Database Tuning Advisor for Microsoft SQL Server 2005

Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, Manoj Syamala

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052.
USA
{sagrawal,surajitc,lubork,arunma,viveknar,manojsy}@microsoft.com

Abstract

The Database Tuning Advisor (DTA) that is part of Microsoft SQL Server 2005 is an automated physical database design tool that significantly advances the state-of-the-art in several ways. First, DTA is capable to providing an *integrated* physical design recommendation for horizontal partitioning, indexes, and materialized views. Second, unlike today's physical design tools that focus solely on performance, DTA also supports the capability for a database administrator (DBA) to specify manageability requirements while optimizing for performance. Third, DTA is able to scale to large databases and workloads using several novel techniques including: (a) workload compression (b) reduced statistics creation and (c) exploiting test server to reduce load on production server. Finally, DTA greatly enhances scriptability and customization through the use of a public XML schema for input and output. This paper provides an overview of DTA's novel functionality, the rationale for its architecture, and demonstrates DTA's quality and scalability on large customer workloads.

1. Introduction

The performance of an enterprise database system can depend crucially on its physical database design. Automated tools for physical database design can help reduce the total cost of ownership (TCO) of databases by reducing the DBA's burden in determining the appropriate physical design. The past few years have seen an emergence of such automated tools. Indeed, today's major commercial database systems include as part of their product, automated tools such as [2,3,4,6,15,18,21] that analyze a representative *workload* consisting of queries and updates that run against the database, and recommend appropriate changes to physical design for the workload.

1.1 Requirements of a Physical Design Tool

While these state-of-the-art tools represent an important step in the direction of reducing TCO, there are a number of important requirements, described below, in which currently available tools are still lacking that can make it difficult to use in an enterprise environment.

Integrated selection of physical design features: Today's database engines offer a variety of physical design features such as indexes, materialized views and horizontal partitioning, each of which can have a significant impact on the performance of the workload. A physical design tool should ideally provide an integrated "console" where DBAs can tune all physical design features supported by the server. In supporting such a console, it may appear natural (for scalability reasons) to employ a staged solution to the physical design problem, -- for example, first choose partitioning of tables only, then pick indexes, then pick materialized views etc. However, as shown in [3,4] (and discussed in Section 3), due to the strong interaction among these features, such staging can potentially lead to an inferior physical design. Thus, a tool that is capable of making an integrated physical design recommendation that takes into account the interactions among all these features is important since otherwise: (a) Ad-hoc choices need to be made on how to stage physical design selection; and (b) It is difficult to quantify how much performance is compromised for a given database and workload as a result of staging. There are tools that integrate tuning of certain physical design features, e.g., [3,21], but to the best of our knowledge, no tool until now offers a fully integrated approach to tuning indexes, materialized views and horizontal partitioning together.

Incorporating manageability aspects into physical design: The focus of today's physical design tools is on improving performance. However, manageability of physical design is often a key requirement. For example,

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

DBAs often use horizontal range partitioning to ensure easy backup/restore, to add new data or remove old data. In these scenarios, having a table and all of its indexes *aligned*, i.e., partitioned identically makes these tasks easier. On the other hand, the manner in which a table (or an index) is partitioned can have significant impact on the performance of queries against the table (using the index). Therefore, it becomes important for an automated physical design tool to allow DBAs the ability to specify alignment requirements while optimizing for performance.

Scaling to large databases and workloads: Enterprise databases can be large and a typical representative workload can also be large (e.g., number of queries/updates that execute on a server in one day can easily run into hundreds of thousands or more). Thus, to be effective in an enterprise environment, these tools need to be able to scale well, while maintaining good recommendation quality.

Ability to tune a production database with very little overhead: Consider a case where a DBA needs to tune a large workload. Tuning such a workload can incur substantial load on the production server. Therefore, there is a need to tune these databases by imposing very little overhead on production server. Sometimes test servers exist, but it is often infeasible or undesirable to copy the entire database to the test server for tuning. Moreover the test server may have different hardware characteristics, and thus the recommendations of the tool on the test server may not be appropriate for the production server.

Scriptability and customization: As physical design tools become more feature rich, and get increasingly used in enterprises, the ability to script these tools for DBAs and build value added tools on top by developers becomes more important. Moreover different degrees of customization are necessary for different scenarios. At one extreme, the tool should be able to make all physical recommendations on its own. At the other extreme, the DBA should be able to propose a physical design that is simply evaluated the tool. In between, the DBA should be able to specify a physical design partially (e.g., clustering or partitioning of a given table) and the tool should complete the rest of the tuning. Such support is inadequate in today's physical design tools.

1.2 Advancements in DatabaseTuning Advisor

In this paper, we describe **Database Tuning Advisor** (**DTA**), an automated physical design tool that is part of Microsoft SQL Server 2005. DTA significantly advances functionality, manageability, scalability, and scriptability relative to the state-of-the-art physical design tools. DTA is the next generation physical design tool that builds upon the Index Tuning Wizard in Microsoft SQL Server 2000. First, DTA can provide *integrated* recommendations for indexes, materialized views as well as single-node horizontal partitioning for a given workload (Section 3). DTA provides the user the ability to specify the requirement that the physical database design should be *aligned*, i.e., a table and its indexes should be partitioned identically (Section 4). DTA scales to large databases and workloads using several novel techniques including: (a) workload compression that helps scaling to large workloads; (b) reduced statistics creation that helps to reduce time for creating statistics for large databases; and (c) exploiting test server to reduce tuning load on production server. These techniques for improving scalability are discussed in Section 5. The input and output to DTA conforms to a public XML schema for physical database design which makes scripting and customization easy, and enables other tools to build value-added functionality on top of DTA. Such usability enhancements in DTA are discussed in Section 6. DTA exposes a novel feature called *user specified configuration* that allows DBAs to specify the desired physical design partially (without actual materialization), while optimizing for performance. This allows for greater customizability. These and other important usability enhancements in DTA are discussed in Section 6. In Section 7 we present results of extensive experiments on several customer workloads and the TPC-H 10GB benchmark workload that evaluates: (a) the quality of DTA's recommendation compared to a hand-tuned physical design and (b) its scalability to large databases and workloads. We summarize related work in Section 8. We begin with an overview of DTA functionality and its internal architecture in Section 2.

