# AutoAdmin "What-if" Index Analysis Utility

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

## Abstract

As databases get widely deployed, it becomes increasingly important to reduce the overhead of database administration. An important aspect of data administration that critically influences performance is the ability to select indexes for a database. In order to decide the right indexes for a database, it is crucial for the database administrator (DBA) to be able to perform a *quantitative analysis* of the existing indexes. Furthermore, the DBA should have the ability to propose hypothetical ("what-if") indexes and quantitatively analyze their impact on performance of the system. Such impact analysis may consist of analyzing workloads over the database, estimating changes in the cost of a workload, and studying index usage while taking into account projected changes in the sizes of the database tables. In this paper we describe a novel index analysis utility that we have prototyped for Microsoft SQL Server 7.0. We describe the interfaces exposed by this utility that can be leveraged by a variety of front-end tools and sketch important aspects of the user interfaces enabled by the utility. We also discuss the implementation techniques for efficiently supporting "what-if" indexes. Our framework can be extended to incorporate analysis of other aspects of physical database design.

## 1. Introduction

Enterprise-class databases require database administrators who are responsible for performance tuning. Database Administrators (DBAs) need to take into account resources on the database system, application requirements, and characteristics of the workload and DBMS. With large-scale deployment of databases, minimizing database administration function becomes important. The AutoAdmin project at Microsoft Research [1] is investigating new techniques to make it easy to tune external and internal database system parameters to achieve competitive performance. One important area where tuning is required is in determining physical database design and specifically in the choice of indexes to build for a database.

The index selection problem has been studied since the early 70's and the importance of this problem is well recognized. Despite a long history of work in this area, there are two fundamental reasons why this problem has not been addressed. First, index selection is intrinsically a hard search problem. For an enterprise class database, there are a large number of possible single and multi-column indexes. Moreover, since modern query processors use indexes in several innovative ways (e.g., index intersection, indexed-only access), it is hard to enumerate the search space efficiently. Next, the problem of picking the right of set of indexes cannot be simply solved by a good search algorithm. Enterprise databases are simply too complex for the DBA to hit the "accept " button on the recommendations of an index selection tool until he/she has been able to perform an *impact analysis* of the suggested index recommendations. Some examples of impact analysis are: (1) Which queries and updates that we executed in the last 3 days will slow down because of the changes? (2) Which queries will benefit from the index that you are proposing to add and to what extent? To the best of our knowledge, no adequate utility exists that allows DBAs to undertake an impact analysis study. Indeed, even in the absence of an index selection tool, such an index analysis utility is of great importance since it allows the DBA to propose hypothetical ("what-if") indexes and quantitatively analyze their impact on performance of the system. In this paper we use the terms hypothetical and "what-if" interchangeably. Such a utility also provides a natural back-end for an index selection tool to enumerate and pick an appropriate set of indexes by using the index analysis utility as the "probe" to determine the goodness of the set of indexes. In the context of the AutoAdmin project, we have built an index selection tool as well as an index analysis utility. The index selection tool has been described in [4] and it leverages off the index analysis component. This paper focuses on the index analysis utility. We now provide an overview of the "what-if" index analysis utility and the system architecture for index selection in AutoAdmin.

### 1.1 Overview of Architecture

Figure 1 illustrates the related system components for the task of index selection. We use Microsoft SQL Server 7.0 as the database server. In this paper, we use the term *configuration* to mean a set of indexes, and the sizes of each table in the database. *A hypothetical configuration* may consist of existing ("real") indexes as well as hypothetical ("what-if") indexes. We define a *workload* to be a set of SQL statements. The hypothetical configuration analysis (HCA) engine supports two sets of interfaces for (a) simulating a hypothetical configuration (b) summary analysis on the data resulting from the simulation. The HCA engine can be implemented either as a library that client tools can link to, as a middle-ware process that serves multiple clients or directly as server extensions. In our prototype, we have implemented the HCA engine as a dynamic linked library (DLL).

Using the hypothetical configuration simulation interfaces, client tools can define workloads, define

hypothetical configurations, and evaluate a workload for a hypothetical configuration. By evaluating a workload for a configuration, we can *estimate* the cost of queries in the workload *if* the configuration were made "real" (i.e. the indexes in the configuration were materialized). In addition, we can tell for each query, which indexes in the configuration would be used to answer that query. The dotted line in Figure 1 shows that the interfaces for hypothetical configuration simulation are available directly as SQL Server extensions. However, for software engineering reasons, the HCA engine encapsulates this functionality and provides the complete set of interfaces for index analysis to client tools. In this paper, we will discuss the HCA interfaces and discuss its implementation over a SQL Database, describing the necessary extensions to server interfaces.

The summary analysis interface makes it possible to perform sophisticated summarization of workloads, configurations, performance of the current configuration and projected changes for a new configuration. Examples of such analysis are: (a) Analyze a workload by counting each type of query – SELECT, INSERT, UPDATE, DELETE. (b) Estimate the storage space of a hypothetical configuration. (c) Identify queries in the workload that are most affected by the addition (or removal) of indexes.

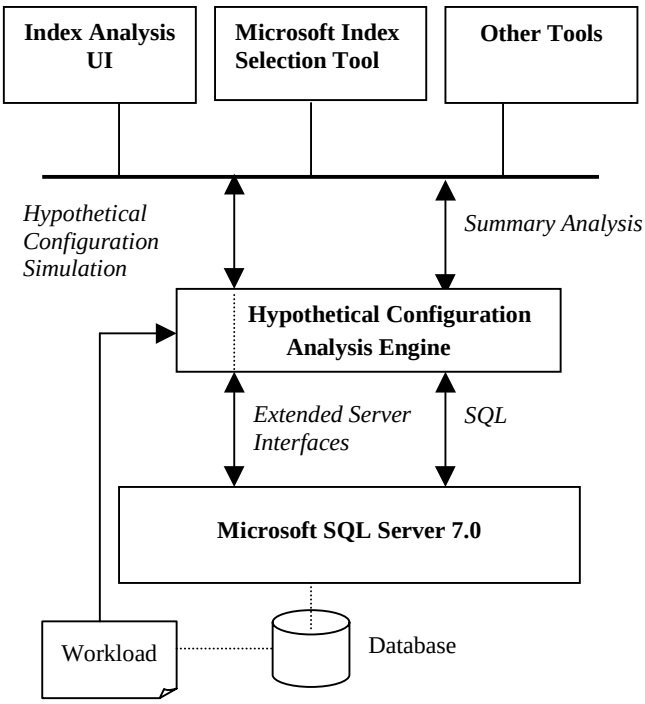


**Figure 1. Architecture Overview**

The rest of the paper is organized as follows. In Section 2, we review the related work in this area. Section 3 presents the interfaces for hypothetical configuration simulation, and describes their implementation on Microsoft SQL Server. Section 4 describes the summary analysis interfaces and their implementation, and provides an example "session" that illustrates how the synergy among different summary analysis components can assist the DBA in selecting the right indexes for a database. We discuss future work and conclude in Section 5.

