggregate at various levels in the hierarchy. Figure 4 shows the results. Note that we only show the 80 to 150 query cases for no compression because that is where the interesting differences occur. The results indicate that as the number of queries increases, the advisor execution time also increases with and without compression. However, under the medium compression, the Design Advisor runs twice as fast when there are 80 and 100 queries in the workload, and an order of magnitude faster when there are 150 queries (note the logarithmic scale on the y-axis). Although not shown here, a higher level of compression provides a more significant reduction of the execution time of the Advisor. There is a slight increase at the 80 query workload mainly because its random-swapping phase was longer than that in the 100 query case. In the 100 query case, the advisor found a very good solution early and thus finished the algorithm earlier. We also compared these results to our competitors' and found that the Design Advisor achieved a more significant reduction in the execution times.

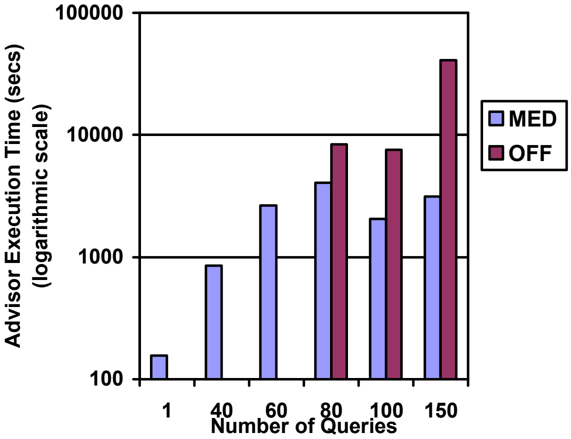

Figure 4. Medium (MED) and no (OFF) workload compression advisor execution time comparison using the MOLAP schema workload

There is a trade-off between workload compression and the quality of the design recommendations. The higher the compression level, the faster the advisor runs, but potentially the lower the quality of the recommendations. In Figure 5, we present the progress made by the Design Advisor with and without workload compression. Each point in the figure represents the workload improvement achieved by the Advisor after running for a certain amount of time. In the end, the advisor ended up with a 77% estimated improvement with the medium compression, and an improvement of 93% with no compression. However, with medium compression, the advisor finished about 4,300 seconds sooner than with no compression. As a result, medium compression provides a good compromise between execution time and design quality. Figure 5 also demonstrates that a large portion of the performance improvement is achieved in a relatively short amount of time by the Design Advisor (both with compression and without). Subsequent searching only further improves the performance marginally. This is very useful for designing the stopping criteria in our main algorithm. It becomes reasonable to stop the Advisor when no improvement has been made after the advisor has executed for a certain number of iterations.

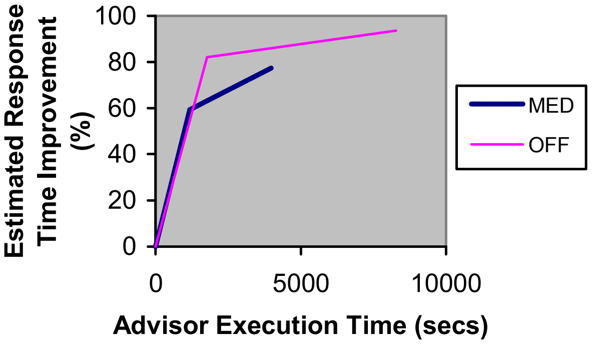

Figure 5. Workload improvement vs. advisor execution time for medium (MED) and no (OFF) workload compression for the 80 query MOLAP schema workload

7.3 Summary

To summarize, our experiments demonstrate that the Design Advisor is capable of recommending a design that includes all the four features, significantly improves the performance of the workload over a benchmark baseline, and completes in a reasonable amount of time. We also validate the effectiveness of our workload compression technique that allows the Design Advisor to scale with the increase of workload size. Our analysis shows that a medium compression level reduces the Advisor execution time considerably without compromising the quality of the recommendations.

8 Future Work

While we categorize the interaction between MQTs and partitionings as weak in Table 1, a novel usage of MQTs can change that. For example, suppose that two queries Q1 and Q2 prefer a table T to be partitioned using P1 and P2 respectively. Normally, we can only choose one of the partitionings for table T. However, it's possible to define an MQT that duplicates T and also carries a different partitioning than T. That way, we can use T to serve one of the queries, say Q1, and use the MQT to serve Q2. When used this way, MQTs become strongly coupled with partitionings. We'd like to investigate an efficient way to support this special case in the future, although we have addressed this issue partially through the replicated partitioning recommendations made by the P component.