2. Overview of Database Tuning Advisor

2.1 Functionality

Figure 1 shows an overview of the Database Tuning Advisor (DTA) for Microsoft SQL Server 2005. DTA is a client physical database design tuning tool. It can be run either from a graphical user interface or using a command-line executable.

Input: DTA takes the following inputs:

- A set of databases on a server. Many applications use more than one database, and therefore, ability to tune multiple databases simultaneously is important.
- A workload to tune. A workload is a set of SQL statements that execute against the database server. A workload can be obtained by using SQL Server Profiler, a tool for logging events that execute on a server. Alternatively, a workload can be a SQL file containing an organization or industry benchmark, for example.

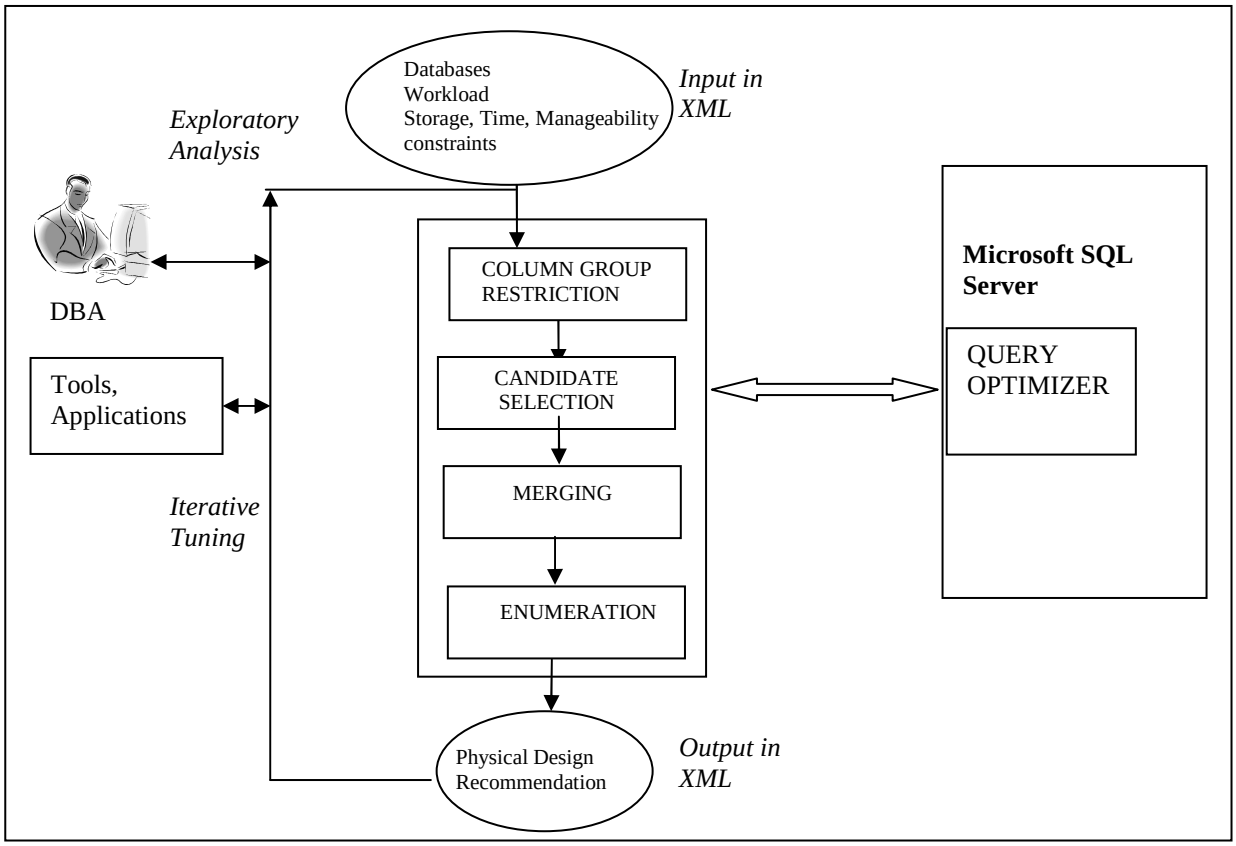

Figure 1. Overview of Database Tuning Advisor

- Feature set to tune. Although DTA is capable of tuning indexes, materialized views, and partitioning together, DBAs may sometimes need to limit tuning to subsets of these features. For example, a DBA of an OLTP system may decide *a priori* to not include any materialized views.
- Optional *alignment* constraint that a table and all its indexes must be partitioned identically (although different tables may be partitioned differently).
- User can specify a partial configuration (e.g., clustered index on a table, partitioning of a table or materialized view), and constrain DTA's recommendation to honor this specification.
- An optional storage constraint: DTA allows specifying a bound on the storage that the physical design it recommends can consume.
- An optional time constraint: an upper bound on the time that DTA is allowed to run. Although DTA's ability to perform tuning within a time bound is an interesting technical problem in itself, in this paper we do not focus on the problem of time-bound tuning or on DTA's specific solution.

Output: The output of DTA is a physical design recommendation (which we refer to as a *configuration*) consisting of indexes, materialized views, and a recommendation on horizontal range partitioning of tables, indexes, and materialized views.

2.2 Architecture

DTA extends the architecture developed in the context of previous work for index and materialized view selection [3] (used by Index Tuning Wizard in Microsoft SQL Server 2000). DTA's architecture is shown in Figure 1. Several properties of this architecture should be noted: (a) It is not specific to a particular physical design feature, and in principle, requires little changes as new physical design features are added; (b) It provides an integrated physical design recommendation; (c) It is in-sync with the query optimizer component of the database server. Below we briefly summarize the salient aspects of this architecture, and highlight extensions made compared to [3], where appropriate.