## 2. Related Work

There is a substantial body of literature on physical database design dating back to the early 70's. Nonetheless, to the best of our knowledge, no previous work has addressed the problem of estimating the impact of possible changes to the index configurations and database size in a comprehensive manner. Stonebraker [12] discusses the use of views in simulating hypothetical databases. His approach creates a query to "simulate the hypothetical database" and therefore relies on actual execution. This is very computation intensive. Our approach is based on relative estimation of the cost that enables a large class of analysis at low cost. Furthermore, we have provided an efficient mechanism to implement hypothetical structures using sampling based techniques.

The index selection algorithms in [4,6] can exploit the infrastructure presented in this paper for exploring the space of alternatives to pick an optimal index configuration. Those papers focus on efficiently searching the space of alternatives. In addition to the above, there is a significant body of work in index selection, including [2,5,7,9,11]. Most of the other work in index selection has the serious shortcoming that the index selection tools do not stay in step with the optimizer (see [3] for a discussion). In any case, these papers do not discuss support for hypothetical configurations.

## 3. Simulating Hypothetical Configurations

In this section we present the interfaces for hypothetical configuration simulation and describe how these interfaces are implemented efficiently. We first present the foundational concepts supported by the HCA engine that set the context for the rest of the section.

*(a) Workload.* A workload consists of a set of SQL statements. Most modern databases support the ability to generate a representative workload for the system by logging activity on the server over a specified period of time. For example, in Microsoft SQL Server, the SQL Server Profiler provides this functionality.

*(b) Hypothetical Configuration.* A configuration consists of a set of indexes that are consistent with schema constraints. For example, if a table has a uniqueness constraint on a column C, then an index on C must be part of every configuration. Likewise, a table can have at most one clustering index. A configuration may also have a *database scaling value* associated with it. A database scaling value is a set of multipliers that captures the *size* of the database. A multiplier $m_j$ is associated with each table $T_j$. A hypothetical configuration with a database scaling value represents a database where each table $T_j$ in the database has $m_j$ times the number of rows in the current database. Thus, the scaling factor can be used to represent not only a database that is significantly larger or smaller than the current database, but also a database where the relative sizes of the tables are different from

today's database. As a result, the HCA engine makes it possible able to project changes to the current database along two dimensions: changes in configuration as well as changes in database size.

(c) *Estimation of projected changes.* The *effect* of the projected changes to the current database is captured in two ways. First, the HCA engine supports the ability to estimate the *cost* of a query in the workload with respect to a hypothetical configuration. Second, the HCA engine can estimate the *index usage* since it can project which indexes in the hypothetical configuration would be used to answer a query in the workload.

## 3.1 Interfaces for Hypothetical Configuration Simulation

Our approach to designing the interfaces for hypothetical configuration simulation is influenced by the observation that simulating a hypothetical configuration consists of estimating (a) the *cost* of queries in the workload and (b) *usage* of indexes. In presenting these interfaces, we focus on their functionality, and not on syntax.

*A) Define Workload < workload_name > [From <file> | As ($Q_1$ ,$f_1$), ($Q_2$ ,$f_2$), …. ($Q_n$,$f_n$)]*
*B) Define Configuration <configuration_name> As ($Table_1$, $column\_list_1$),..,($Table_i$, $column\_list_1$)*
*C) Set Database Size of <configuration_name> As ($Table_1$, $rowcount$),…, ($Table_i$, $row\_count_i$)*
*D) Estimate Configuration of <workload_name> for <configuration_name>*
*E) Remove [Workload <workload_name>| Configuration <configuration_name> | Cost-Usage <workload_name>, <configuration_name>]*

For each command (A)-(E), we now describe the semantics of the command, and the information generated when the command is executed . We refer to this information as *analysis data*. For simplicity, we present the analysis data as relations in non-first-normal form). In our implementation, we use multiple (normalized) tables to store analysis data.

The *Create Workload* command associates a name with a set of queries. These queries can be specified from a file or can be passed in directly through the command. The workload information generated as a result of executing the command is shown in Table 1. A *frequency* value $f_i$, which is associated with query $Q_i$ in the workload. The frequency is interpreted by the HCA engine to mean that the workload consists of $f_i$ copies of query $Q_i$. In addition, the HCA engine uses extended server interfaces that expose the parsed information of a query to associate a set of properties with a query. Some examples of query properties are: (a) The SQL Text (b) Set of tables referenced in the query (c) Columns in the query with conditions on them.

The *Define Configuration* command has the effect of registering a new configuration and associating a set of indexes with that configuration. The indexes may be existing ("real") indexes or hypothetical indexes. If the index exists, then the index name can optionally be substituted for (Table, column_list). The *Set Database Size* command sets the scaling values for each table in the database. Table 2 shows the information associated with a configuration by the HCA engine. The information associated with each index is shown in Table 3. Since indexes (real and hypothetical) are entities supported by the database, this information is available in the system catalogs. (In Section 3.2 we describe how a *hypothetical* index is created). We note that the syntax of the *Define Configuration* is general enough to include other features of physical database design (e.g. materialized views) in addition to indexes. However, the main challenge in adding new hypothetical features arises not from having to extend the syntax of *Define Configuration*, but because the creation and use of these hypothetical features must be supported efficiently.

| Workload name | Query ID | Frequency | Query Properties |
|---|---|---|---|
| Wrkld_A | 1 | 1 | <SQL Text>, {$T_1$, $T_2$}, etc. |

**Table 1. Workload information**

| Configuration Name | Indexes in Configuration | Scaling values |
|---|---|---|
| Current_Conf | Ind_A, Ind_B, Ind_D | ($T_1$, 1), ($T_2$, 5) |

**Table 2. Configuration Information**

| Index Name | Table Name | Num Rows | Num. Pages | Cols. | Index Statistics |
|---|---|---|---|---|---|
| Ind_A | R | 100,000 | 1865 | R.a | <histogram> |

**Table 3. Index Information**

| Config name | Query ID | Cost | Indexes Used |
|---|---|---|---|
| New_Config | 1 | 0.02 | Ind_A, Ind_D |
| New_Config | 2 | 0.11 | Ind_B |

**Table 4. Information generated by *Estimate Configuration***

The result of executing *Estimate Configuration* for a given workload and configuration is a relation that has the format shown in Table 4. Conceptually, the relation has as many rows as there are queries in the workload. The unit of *Cost* is relative to the total cost of the workload in the current configuration. The attribute *Indexes-Used* for a query represents the indexes that are expected to be used by the server to answer the query if the hypothetical configuration existed.