The Design Advisor is currently focused on physical database design. In the future, we'd like to investigate the possibility of extending it to support logical designs as well. Logical design is currently done by tools such as Rational® Rose® and XDE™ [10], when database schemas are derived from modelling specifications such as UML. How to integrate the Design Advisor with such tools is an interesting study for the future.

9 Conclusion

The DB2 Design Advisor is the first comprehensive physical database design tool to recommend indexes, materialized views, partitioning, and clustering for multiple dimensions in an integrated and scalable fashion. We have described a framework that permits any combination of recommending some features while holding others fixed to a given solution, as well as a hybrid algorithm that efficiently searches through the enormous space of possible solutions while taking into consideration the interactions of related features. The Design Advisor also has built-in workload compression for reducing the execution time of the Advisor without sacrificing quality in the solution. Initial experimental results verify that solutions selected by the Design Advisor improve by almost 100% the performance of workloads of hundreds of queries after running for under three hours, less time than it would take a human DBA to evaluate a handful of possible solutions, and represents a major advance in automating perhaps the most complex and time-consuming task that DBAs now perform.

References

1 Sanjay Agrawal, Surajit Chaudhuri and Vivek R. Narasayya, Automated Selection of Materialized Views and Indexes in SQL Databases, Proceedings of 26th International Conference on Very Large Data Bases, 2000: 496-505.

2 Sanjay Agrawal, Vivek R. Narasayya, Beverly Yang, Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. SIGMOD Conference 2004.

3 Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, Walter G. Wilson: An Overview of DB2 Parallel Edition. SIGMOD Conference 1995: 460-462

4 Surajit Chaudhuri, Ashish Gupta, Vivek R. Narasayya: Compressing SQL workloads. SIGMOD Conference 2002: 488-499

5 Surajit Chaudhuri and Vivek R. Narasayya, Microsoft Index Tuning Wizard for SQL Server 7.0, Proceedings ACM SIGMOD International Conference on Management of Data, 1998: 553-554.
6 Surajit Chaudhuri and Vivek R. Narasayya, AutoAdmin "What-If" Index Analysis Utility. Proceedings of ACM SIGMOD, Seattle, 1998.
7 S. Finkelstein and M. Schikolnick and P. Tiberio, Physical Database Design for Relational Databases, ACM Transactions of Database Systems, 13(1): 91-128, 1988.
8 Goetz Graefe: The Value of Merge-Join and Hash-Join in SQL Server. VLDB 1999: 250-253
9 http://www.ibm.com/software/ db2/
10 http://www.ibm.com/software/ rational/
11 http://www.ibm.com/software/data/informix/ redbrick/
12 Sam Lightstone and Bishwaranjan Bhattacharjee, Automated design of Multi-dimensional Clustering tables for relational databases, VLDB 2004.
13 Guy M. Lohman, Sam Lightstone: SMART: Making DB2 (More) Autonomic. VLDB 2002: 877-879
14 http://www.oracle.com/
15 Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Timothy Malkemus, Leslie Cranston, Matthew Huras: Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. SIGMOD Conference 2003: 637-641
16 Query Patroller, http://www.ibm.com/software/data/ db2/querypatroller/.
17 Jun Rao, Chun Zhang, Nimrod Megiddo, Guy M. Lohman: Automating physical database design in a parallel database. SIGMOD Conference 2002: 558-569.
18 http://researchweb.watson.ibm.com/autonomic/ manifesto/
19 http://www.sap.com/
20 TPC-H benchmark, http://www.tpc.org/
21 Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy Lohman and Alan Skelley, DB2 Advisor: An optimizer smart enough to recommend its own indexes, Proceedings of the ICDE Conference, 2000: 101-110.
22 Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, Monica Urata: Answering Complex SQL Queries Using Automatic Summary Tables. SIGMOD Conference 2000: 105-116
23 Daniel C. Zilio, et al, Recommending Materialized Views and Indexes with IBM's DB2 Design Advisor, International Conference on Autonomic Computing 2004.

Trademarks

AIX, DB2, DB2 Universal Database, IBM, and Informix, Rational, Rational Rose, and XDE are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Windows is a registered trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.