DTA's Cost Model: The basis of DTA's recommendations is the "what-if" analysis interfaces of Microsoft SQL Server described in [9], which have been extended to also support simulation of horizontally partitioned tables, indexes and materialized views [4] (in addition to non-partitioned indexes and materialized views). Microsoft SQL Server 2005 supports single-column range partitioning. Using these interfaces, for a given query Q and a configuration C, DTA can obtain the

optimizer-estimated cost of Q as if configuration C were materialized in the database. Among all the configurations that DTA explores for the given workload, it recommends the one with the lowest optimizer-estimated cost for the workload. There are several advantages of being "in-sync" with the query optimizer: (a) The configuration DTA recommends, if implemented, will in fact be used by the query optimizer; (b) As query optimizer's cost model evolves over time, DTA is able to automatically benefit from improvements to it; (c) DTA takes into account all aspects of performance that the query optimizer can model including the impact of multiple processors, amount of memory on the server, and so on. It is important to note, however, that query optimizers typically do not model all the aspects of query execution (e.g. impact of indexes on locking behavior, impact of data layout etc.). Thus, DTA's estimated improvement may be different from the actual improvement in execution time.

Column-Group Restriction: The space of all physical design structures (indexes, materialized views, partitioning) that need to be considered for a given workload can be very large. The explosion in the number of physical design structures that must be considered is a result of the large number of column-groups (i.e., sets of columns) that are, in principle, relevant for the workload. Column-group restriction a pre-processing step that eliminates from further consideration a large number of column-groups that can at best have only a marginal impact on the quality of the final recommendation (in particular, column-groups that occur only in a small fraction of the workload by cost). The output of this step is a set of "interesting" column-groups for the workload. Indexes and partitioning considered by DTA is limited only to interesting column groups, thereby significantly improving scalability with little impact on quality [4]. For scalability, we construct the interesting column-groups for a workload in a bottom-up manner leveraging the idea of frequent itemsets [5]. This module is an extension of the previous architecture presented in [3].

Candidate Selection: The Candidate Selection step selects for each query in the workload (in particular, one query at a time), a set of very good configurations for that query in a cost-based manner by consulting the query optimizer component of the database system. A physical design structure that is part of the selected configurations of one or more queries in the workload is referred to as a *candidate*. We use a greedy search strategy, called Greedy (m,k) [8] to realize this task. To recap, Greedy (m,k) algorithm guarantees an optimal answer when choosing up to m physical design structures, and subsequently uses a greedy strategy to add more (up to k) structures.

Merging: Since Candidate Selection operates only at the level of an individual query, if we restrict the final choice of physical design to only be a subset of the candidates selected by the Candidate Selection step, we can potentially end up with "over-specialized" physical design structures that are good for individual queries, but not good for the overall workload. Specifically, when storage is limited or workload is update-intensive, this can lead to poor quality of the final solution. The goal of the Merging step is to consider new physical design structures, based on candidates chosen in the Candidate Selection step, which can benefit multiple queries in the workload. The Merging step augments the set of candidates with additional "merged" physical design structures. The idea of merging physical design structures has been studied in the context of un-partitioned indexes [8], and materialized views [3]. Merging becomes a lot harder with the inclusion of partitioning, and requires new algorithmic techniques. Our techniques for merging partitioned indexes and materialized views are described in [4].

Enumeration: This step takes as input the union of the candidates (including candidates from Merging), and produces the final solution: a physical database design. We use Greedy (m,k) search scheme. As shown in [4], with the additional constraint that the physical design for each table should be aligned, introducing new candidates that are required (for alignment) eagerly can be unscalable. In [4], we describe how it is possible to delay the introduction of additional candidates that need to be introduced to satisfy the alignment constraint *lazily*, thereby greatly improving the scalability of the enumeration step.

3. Integrated Physical Design Recommendations

An important challenge in physical design tuning is being able to judiciously deal with the tradeoffs in performance of various physical design *features* available (indexes, materialized views, partitioning etc.). This tradeoff is challenging for the following two reasons. First, for a given query, different features can overlap in their ability to reduce the cost of execution for query.

Example 1. Consider the query: Select A, COUNT(*) FROM T WHERE X < 10 GROUP BY A. For this query several different physical design structures can reduce its execution cost: (i) A clustered index on (X); (ii) Table range partitioned on X; (iii) A non-clustered, "covering" index on (X, A); (iv) A materialized view that matches the query, and so on.

Second, these features can have widely varying storage and update characteristics. Thus in the presence of storage constraints or for a workload containing updates, making a global choice for a workload is difficult. For example, a clustered index on a table and horizontal partitioning of a table are both non-redundant structures

(incur negligible additional storage overhead); whereas non-clustered indexes and materialized views can potentially be storage intensive (similarly their update costs can be higher). However, non-clustered indexes (e.g., "covering" indexes) and materialized views can often be much more beneficial than a clustered index or a horizontally partitioned table.

Thus, a physical design tool that can give an integrated physical design recommendation can greatly reduce/eliminate the need for a DBA to make ad-hoc decisions such as: (a) how to stage tuning, e.g., pick partitioning first, then indexes, and finally materialized views; (b) How to divide up the overall storage to allocate for each step in this staged solution, etc. The following example (showing the interaction between indexes and horizontal partitioning) illustrates the pitfalls of a staged solution.

Example 2. Consider the query from Example 1 and suppose that we wish to consider only clustered indexes and horizontal range partitioning of the table. We compare two approaches for the query above. (1) A staged solution that first selects the best clustered index, and in the next step considers horizontal range partitioning of the table. (2) An integrated solution that considers both features together. Observe that both a clustered index on column X or a range partitioning on column X can help reduce the selection cost, whereas a clustered index on column A is likely to be much more beneficial than a horizontal range partitioning on A as far as the grouping is concerned. Thus, if in the first step of the staged solution we recommend a clustered index on X, then we can never expect to find the optimal solution for the query: a clustered index on A and horizontal range partitioning of the table on X. On the other hand, an integrated solution is capable of finding this solution since it considers these features together.

As noted earlier, DTA can recommend indexes, materialized views, and horizontal partitioning in an integrated manner. Although doing so is important for the reasons described above, when all of the physical design features are considered together, the space of configurations that need to be considered for a given workload can become very large. The techniques that DTA uses to reduce the space of configurations that are explored without impacting the quality of recommendations significantly have been summarized in Section 2.2 and are described in greater detail in [4].

Finally, we note that DTA allows DBAs to choose only a subset of the available physical design features should they wish to do so. For example, in certain environments, a DBA may not wish to consider materialized views. In this case, the DBA can specify that DTA should only consider indexes and partitioning as the physical design options.