The *Remove* command provides the ability to remove analysis data generated by commands (A)-(D). We observe that when *Remove Workload* (respectively *Configuration*) is invoked, all information about the Workload (Configuration) is removed, including any cost and usage information. However, when *Remove Cost-Usage* is invoked, only the cost-usage information for the specified workload and configuration is removed, but the workload and configuration information is retained.

Finally, we note that user interfaces in the AutoAdmin index analysis utility makes it easy to define a workload and configuration. In Microsoft SQL Server, a representative workload for the system can be generated by logging events at the server over a specified period of time using the SQL Server Profiler to a file. In addition, filters can be specified so that only relevant events are logged. Alternatively, a workload can be dynamically created from the Query Analyzer interface. In this approach, a highlighted buffer of queries is used to define a workload dynamically. While defining the configuration, the user-interface presents successive screens to set the indexes and the table sizes. In each of the screens, the user (typically the DBA) is presented with the list of objects for the current ("true") configuration and can create a new configuration by adding (or removing) indexes to the current configuration.

## 3.2 Implementing the Hypothetical Configuration Simulation Interface

The commands (A)-(C) are primarily definitional and do not pose implementation challenges. Likewise command (E) involves deleting rows from the analysis data tables corresponding to the workload, configuration or cost-usage specified in the command. Therefore, in this section we focus on the issue of efficiently implementing the core functionality of the hypothetical configuration simulation interface: the *Estimate Configuration* command.

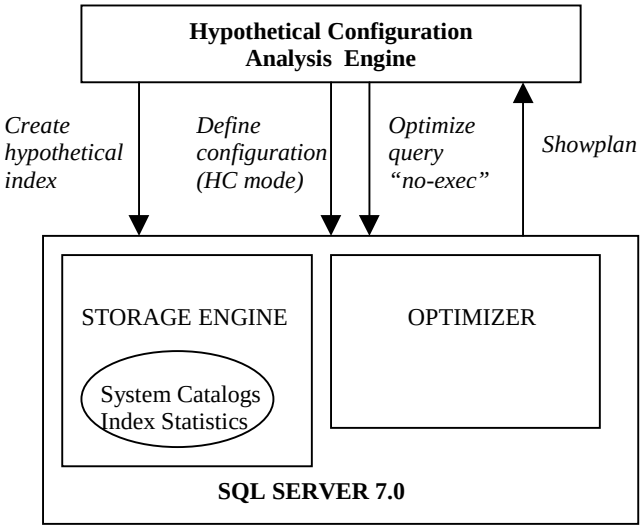


**Figure 2. Interfaces between HCA Engine and SQL Server**

The simplest option of simulating a hypothetical configuration by *physically* altering the current configuration is not viable since it incurs the serious overhead of dropping and creating indexes. Perhaps, even more seriously, such an approach is flawed since changing indexes affects operational queries and can seriously degrade the performance of the system. Likewise, updating the system tables with the database scaling value can lead to error in optimizer's estimates of operational queries. Therefore, we need an alternative where indexes in the hypothetical configuration do not need to be constructed and where changes in the database scaling value does not affect the system tables directly.

The solution to this problem relies on the observation that the *cost* metric of a query that we are interested in is the optimizer-estimated cost and *not* the actual execution cost. This metric is justified since the "consumer" of a configuration is the optimizer. In other words, unless the optimizer finds a hypothetical index useful, it is unlikely to make use of that index when it is made "real" (see [4] for additional justification). An optimizer's decision on whether or not use an index is solely based on the statistical information on the column(s) in the index. Such information consists of (a) a histogram on the column values on which the index is defined (b) density. Moreover, to gather these statistics it is not necessary to scan all rows in the table. These statistical measures can be efficiently gathered via sampling, without significantly compromising accuracy [3]. Once these statistics have been collected, it is possible for the optimizer to consider the hypothetical index for plan generation (although execution of that plan is not possible). We will discuss our approach to collecting the statistics in Section 3.2.1.

The steps in executing *Estimate Configuration* for a query are illustrated in Figure 2.

1. Create all needed hypothetical indexes in the configuration.
2. Request the optimizer to: (a) Restrict its choice of indexes to those in the given configuration. (b) Consider the table and index sizes in the database to be as adjusted by the scaling values.
3. Request the optimizer to produce the optimal execution plan for the query and gather the results: (a) the cost of the query (b) indexes (if any) used to answer the query.

These steps are repeated for each query in the workload. We now discuss implementation details of each of these steps (1) – (3) on SQL Server and provide an example that illustrates the operation of the hypothetical configuration simulation module.

1. *Sample-table* = Sample **m** pages
2. *New-Sample* = {}
3. **While not** (*convergence-of-measures*) **do**
4. *Sample-Table* = *Sample-Table* Union *New-Sample*
5. *New-sample* = Sample another m pages
6. *Convergence-of-measures* = check_for_convergence (*Sample-Table*, *New-Sample*)
7. **End do**

**Figure 3. Adaptive page-level sampling algorithm for histogram construction.**

### *3.2.1 Creation of Hypothetical Indexes*

We extend the CREATE INDEX statement in SQL with the qualifier WITH STATISTICS_ONLY [= <fraction>]. It is

optionally possible to specify the fraction of the table to be scanned when gathering sample data on columns of the index. If <fraction> is not specified, the system determines the appropriate fraction of rows to be scanned. For example:

*CREATE INDEX supplier_stats on ON Orders (supplier) WITH STATISTICS_ONLY*

This command creates a hypothetical index on the *supplier* column of the *Orders* table.