4. Aligned Partitioning

As discussed earlier, DBAs often require the physical design to be *aligned* (i.e., a table and all of its indexes are partitioned identically) so that it is easy to add, remove, backup, and restore data partitions. Horizontal range partitioning is important for manageability reasons, but it can also have a significant impact on performance of the workload. Therefore, DTA allows users the option of specifying that the physical design should be aligned. Choosing this option implies that the physical design recommended by DTA will satisfy the property of alignment for each table. The impact of specifying the alignment requirement on DTA is that it constrains the overall search space that DTA needs to traverse. The key technical challenge arising out of alignment is that different queries that reference a given table T may benefit from different ways of partitioning T or indexes on T. Thus, efficiently finding a compromise that works well across all queries in the workload is difficult. DTA efficiently incorporates alignment constraints into its search algorithm by exploiting the fact that in many cases, it is sufficient to introduce new candidates for the purposes of satisfying alignment *lazily*. Such lazy introduction of candidates during the Enumeration step (Section 2.2), can significantly improve scalability. The details of this technique are described in [4].

5. Improved Scalablity

5.1 Scaling to Large Workloads

One of the key factors that affect the scalability of physical design tools is the size of the workload, i.e., the number of statements (queries, updates) in the workload. As explained earlier, the workload given to DTA for tuning can be very large. In such cases, a natural question to ask is whether tuning a much smaller subset of the workload would be sufficient to give a recommendation with approximately the same reduction in cost as the recommendation obtained by tuning the entire workload.

Two obvious strategies for determining a subset of the workload to tune have significant drawbacks. The approach of sampling the workload uniformly at random ignores valuable information about queries in the workload (such as cost and structural properties), and thus may end up tuning a lot more queries than necessary. The other strategy of tuning the top k queries by cost, such that at least a pre-defined percentage of the total cost of the workload is covered, suffers from a different problem. Queries in the workload are often *templatized* (e.g., invoked via stored procedures). In such cases, often all queries belonging to one template may have higher cost than any query belonging to another template. Thus, the above strategy can end up tuning a disproportionate number of queries from one template, while never tuning queries from a different template.

The technique of *workload compression* in the context of physical design tuning has been studied in [7]. The idea behind workload compression is to exploit the inherent templatization in workloads by partitioning the workload based on the "signature" of each query, i.e., two queries have same signature if they are identical in all respects except for the constants referenced in the query. The technique picks a subset from each partition using a clustering based method. We have adapted the above technique and integrated it into DTA, which allows us to dramatically reduce the amount of time spent in tuning without significantly compromising the quality of physical design recommendation. We demonstrate the effectiveness of this technique on large workloads in Section 7.4.

5.2 Reduced Statistics Creation

As part of its tuning, DTA needs to consider indexes that may not exist in the current database, i.e., "what-if" indexes [9]. To simulate the presence of such a what-if index to the query optimizer, DTA needs to create the necessary statistics that would have been included in that index had it been actually created. Thus for a given set of indexes, there is a corresponding set of statistics that DTA needs to create. This is achieved using the standard "create statistics" interface available in SQL Server (using the sampling option of create statistics for scalability). The problem is that despite the use of sampling, the naïve strategy of creating all statistics can become very expensive, particularly on large databases,, because each "create statistics" statement incurs I/Os for sampling the pre-defined number of pages from the table, sorting it, and creating the summary statistics.

When SQL Server creates a statistic on columns (A,B,C), it generates a histogram on the leading column only (i.e., column A) and computes *density* information for each leading prefix (i.e., (A), (A,B), (A,B,C)). The density of a set of columns is a single number that captures the average number of duplicates of that set of columns. Since density is defined on a *set* of columns, it is not dependent on the order of the columns, i.e., Density (A,B) = Density (B,A).

Example 3. Suppose that DTA needs to consider indexes on (A), (B), (A,B), (B,A) and (A,B,C). Using the naïve approach, we would need to create all five statistics. However, it is easy to see that if we only create statistics on (A,B,C) and (B), these two statistics contain the same information (histograms and density) as when all five statistics are created.

Thus our problem can be stated more formally as follows:

> Given a set of statistics $S = \{s_1, \ldots s_n\}$ where each s_i contains a histogram on its leading column and density information on each leading prefix; find a subset S' of S of smallest cardinality such that it contains the same histogram and density information as S does.

Note that we seek to minimize cardinality of S' rather than to minimize the time to create S'. In reality, this simplification is reasonable since for a large table, the cost of creating a statistic is typically dominated by the I/O cost of sampling the table, which is independent of the specific statistic being created.

DTA's solution to the above problem is outlined below.

Step 1. Using S, create two lists: H-List and D-List. The H-List is a list of columns over which histograms need to be built. The D-List is the set of column groups over which density information needs to be obtained. In essence, the H-List and D-List identify the distinct statistical information that still needs to be created. In Example 3 above, the H-List is {(A), (B)} and the D-List is {(A), (B), (A,B), (A,B,C)}. Note that we do not need (B,A) in the D-List since Density (A,B) = Density (B,A).
Step 2. From the remaining statistics in S, pick the one (say *s*) whose creation "covers" as many elements of H-list and D-List as possible. In the above example, we pick (A,B,C) and create statistics on it.
Step 3. Remove all elements from the H-List and D-List which have been covered by the creation of *s* in Step 2 above. Remove *s* from S. In our example, we would remove (A) from the H-List and (A), (A,B), (B,A) and (A,B,C) from the D-List., and (A, B, C) from S.
Step 4. Repeat 2-3 until both H-List and D-List are empty. In our example, we would end up creating {(A,B,C}, (B}}.

The above greedy algorithm works well in practice (see Section 7.5 for an experimental evaluation) since the cost of creating a statistic depends mainly on the table size and not much on how many columns are present in the statistic. Thus it is usually beneficial to pick the largest remaining statistic to create, which contains the most statistical information. A more formal treatment of this problem is an interesting area of future work.

We note that the technique presented above simply reduces redundant statistical information that DTA creates. This is orthogonal to the techniques presented in [10] for determining whether or not a particular statistic actually impacts the plan chosen (or cost of plan chosen) by the query optimizer. Thus both of these techniques can be applied to reduce creation of unnecessary statistics. Finally, we note that the technique presented in this

section will need to be modified if the database server uses different statistical information, for example, multi-column histograms.