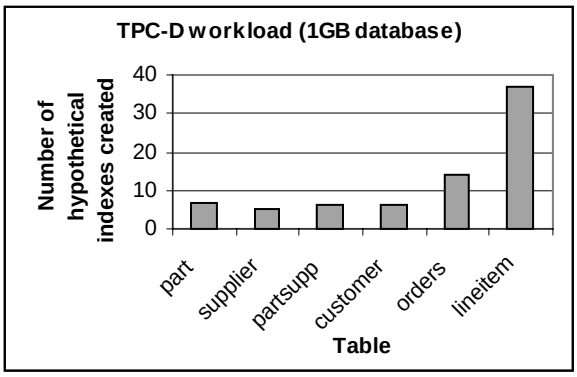


**Figure 4. Number of hypothetical indexes for each table.**

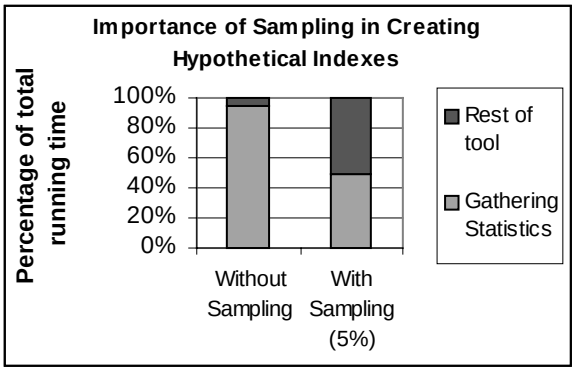


**Figure 5. Fraction of running time spent creating hypothetical indexes by index selection tool.**

We now describe the sampling strategy used for creating hypothetical indexes. We use an adaptive page-level sampling algorithm to efficiently gather statistical measures relevant to query optimization. The algorithm, shown in Figure 3, starts with a "seed" sample of **m** pages. In our current implementation we set $\mathbf{m} = \sqrt{\mathbf{n}}$ where **n** is the number of pages in the table. At any given time in the algorithm, the server maintains the sorted list of values in the *Sample-Table* and the set of statistical measures based on *Sample-Table*. In SQL Server, these statistical measures consist of (1) density of the data set and (2) Equi-Depth histograms (characterized by the step boundaries). The data in *New-Sample* is used for cross-validation purposes. In other words, it is checked if the values in *New-Sample* are divided approximately in equal numbers in each bin of the histogram (2). Our empirical results indicate that when the above test is true, the density measure also reaches convergence. If the test for the convergence fails, then the new sample is added to *Sample-Table*. This addition is done via a merge algorithm to build a new S*ample-Table* that is in sorted order. In the absence of convergence, the above step is repeated. The technical details of the algorithm and its behavior on varying data distributions are presented elsewhere [3].

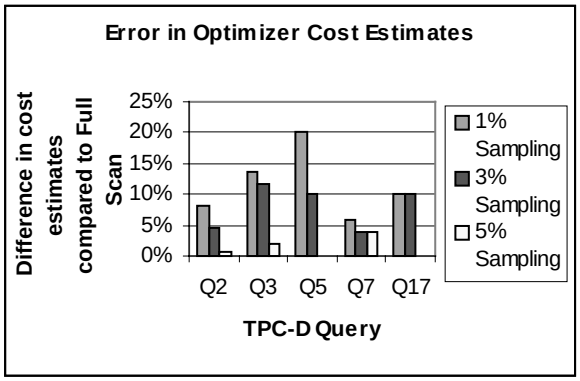


**Figure 6. Effect of sampling on optimizer cost estimates.**

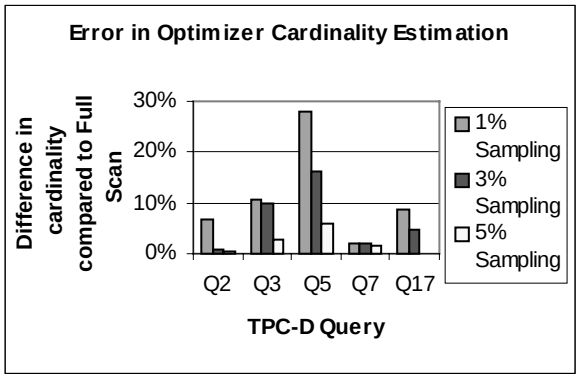


**Figure 7. Effect of sampling on estimated number of rows.**

As an example of the effectiveness of this server extension, we present the requests made by an index selection tool [4] to the HCA engine to create hypothetical indexes. Figure 4 shows the distribution of hypothetical indexes explored by the tool over tables of the TPC-D 1GB database. Figure 5 confirms our expectation that sampling can significantly reduce the cost of creating a hypothetical index. In fact, the total running time was reduced by a factor of 16. In both cases, the index selection tool recommended the *same* final set of indexes.

We now present an example to show that using sampling to create hypothetical indexes does not adversely affect the optimizer's estimates. We ran an index selection tool [4] on a workload consisting of the five most expensive queries on the TPC-D 1GB database. The adaptive sampling algorithm sampled about 5% of the data for hypothetical indexes on largest table (*lineitem*). We then ran the tool with fixed sampling rates of 1%, 3%, 5% and full table scan, and recorded the optimizer cost estimates and the estimated number of rows in each case. Figure 6 shows that the maximum error in cost estimation when using a 5% sample was

only 4% when compared to a full table scan. Figure 7 shows that similar results hold for the maximum error in the estimated number of rows.

### *3.2.2 Defining a Hypothetical Configuration*

A key issue in supporting hypothetical configurations and database scaling value is ensuring that operational queries can run concurrently on the real database while queries on a hypothetical configuration are being optimized. The optimizer obtains information on tables, indexes and their sizes from system catalogs. Therefore, a hypothetical configuration *cannot* be supported by updating system catalogs. Instead, the information for the hypothetical configuration must be conveyed to the optimizer in a connection-specific manner. This is achieved by augmenting the server with a connection-specific *HC mode* call using extensible interfaces in Microsoft SQL Server. The HC mode call takes as arguments: (1) Set of indexes corresponding to the hypothetical configuration to be used in generating a query plan. (2) The "base index" for each table in the configuration. The base index for a table is either the clustered index on the table or the heap structure for the table (if no clustered index is present). In SQL Server, the leaf node of a non-clustered B+-tree index contains the keys of the clustered index (if any) on that table. Since the plan chosen by the optimizer depends on the columns available in the index, it is necessary to indicate the base index to the optimizer. (3) Sizes of tables and indexes in the database. The HCA engine projects the size of each index in the configuration based on the database scaling value. In addition, it accounts for the fact that in SQL Server, the *size* of a non-clustered index depends on the clustered index (if any) on that table. For example, if there is a clustered index is on column A, and a non-clustered index is on column B, then the size of the index on B is proportional to the Width(column B) + Width(column A). Thus, if $I_1$ and $I_2$ are hypothetical clustered indexes, and $I_3$ is a non-clustered index, when simulating a hypothetical configuration $\{I_1, I_3\}$, the HCA engine computes a different value for the size of $I_3$ than when simulating the configuration $\{I_2, I_3\}$.

### *3.2.3 Obtaining Optimizer Estimates*

Once the hypothetical configuration is defined via the HC mode, the task of obtaining the optimizer estimates uses the traditional SQL Server API to optimize queries in the "no-execution" mode. Such a mode is supported in Microsoft SQL Server and other database systems. The results of query optimization are obtained through the Showplan interface. In addition to providing the optimizer's cost estimate, Showplan also provides the execution plan for the query, including the indexes used to answer the query.