5.3 Tuning in Production/Test Server Scenario

The process of tuning a large workload can impose significant overhead on the server being tuned since DTA needs to potentially make many "what-if" calls to the query optimizer component. In enterprise databases, it is common for DBAs to use *test servers* in addition to the production server(s). A test server can be used for a variety of purposes including performance tuning, testing changes before they are deployed on the production server and so on. A straightforward way of reducing the impact of tuning on a production server is to use a test server as follows:

- Step 1: Copy the databases one wants to tune from the production server to the test server.
- Step 2: Tune the workload on the test server.
- Step 3: Apply the recommendation that one gets on the test server to the production server.

The advantage of such a simplistic approach is that once the databases are copied out to the test server, then there is no tuning overhead imposed on the production server. However, the approach suffers from many drawbacks that severely limit its applicability. First, the databases can be large (production databases can run into hundreds of gigabytes or more) or changing frequently. In such situations, copying large amounts of data from production to test server for the purposes of tuning can be time consuming and resource intensive. Second, the hardware characteristics of test and production servers can be very different. Production servers tend to be much more powerful than test servers in terms of processors, memory etc. Since the tuning process relies on the optimizer to arrive at a recommendation and that in turn is tied to the underlying hardware, this can lead to vastly different results on the test server.

DTA provides the novel capability of exploiting a test server, if available, to tune a database on a production server *without* copying the data itself. The key observation that enables this functionality is that the query optimizer relies fundamentally on the *metadata* and *statistics* when generating a plan for a query. We leverage this observation to enable tuning on the test server as outlined in Figure 2.

The steps in the tuning process are as follows:

- Step 1: Copy the *metadata* of the databases one wants to tune from the production server to the test server.. We do *not* import the actual data from any tables. Note that this requires not only importing the (empty) tables, and indexes, but also all views, stored procedures, triggers etc. This is necessary since the queries in workload may reference these objects. The metadata can be imported using the scripting capability that is available in today's database systems. This is generally a very fast operation as it requires working with system catalog entries and does not depend on data size.

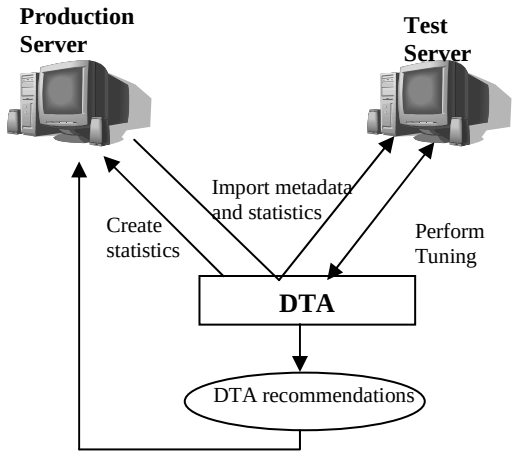

Figure 2. Tuning in a Production/Test Server Scenario.

- Step 2: Tune the workload on the test server. For getting the same plan on the test server as we would have got on the production server, we need two important functionalities from the database server. (1) During the tuning process, DTA may determine that certain statistics may need to be present (on the test server) so that the query optimizer can optimize the query accurately. However the statistics creation requires access to the actual data, and that is present only on production server. When that happens, DTA imports statistics from the production server (creating the statistics on the production server if necessary) and into the test server. (2) The hardware parameters of production server that are modeled by the query optimizer when it generates a query plan need to be appropriately "simulated" on the test server. For example, since query optimizer's cost model considers the number of CPUs and the available memory, these parameters need to be part of the interface that DTA uses to make a "what-if" call to the query optimizer. Microsoft SQL Server 2005 has been extended to provide us with both of these functionalities.
- Step 3: Apply the recommendation that one gets on the test server to the production server.

Note that the only overhead introduced on the production server during tuning is the creation of additional statistics (if any) that are necessary as tuning progresses (Step 2). The rest of the tasks that include simulating what-if interfaces, optimizing queries under different physical designs etc. are all performed on the test server. In

Section 7.3, we present experimental results that quantify the reduction in overhead on the production server as a result of exploiting this feature in DTA.

6. Enhanced Usability

6.1 Scriptabilty

As described in the introduction, tools for automated physical database design have emerged over the past few years and become part of commercial database systems. As such physical database design tools become more widely used, it can be advantageous to define a public XML schema for physical database design that forms the basis of input and output to such tools. First, having a public schema facilitates development of other tools that can program against the schema. Moreover, scriptability of the physical design tool can become much easier once such a schema is available. Second, as physical design tools evolve over time, having an XML schema makes it easy to extend it to meet the changing input/output needs of physical design tools. Finally, an XML schema makes it possible for different users/tools to interchange and communicate physical database design information within and across organization boundaries. We have defined such an XML schema for physical database design that is used by DTA, and that will be made public with the release of Microsoft SQL Server 2005 at [11]. The schema is described using the "XML Schema" schema definition language [19,20]. The schema defines elements to describe the common entities in physical database design, for example, databases (and tables within them), workloads, configurations, and reports.

6.2 Customizability via Partial Specification of Physical Design

In certain cases, DBAs may need to tune workload while being able to specify the physical design partially. Consider the following scenario where the DBA needs to decide whether a large table T (e.g., a fact table) should be range partitioned by month or quarter. Although either form of partitioning may be acceptable from a manageability standpoint, the DBA would prefer the option that results in better performance for the workload.

DTA allows the user to provide as (an optional) input a *user specified configuration*. A user specified configuration is a set of physical design structures that is valid, i.e. all physical design structures specified must be realizable in the database. An example of a physical design that is not valid is when there are more than one ways of clustering specified on the same table as a part of the same user specified configuration. DTA tunes the given workload and provides a recommendation while honoring the constraint that the user specified configuration is included in the recommendation provided by DTA. Thus in the above scenario, the DBA can first invoke DTA with a user specified configuration C_1 in which T is partitioned by month, and have DTA recommend the best physical design while honoring the constraint. The DBA can then run DTA again, this time with a user specified configuration C_2 in which T is partitioned by quarter. Finally, the DBA can compare the two recommendations and pick the one with better overall performance.

Since DTA never needs to physically alter the partitioning of the table, a DBA can try out various physical design alternatives efficiently without interfering with the normal database operations. In the scenario mentioned above, the DBA never needs to physically alter the partitioning of the large table (which can be an expensive operation) while making the decision of which partitioning to pick.

6.3 Exploratory Analysis and Iterative Tuning

A common scenario that DBAs encounter is the need to perform exploratory or "what-if" physical design analysis to get answers to question of the following type: What is the performance impact on a given set of queries and updates if I add a particular index or change the partitioning of a particular table? DTA enables this functionality through the *user specified configuration* mechanism described earlier (see Section 4.2). In particular, the user can specify a configuration, i.e., a valid physical design consisting of existing or hypothetical objects, and request DTA to *evaluate* the configuration for a given workload. DTA exploits the "what-if" interfaces of Microsoft SQL Server [9] to simulate the given configuration, and it evaluates each statement in the workload for the given configuration. DTA returns as output the expected percentage change in the workload cost compared to the existing configuration in the database. In addition, DTA provides a rich set of analysis reports that provides details about changes in cost of individual statements in the workload, usage of indexes and materialized views, and so on.

Another common scenario that DBAs face is the need for iterative tuning and refinement. For example, a DBA obtains a recommendation from DTA for a particular workload, and wishes to modify the recommendation and re-evaluate the modified configuration for the same workload. The DBA may repeat such evaluation until she is satisfied with the performance/impact of the configuration. Once again, such a scenario becomes simple via the user specified configuration feature of DTA. Moreover, given that the DTA input and output conform to an XML schema, it is easy for users or other automated tools to take the output configuration from one run of DTA and feed a modified version of it as input into a subsequent run of DTA.

7. Experiments

In this section, we present results of experiments that evaluate:

- Quality of recommendations by DTA on several customer workloads, by comparing it to a hand-tuned physical design.
- Quality of DTA on TPC-H 10GB benchmark workload.
- Impact on production server overhead because of DTA's ability to exploit a test server for tuning.
- Impact of workload compression technique (Section 5.1) on quality and scalability of DTA.
- Effectiveness of technique for reduced statistics creation (Section 5.2).
- An end-to-end comparison of DTA with Index Tuning Wizard for SQL Server 2000.

7.1 Quality of DTA vs. hand-tuned design on customer workloads

In this experiment, we compare the quality of DTA's physical design recommendation with a manually-tuned physical design. The databases and workloads used in this experiment were obtained from internal customers of Microsoft SQL Server. The methodology we adopt for this experiment is as follows. For each workload, we obtain the optimizer estimated cost of the workload for the current physical design (put in place by the DBA of that database). We refer to this cost as $C_{Current}$. We then drop all physical design structures (except those that enforce referential integrity constraints) and once again obtain the cost of the workload. We refer to this cost as C_{raw}. We then tune the workload using DTA, and obtain the cost of the workload for DTA's recommended configuration. We refer to this cost as C_{DTA}. We define the quality of DTA as $(C_{raw} - C_{DTA})/C_{raw}$ and the quality of the hand-tuned design as $(C_{raw} - C_{Current})/C_{raw}$, i.e., the percentage reduction relative to C_{raw}.

A brief overview of each customer database/workload used in this experiment in shown in Table 1. We note that depending on the type of application, the amount of updates in these workloads varies (higher for CUST3).

Database	Total size (GB)	#DBs	#Tables
CUST1	10.7	31	4374
CUST2	1.4	6	48
CUST3	105.9	23	1624
CUST4	0.06	2	11

Table 1: Overview of customer databases and workloads.

As can be seen from Table 2, in all the customer workloads, the quality of DTA is comparable to the hand-tuned physical design for the CUST 1 workload. DTA is significantly better for CUST2 and CUST4. In CUST4 (which is a small database), the hand-tuned design consisted of only the primary-key and unique indexes, whereas DTA was able to considerably improve upon that design. In CUST3, the hand-tuned design was worse than the raw configuration due to presence of updates. For this workload, DTA correctly recommended no new physical design structures to be created. This experiment demonstrates DTA's ability to effectively tune real-world workloads. Finally, we note that the tuning time of DTA can vary depending on the complexity of queries in the workload. For example, in this experiment DTA tuned anywhere between 134 events/min (CUST4) to about 500 events/min (CUST2).

Workload	Quality of hand-tuned design	Quality of DTA	#events tuned	Tuning time (hr:min)
CUST1	82%	87%	15K	0:35
CUST2	6%	41%	252K	8:21
CUST3	-5%	0%	176K	15:14
CUST4	0%	50%	9K	1:07

Table 2: Quality of DTA vs. hand-tuned design on customer workloads.

7.2 Evaluation on TPC-H 10GB benchmark workload

In this experiment, we evaluate DTA on the TPC-H 10GB benchmark workload [17]. We start with a raw database (i.e., consisting of only indexes that enforce referential integrity constraints), and tune the benchmark workload consisting of 22 queries. The size of the raw database is about 12.8 GB. The expected improvement reported by DTA for this workload is 88% (the total storage space allotted was three times the raw data size). We implemented DTA's recommendations and executed the queries (warm runs). For each query, we conduct 5 warm runs, discard the highest and lowest readings and take the average of the remaining 3 readings. The actual improvement in execution time for the workload is 83%. This experiment shows that (a) DTA is able to achieve significant improvements on a workload containing complex queries; and (b) the query optimizer's estimates are reasonably accurate for this workload.

7.3 Production/Test Server Scenario

The following experiment illustrates the impact of DTA's ability to use a test server for tuning purposes. The database used for this experiment is TPC-H [17], the 1 GB configuration. TPCHQ1-I (and TPCH22-I) denotes the first query (resp. 22 queries) in TPC-H benchmark and only indexes are selected for tuning. TPCHQ1-A (and

TPCH22-A) denotes the first query (resp. all 22 queries) in TPC-H benchmark and both indexes and materialized views are considered during the process of tuning. We use a test server to tune the production server and compare the overhead on the production server to the case where we had no test server. The *overhead* is measured as the total duration (i.e., elapsed time) of all statements that were submitted to the production server by DTA. No other queries were running on the test and production servers during the period of the experiment.