***Example:*** Consider a database whose current configuration consists of a table T with indexes $I_1$ and $I_2$, where $I_1$ is the clustered index for T. The table T has 1 million rows. For a given workload W, we wish to simulate a hypothetical configuration $\{I_1, I_3\}$ when the table T has 10 million rows. To simulate the proposed configuration for W, the HCA engine would execute the following sequence of steps:

- Since the index on $I_3$ does not exist, the HCA engine first calls the CREATE INDEX command with the "WITH STATISTICS_ONLY" clause to create the hypothetical index $I_3$.
- The HCA engine computes the new sizes of the indexes $I_1$ and $I_3$, when the number of rows is scaled to 10,000,000 taking into account the fact that $I_1$ is the clustered index. Let these sizes be $S_1$ and $S_3$ respectively.
- HC-mode( $(I_1,I_3)$, $(1,0)$, $(S_1,S_3)$) This first argument indicates that $I_1$ and $I_3$ are to be considered by the optimizer for plan generation. The second argument indicates that $I_1$ is the "base index" for table T in the proposed configuration. The third argument passes the sizes of each index in the configuration.
- The HCA engine then executes each query in the W in the "no-execute" mode and obtains the cost and index usage information via S*howplan*.

## 3.3 Maintaining Analysis Data Tables

In Section 3.1 we described the schema of each analysis data table and discussed how the data is generated. We now address the issue of maintaining the analysis data tables in the system. We observe that once the properties of entities supported by HCA engine (queries, indexes etc.) are determined, the schema of the analysis data tables can be assumed to be fixed. Therefore, the important issues are: (a) How are these tables named? (b) Where are they stored? We now propose two alternatives to this problem.

### *3.3.1 Analysis Data in System Catalogs*

In this approach, each analysis data table is a system catalog. This solves the naming issue since system catalog names are fixed a-priori. When any of the server interfaces to simulate a hypothetical configuration is invoked, the server writes the resulting data to the appropriate system catalog. These tables can be accessed (a) directly by the user using SQL (b) via the summary analysis interfaces of the HCA engine.

### *3.3.2 Analysis Data in User Specified Tables*

In this approach, the HCA engine writes the analysis data returned by the server into temporary tables that are connection specific. When an index analysis session with the HCA engine is complete, the user is provided the option of saving the analysis data generated during the session into user specified tables. This approach requires the hypothetical configuration simulation interfaces to be augmented with a *Save* command. Subsequently, the user can name the saved tables to the HCA engine and perform summary analysis on the data, or can directly post arbitrary SQL queries against these tables. In our current implementation, we have adopted this approach.

## 4. Summary Analysis

The ability to simulate hypothetical configurations provides the foundation for summary analysis. In this section, we show how the AutoAdmin index analysis utility builds on that infrastructure to provide sophisticated analyses of proposed changes. Figures 8 through 10 provide some examples of summary analysis that a

database administrator finds useful. Figure 8 shows a breakdown of the workload by the type of queries. Figure 9 "drills-down" on the queries of type selection and provides a breakdown of selection conditions in queries by table. Such summary analysis provides the DBA with a better grasp of the workload that the system is facing. Figure 10 is an example where the DBA can view the relative frequencies of usage of indexes in the current configuration. The DBA may use this information to identify indexes that are rarely used and perhaps are good candidates for dropping. Indeed, one can think of many useful ways to analyze the information gathered during hypothetical configuration simulation (Tables 1-4 in Section 3.1). (We refer to this information as analysis data).

One option for producing summary statistics on analysis data is to allow the DBA as well as other tools to directly use the SQL interface to query the information collected during the process of hypothetical configuration simulation. Unfortunately, the approach of generating SQL queries is a relatively low-level interface for performing summary analysis, since it shifts the burden of analysis to the consumer of the information. The other option is to provide a set of "canned" queries that support a set of predetermined summary analyses. However useful, the canned queries do not provide an extensible framework for generating new summary statistics from the available analysis data. What is needed is an interface that retains the flexibility of formulating ad-hoc requests for summary analysis, but without the overhead of manually generating complex SQL queries over the analysis data. In the next section, we describe a query-like interface that the HCA engine supports to fulfil this need. The interface that we describe was used in AutoAdmin for a principled design of a powerful user interface that can invoke the HCA summary analysis interfaces in a rich way.

We begin in Section 4.1 with a description of a generic summary analysis interface that captures the structure of questions that can be posed against the analysis data. In Section 4.2, we present the summary analysis interfaces and describe the properties of query, index and cost-usage analysis objects that are used to formulate queries. Section 4.3 provides examples of interesting queries that can be posed using the analysis interfaces, and gives a flavor of user interfaces. We discuss the implementation of the summary analysis interface in Section 4.4. In Section 4.5 we present a sample session that a database administrator might have with the index analysis utility.

## 4.1 Conceptual Model for Summary Analysis

Our model for summary analysis recognizes that the three foundational objects for analysis are:

- *Workload Analysis*, which consists of *queries* and their structural properties
- *Configuration Analysis*, which consists of *indexes* and their structural properties
- *Cost and Index Usage Analysis*, which represent relationship properties between a query and a configuration

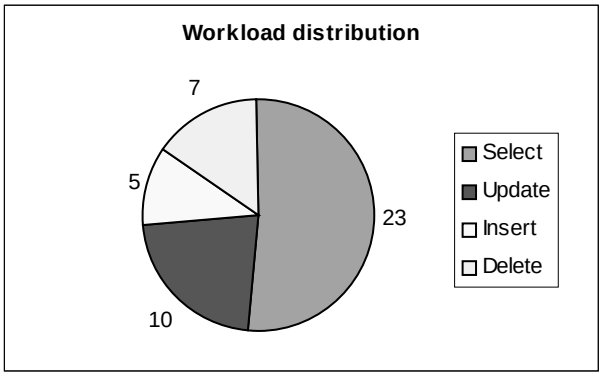


**Figure 8. Distribution of workload by SQL Type.**

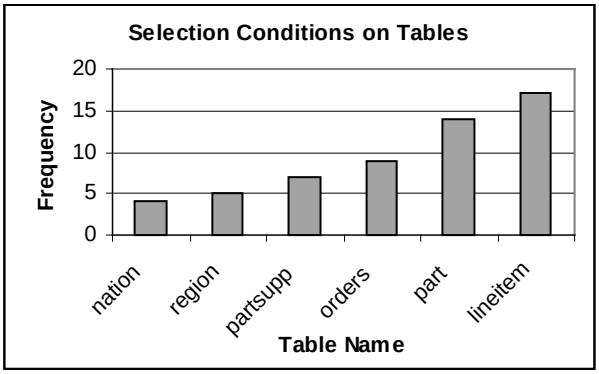


**Figure 9. Distribution of conditions over tables.**

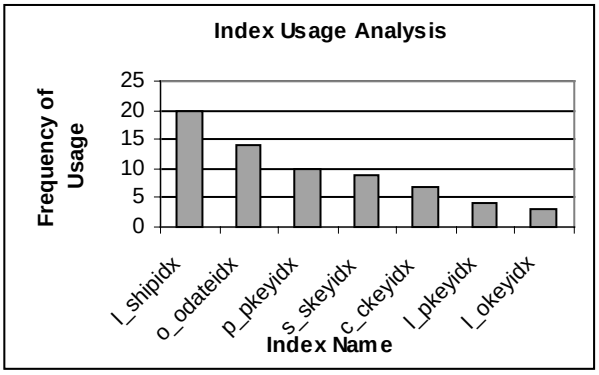


**Figure 10. Analyzing the frequency of index usage for a given configuration and workload.**

While the specific summary questions that are of interest relate to one of the above three objects, the HCA summary analysis interface provides a generic querying model. Such a generic analysis interface has the advantage that as we extend the framework into more complex physical database design, it exploits the common thread that runs through each kind of analysis.

### *4.1.1 Objects and Properties*

Any summary analysis is over a *set of objects*. For example, for workload analysis, it is a set of queries. In configuration analysis,

the objects in question are indexes. For cost and index-usage analysis, the objects are relationship objects that capture the interaction between a specific configuration and queries in a workload. In each case, the set of objects for summary analysis can be implicitly identified. For workload analysis, the workload name uniquely identifies the queries in the workload. Likewise the configuration name identifies the indexes in the hypothetical configuration being analyzed. For Cost and Index Usage Analysis, the combination of the workload name and configuration name uniquely identifies the set of objects to be analyzed.

Each object has *properties* associated with it. These properties can be classified as follows: (a) *Properties with an atomic value*, e.g. the number of tables in a query. A special case of atomic value is Boolean (e.g., whether or not an index is clustered). (b) *Properties with a list (or set) value*, e.g. the *list* of columns in an index or the *set* of tables referenced in a query. For each of the three types of objects we now list the properties of that object that are gathered by the HCA engine in AutoAdmin.

#### *4.1.1.1 Properties of Queries*

For workload analysis, the object in consideration is a query. Below, we list properties that are currently parsed by the HCA engine for a query. The properties with *atomic value* are query type (Insert/Delete/Update/Select), whether the query has a Group By clause (Boolean), whether the query has an Order By clause (Boolean), whether the query has nested subqueries (Boolean). The *properties with list values* are tables referenced in query, required columns from each table, columns on which selection conditions exist, columns on which join conditions exist, Equi-join conditions. We note that above list of properties can be easily extended by collecting additional interesting parsed query information.

***Example:*** Consider the query:
SELECT R.a, S.c FROM R, S WHERE R.a = S.c AND R.b = $v_1$
The properties are: (1) SQL Type: Select. (2) The required tables are {R, S}. (3) The required columns are: {R.a, R.b, S.c} (4) Columns on which selection conditions exist: {R.b} (5) Equi-join conditions: {(R.a = S.c)}

#### *4.1.1.2 Properties of Indexes*

For configuration analysis, the objects of interest are indexes. The properties of indexes with *atomic value* are table on which index is built, width of index (number of columns), storage space, time of creation, whether or not the index is clustered (Boolean). The property with list values is the list of columns in the index (in major to minor order).

***Example***: Consider a clustered index $I_1$ on columns ($C_1$,$C_2$) of table T. The properties of $I_1$ are (1) Table = T (2) Width = 2 (3) Clustered = True (4) list of columns = {$C_1$, $C_2$}. The storage space and time of creation properties would also be filled appropriately.

#### *4.1.1.3 Properties of Relationship Object of Query and Configuration*

For the relationship object of a query and a configuration, the property with *atomic value* is cost of the query for the configuration. The property with *list value* is the list of indexes in the configuration used to answer the query. We note that the properties of the relationship object between a query and a configuration can be augmented with other information about the query execution plan (e.g. operators used in the plan).

### *4.1.2 Measures and Aggregate Measures*

Objects and properties form the fundamental primitives for summary analysis. However, derived measures are useful for posing queries against analysis data. With each property of an object, we can derive one or more numerical *measures*. For an atomic property this measure could be the value of the property itself (e.g. storage for an index) or a user defined function of the value. For a list or set property, the measure may be the *count* of the number of elements in the list or set, e.g., the number of tables referenced in a query. A measure for a list/set property may be derived also by applying one of the *aggregate functions* (e.g., SUM, AVERAGE) on the values in the list/set. For example, the aggregate functions may be used on measures to obtain a derived measure for a set of objects. Thus the specification of a numerical measure consists of (a) a property name (b) an expression that derives a numerical measure from the property value. For list/set valued property p, the aggregate measure is $f$ (p), where $f$ is an aggregation function. In our current implementation, $f$ can be *Count*, *Min*, *Max*, *Sum*, *Average*.

We lift the notion of a numerical measure to a list/set of objects to derive an *aggregate measure* in the obvious way. Given a measure **m** for each object, we derive a corresponding measure $f$(**m**) over a set of objects by applying an aggregate function $f$. For example, given a workload (a set of queries), we can compute the average number of tables referenced per query in the workload.

## 4.2 Summary Analysis Interfaces

Although simple, the abstractions of objects, properties and measures provide an approach to the problem of defining a convenient and yet powerful query-like interface for summary analysis. The generic analysis interface that we present in this section is geared towards supporting the following paradigm of analyzing information from hypothetical configuration simulation:

1. Determine a **class of analysis** (workload analysis, configuration analysis, cost/index usage analysis)
2. Specify necessary information to uniquely **identify** the set of **objects** of analysis. E.g., specify workload name to identify associated queries.
3. **Filter** a subset of objects (based on their properties) to focus on objects of interest, e.g., consider queries that reference a table “Orders”. There may be successive filtering operation, supported through “drill-down” using user interfaces, e.g., consider queries that reference the column “Supplier” in “Orders”.

4. **Partition** the filtered objects in a set of classes by a measure. For example, the queries that survive the filter in (3), maybe partitioned by their query type (Insert/Delete/Update/Select). The partitions need not be disjoint.
5. **Rank or Summarize** the objects. Ranking is achieved by associating a measure with each object (e.g. for a query, the number of tables referenced in the query) and using the measure to order the objects. Thus the interface supports picking the top **k** objects ranked by the measure. The interface also supports summarizing the objects based on an aggregate measure (e.g., average number of tables referenced in queries). If no partitioning is mentioned, then the ranking and summarization is done for all objects that qualify the filter. Otherwise, it is done for each partition but all partitions share the same ranking/summarization criteria.