Figure 3 shows that even in the simplest case of single query tuning and only indexes are considered, TPCHQ1-I, during tuning, the overhead on the production server is reduced by about 60%. For TPCH22-A the reduction is significant (90%). Note that as the complexity of tuning increases (e.g., larger workload, or more physical design features to consider) we expect the reduction in overhead to become more significant (as seen in Figure 3). We have observed similar benefits of reduced overhead for several customer workloads as well.

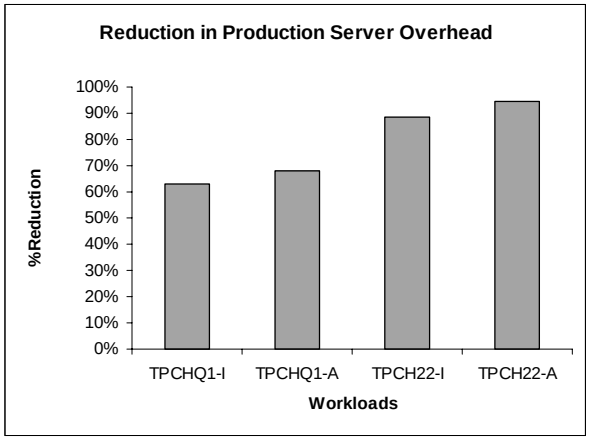

Figure 3. Reducing load on production server by exploiting test server

7.4 Effectiveness of Workload Compression

The following experiment illustrates the impact of workload compression (see Section 5.1) on the quality of DTA's recommendations as well on its running time for a few real and synthetic workloads. We use three database and workloads for this comparing (a) TPCH22 queries on TPC-H 1GB data; (b) PSOFT is a customer database (a PeopleSoft application) where the workload consists of about 6000 queries, inserts, updates and deletes, the database is about .75 GB; and (c) SYNT1 is a synthetic database that conforms to the well known SetQuery benchmark schema, and contains 8000 queries in the workload. The queries are SPJ queries with grouping and aggregation, with approximately 100 distinct "templates." Each query was generated by randomly selecting selection columns, grouping columns, aggregation columns/functions, etc.

Table 3 summarizes the results of this experiment. For TPCH22 (which contains only 22 queries), the queries in the workload are all very different and no workload compression is possible using our technique. The benefits of workload compression are much larger when we have larger workloads and this is reflected in PSOFT and SYNT1. We observe a speed up of more than a factor of 5x for the PSOFT workload and more than 40x for SYNT1. This is because these workloads have many queries but relatively few distinct *templates* (see Section 5.1). For PSOFT, DTA tunes about 10% of the number of queries in the workload, whereas for SYNT1 this was about 2%. Thus, by leveraging the ideas of workload compression, we are able to significantly improve the performance of DTA with almost no adverse impact on the quality of recommendations.

Database and Workloads	% Decrease in Quality compared to no workload compression	Reduction in Running Time of DTA compared to no workload compression
TPCH22	0	-1%
PSOFT	0.5%	5.8x
SYNT1	1%	43x

Table 3: Impact of Workload Compression on Quality and Running Time of DTA.

7.5 Impact of Reduced Statistics Creation

In this experiment we measure the reduction in statistics creation time using techniques described in Section 5.2 for two different workloads, TPC-H 10GB and PSOFT. We measure: (a) reduction in number of statistics created (b) reduction in statistics creation time. The number of statistics created was reduced by 55% for TPC-H and about 24% PSOFT. Similarly, the reduction in statistics creation time was 62% and 31%, respectively. In both of these cases there was no difference in the quality of DTA's recommendation since the above technique only reduces creation of redundant statistical information.

7.6 End-to-End comparision against ITW in SQL Server 2000 (SS2K)

The Index Tuning Wizard (ITW) in Microsoft SQL Server 2000 is one of the currently available physical design tools for Microsoft SQL Server. In this section, we conduct an end-to-end comparison of the quality and running time of DTA compared to ITW in Microsoft SQL Server 2000. We compare the overall quality of the recommendation produced by the tools, and running time of DTA compared to that of Index Tuning Wizard in Microsoft SQL Server 2000. We measure the quality by

the percentage improvement in the optimizer estimated cost for the workload.

The experiments below compare the running time and quality. We use the three databases and workloads TPCH22, PSOFT and SYNT1 described in Section 7.4. To ensure a fair comparison, in this experiment, we consider only indexes and materialized views (since ITW is not capable of recommending partitioning). The comparisons were performed by running both the tools against the same server (a Microsoft SQL Server 2000 server). Figures 4 (and 5) compare the quality (resp. running time) of DTA compared to ITW in SQL Server 2000. We observe that for the same database/workload, we get comparable recommendation qualities for TPCH22, PSOFT and SYNT1 (DTA is slightly better in all cases), and DTA is significantly faster for the large workloads.

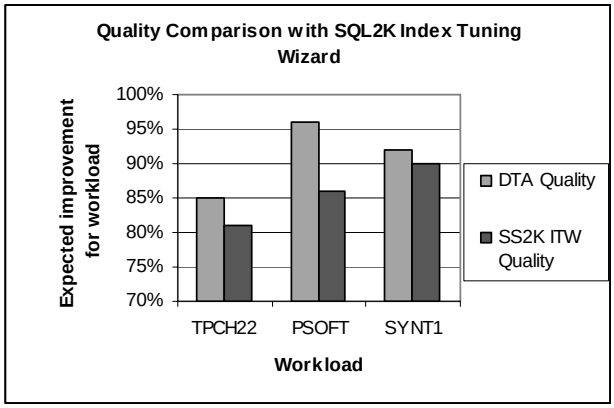

Figure 4. Quality of recommendation of DTA compared to Index Tuning Wizard for Microsoft SQL Server 2000.