The Filter and Partition steps described above are optional. Thus the simplest form of analysis is to rank objects by a given measure or summarize all objects through an aggregation function. We now present the "query-like" summary analysis interface and explain the syntax and semantics of this interface using a series of examples to highlight each aspect. As with the interfaces for hypothetical configuration simulation, our focus is on the functionality enabled by this interface rather than the syntax.

***ANALYZE[WORKLOAD|CONFIGURATION|COST-USAGE]***
***WITH*** <*parameter-list*>
***[TOP*** <*number*>| ***SUMMARIZE USING*** <*aggregation-function*>***]*** ***BY*** <*measure*>
***WHERE*** <*filter-expression* >
*{****PARTITION BY*** <*partition-parameter*> ***IN*** <*number*> ***STEPS****}*

### *4.3.1 Format of Output*

The format of the results produced depends on whether ranking or summarize output is desired. If the query uses SUMMARIZE, then the output consists of one row for each partition. Each row has two columns, one has the value of the partitioning parameter and the other has the summarized value for that partition. For example, **Q1** counts the number of queries in the workload of each type (Select/Update/Insert/Delete).

**Q1**: ANALYZE WORKLOAD WITH Workload_A
SUMMARIZE USING Count
PARTITION BY Query_Type

If the partitioning clause is omitted, then the output is a scalar, representing the summarized value for all objects selected. For example, **Q2** counts the total number of indexes in a configuration.

**Q2**: ANALYZE CONFIGURATION WITH *Current_Config*
SUMMARIZE USING *Count*

When the ranking option is used (i.e., TOP is used), the output format has three columns: the first column has the partitioning attribute, the second column has the object itself (e.g., the query string), and the third column specifies the rank of the object within the partition. If no partitioning clause is present, then there will be altogether <number> of 3-column tuples. For example, **Q3** returns the 20 most expensive queries in Wkld_B for the current configuration. Each tuple in the output for **Q3** is of the form (Workload-name, Query, Rank).

**Q3**: ANALYZE COST_USAGE WITH *Wkld_B, Current_Config*
TOP 20 BY *Cost*

### *4.3.2 Measures*

As described earlier, measures can be useful for posing interesting queries on the analysis data. Measures can be specified in the BY <*measure*> clause and the PARTITION BY <*partition-parameter*> clause. A measure has one of the following two forms: (a) an atomic property of an object. For example **Q4** returns the top 3 indexes in *New_Config* ranked by storage. (b) <*aggregation-function*>(<*list/set property*>). **Q5** counts the number of queries in *Wrkld_A* where a given number of tables are referenced. In our current implementation, we support Count, Max, Min, Sum, and Average for <*aggregation-function*>. We note that when *Count* is used in the SUMMARIZE USING clause, the <*measure*> specification is not required (e.g. **Q1**, **Q2**, **Q5**).

**Q4**: ANALYZE CONFIGURATION *New_Config*
TOP 3 BY *Storage*

**Q5**: ANALYZE WORKLOAD WITH *Wkld_A*
SUMMARIZE USING *Count*
PARTITION BY *Count* (*Tables*)

### *4.3.3 Filter Expressions*

The syntax of <*filter-expression*> is a Boolean expression where base predicates are composed using Boolean connectors. For atomic properties, the base predicates have the form: <*property*> <*operator*> <*value*>. The operator can be any comparison operator, e.g., *storage > 50*. However, for Boolean properties only equality check is legal. For example, **Q6** counts the number of two-column, non-clustered indexes in *Current_Config*.

**Q6**: ANALYZE CONFIGURATION WITH Current_Config
SUMMARIZE USING *Count*
WHERE (*Num-Columns* = 2) AND (*Is-Clustered* = FALSE)

For set-valued properties base predicates have one of the following three forms:

- <*set property* > *[ SUBSET-OF| SUPERSET-OF |= ]* <*set*>. For example, **Q7** returns the 10 most expensive queries in Wkld_A for the current configuration such that the query references at least the tables *part* and *supplier*.
- *f (*<*set property*>*)* <*operator*> <*value*>, where f is an aggregation function and <operator> is any comparison operator, e.g., the following filter is satisfied for queries that reference at least two tables: *Count(Tables) > 1.*

- For list valued properties, all base predicates as for set valued predicates apply by interpreting the list as the corresponding set. However, in addition, the following predicate based on prefix matching is allowed:
  *<list property> [ SUBLIST-OF| SUPERLIST-OF|= ] <list >*. For example, **Q8** counts the number of indexes in Config_A that have part.size as their leading column. Here, *Columns* is the list property of an index that contains the columns in the index.

**Q7**: ANALYZE COST_USAGE WITH *Wkld_A, Current_Config*
TOP 10 BY *Cost*
WHERE *Tables* SUPERSET-OF {part, supplier}

**Q8**: ANALYZE CONFIGURATION WITH *Wrkld_A, Config_A*
SUMMARIZE USING *Count*
WHERE *Columns* SUPERLIST-OF (*part.size*)

### *4.3.4 Partitioning the Results*

The objects being analyzed may be partitioned either by a property (need not be numeric, e.g. **Q1**) or by a numeric measure (e.g. **Q5**). An important special case is when *<partition-parameter>* is the name of a list or a set valued property. In such a case, there is a separate partition for each distinct value of the list or set. A set valued object S belongs to the partition for d if and only d is a member of the set S. For example, in **Q9**, a query belongs to the partition of each table that is referenced in that query. **Q9** computes the average number of indexes used for queries on each table (but eliminating join queries). Finally, when the partitioning domain is numeric, the number of steps allows partitions to be coalesced into fewer steps.

**Q9**: ANALYZE COST_USAGE WITH *Workload-A, Current*
SUMMARIZE USING *Average* BY *Count* (*Indexes-Used*)
WHERE *Count* (*Join-Columns*) = 0
PARTITION BY *Tables*

### *4.3.5 Specifying Objects for Analysis*

In each of the examples **Q1**-**Q9**, depending on the class of analysis, the *<parameter-list>* can contain a (a) workload name (b) configuration name or (c) workload name and configuration name. In general, it is possible to specify multiple workloads and configurations in the *<parameter-list>*, making it possible to *compare* workloads or configurations. We do not discuss details of possible formats of *<parameter-list>* due to lack of space. However, to illustrate the idea, we present **Q10**, which compares the cost of two configurations for queries that reference table Orders.