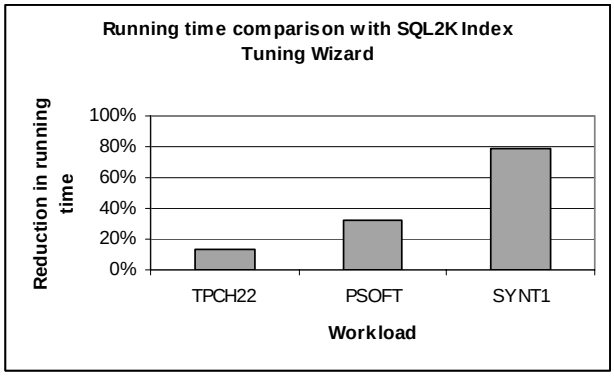

Figure 5. Running time of DTA compared to Index Tuning Wizard for Microsoft SQL Server 2000

8. Related Work

There are currently several automated physical design tools developed by commercial database vendors themselves, and by third-party tool vendors. To the best of our knowledge, the Database Tuning Advisor (DTA) for Microsoft SQL Server 2005 is the first tool with the capability of providing a fully integrated recommendation for indexes, materialized views and horizontal range partitioning, i.e., DBAs do not need to stage the selection of different physical design features.

Today's tools support tuning of different sets of physical design features. Index Tuning Wizard in Microsoft SQL Server 2000 [2] was the first tool to recommend indexes and materialized views in an integrated manner. DTA builds upon the Index Tuning Wizard and significantly advances upon it in several aspects including functionality, scalability and manageability. The algorithmic techniques underlying DTA were developed in the context of the AutoAdmin project [1] at Microsoft, whose goal is to reduce TCO by making database systems more self-tuning and self-managing.

IBM's DB2 Advisor [21] recommends indexes and materialized views, and IBM's Partition Advisor [11] recommends horizontal hash partitioning in a shared-nothing parallel database. Oracle's Index Tuning Wizard [12] and BMC's Index Advisor [6] recommend indexes. Leccotech [13] provides tools for tuning a SQL statement by recommending indexes, and also by suggesting rewritten SQL that can obtain a potentially better execution plan.

The idea of workload compression as a technique to improve scalability of workload based tuning tools was first presented in [7]. We have adapted these ideas and implemented them in the context of DTA, to obtain significant improvement in scalability for large workloads (see Section 5.1). The technique presented in Section 5.2 for reducing the redundant statistical information from a given set of statistics is complementary to the techniques presented in [10] for determining whether or not a particular statistic actually affects the plan chosen (or cost of plan chosen) by the query optimizer. Thus both of these techniques can be applied to reduce creation of unnecessary statistics. The techniques of pruning column-groups based on frequency of occurrence in the workload and cost, and algorithmic techniques for handling alignment requirements inside a physical database design tool as well as merging in the presence of horizontal partitioning were developed in the context of DTA and have been presented in [4].

Finally, there are several aspects of DTA's functionality which are the first for any physical database design tool. These include: (a) the ability to specify an alignment constraint as well as a partially specified configuration; (b) an XML schema for input/output that enhances scriptability and customizability of the tool; and (c) efficient tuning in production/test server scenarios.

9. Conclusion

As tuning tools continue to encompass more features, it is important to keep the usage of these tools simple for DBAs by providing integrated recommendations for various features when possible. Database Tuning Advisor for Microsoft SQL Server 2005 is built around a scalable architecture that makes it easy to incorporate new physical design features, while ensuring integrated recommendations. In the future, we will continue to expand the scope of DTA to include other important aspects of performance tuning.

10. Acknowledgements

We are grateful to Alex Boukouvalas, Campbell Fraser, Florian Waas, and Cesar Galindo-Legaria for helping with necessary extensions to Microsoft SQL Server 2005 for DTA. We thank Djana Milton, Maciek Sarnowicz, and Dima Sonkin who developed the GUI component of DTA.

11. References

[1] The AutoAdmin project. http://research.microsoft.com/dmx/AutoAdmin .

[2] Agrawal S., Chaudhuri S., Kollar L., and Narasayya V. Index Tuning Wizard for Microsoft SQL Server 2000. White paper. http://msdn.microsoft.com/library/techart/itwforsql.htm

[3] Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated Selection of Materialized Views and Indexes for SQL Databases. Proceedings of VLDB 2000.

[4] Agrawal, S., Narasayya, V., and Yang., B. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In Proceedings of ACM SIGMOD 2004.

[5] Agrawal R., and Ramakrishnan S. Fast Algorithms for Mining Association Rules for Large Databases. Proceedings of VLDB 1994.

[6] BMC Index Advisor. http://www.bmc.com .

[7] Chaudhuri S., Gupta A., and Narasayya V. Workload Compression. Proceedings of ACM SGMOD 2002.

[8] Chaudhuri, S., and Narasayya, V. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB 1997.

[9] Chaudhuri, S., and Narasayya, V. AutoAdmin "What-If" Index Analysis Utitlity. Proceedings of ACM SIGMOD 1998.

[10] Chaudhuri S., and Narasayya V. Automating Statistics Management for Query Optimizers. Proceedings of ICDE 2000.

[11] http://schemas.microsoft.com/sqlserver/2004/07/dta/ XML schema for physical database design used by DTA.

[12] http://otn.oracle.com/products/oracle9i/index.html.

[13] LeccoTech's Performance Optimization Solutions for Oracle. White Paper. LeccoTech Inc. http://www.leccotech.com

[14] Program for TPC-D data generation with Skew. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew .

[15] Rao, J., Zhang, C., Lohman, G., and Megiddo, N. Automating Physical Database Design in a Parallel Database. Proceedings of the ACM SIGMOD 2002.

[16] Stohr T., Martens H.., and Rahm E.. Multi-Dimensional Aware Database Allocation for Parallel Data Warehouses. Proceedings of VLDB 2000.

[17] TPC Benchmark H. Decision Support. http://www.tpc.org

[18] Valentin, G., Zuliani, M., Zilio, D., and Lohman, G. DB2 Advisor: An Optimizer That is Smart Enough to Recommend Its Own Indexes. Proceedings of ICDE 2000.

[19] World Wide Web Consortium. Extensible Markup Language (XML). W3C Recommendation (October 2000). http://www.w3.org/XML/. Web-page, 2003.

[20] World Wide Web Consortium. XML Schema. W3C Recommendation (May 2001). http://www.w3.org/XML/Schema. Web-page, 2003.

[21] Zilio D. et al. Recommending Materialized Views and Indexes with IBM's DB2 Design Advisor. Proceedings of International Conference on Autonomic Computing, 2004.