**Q10**: ANALYZE COST-USAGE WITH *Workload-A, (Current_Config, Proposed_Config)*
SUMMARIZE USING *Sum* BY *cost*
WHERE *Tables* SUPERSET-OF *Order*

## 4.3 Examples of Summary Analysis and User Interfaces

The summary analysis interface is expressive and can be used to perform a rich set of analyses. We now provide examples of each *class* of summary analysis and the user interfaces that make it easy for a database administrator to visualize the results of summary analysis. All examples presented below can be expressed using the summary analysis interface.

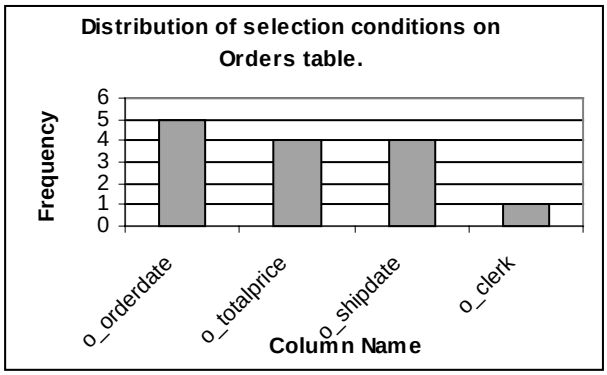


**Figure 11. Distribution of selection conditions on a given table.**

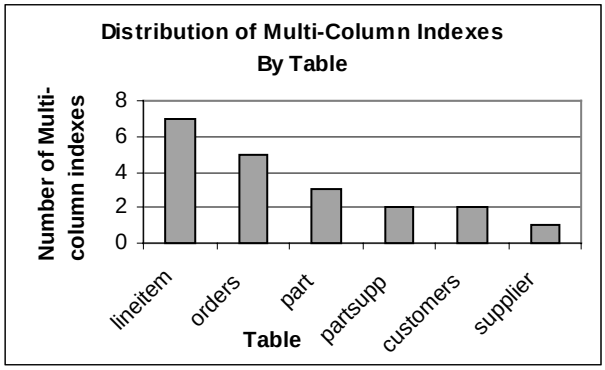


**Figure 12. Analyzing distribution of multi-column indexes over tables.**

### *4.3.1 Workload Analysis*

(1) An example of application of partitioning is to count the number of queries by SQL Type (see Figure 8).
(2) List the top 5 tables on which most queries are posted.
(3) Comparing summary statistics from two workloads.
(4) Drilling-down at a table level to find which columns of the table that have most conditions posted on them (see Figure 11).

### *4.3.2 Configuration Analysis*

(1) An example of application of partitioning is to count the number of indexes for each table.
(2) List the top 6 tables ranked by the count of the multi-column indexes on those tables. (See Figure 12)

### *4.3.3 Cost-Usage Analysis*

(1) Analyzing the frequency of usage of each index in the configuration for the workload (Figure 10).
(2) Analyzing the cost of each query in the workload for the proposed configurations (relative to the current configuration). (Figure 13).
(3) Comparing the cost of two configurations for a given workload by SQL Type. (Figure 14).

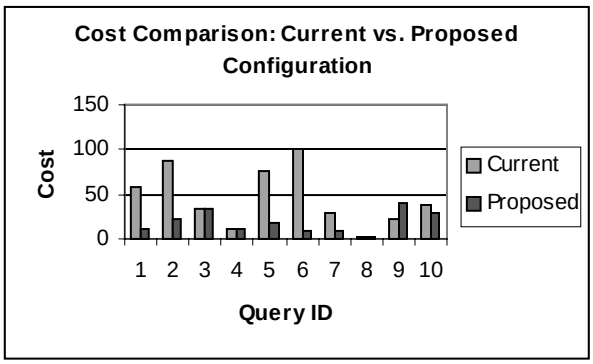


**Figure 13. Comparing cost of the 10 most expensive queries for two configurations.**

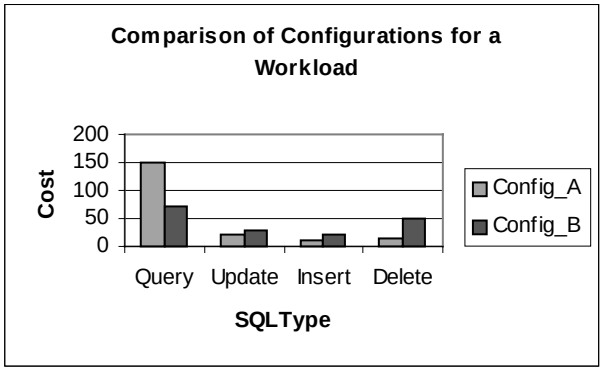


**Figure 14. Comparing cost of workload for two configurations by SQL Type of query.**

## 4.4 Implementation of Summary Analysis Interfaces

In this section, we briefly describe the issues involved in implementing the summary analysis interface. As described in Section 3.3, the data generated during hypothetical configuration simulation are stored in tables at the server. When a query is posed using the summary analysis interface, the HCA engine maps the query into an equivalent SQL query over the analysis data tables. In addition, the HCA engine may further process the results of the SQL query before completing the analysis (e.g. bucketizing results). The fact that the summary analysis interface resembles SQL makes it easier for the HCA engine to map the input into a SQL query. In addition, the implementation of the summary analysis engine also exploits the ability to compose operations using table expressions in the FROM clause of a SQL query.

The kind of analysis (Workload, Configuration, or Cost-Usage) specified in the summary analysis query determines the superset of analysis data tables that need to be joined to answer the query. For example, for Workload analysis, we only need to access the workload information tables (see Table 1). The HCA engine then generates a table expression ($T_1$) that joins the required analysis data tables, only retrieving objects that are specified in the <*parameter-list*>. In addition, the <*filter-expression*> (if any) specified in the query is included in $T_1$. The HCA engine then generates a table expression $T_2$ that partitions $T_1$ using the <*partition-parameter*> (if any), using the GROUP BY construct, and the <*aggregation-function*> is applied to the attribute specified by <*measure*>. If instead, the query requests the TOP <*number*> of rows BY <*measure*>, then $T_2$ is generated by applying an ORDER BY clause on the <*measure*> attribute of $T_1$. A cursor is opened for $T_2$ to return the first <*number*> rows.

## 4.5 Example of a Session

In this section we provide an example of a typical session by a DBA using the impact analysis utility.

### *4.5.1. Analyzing the workload*

As mentioned earlier, in AutoAdmin, evaluation of the current or a proposed design is always done with respect to a workload. Therefore, DBA begins by specifying a workload for the session. This may be a log of queries that have run against the system over the past week. The DBA tries to understand the workload mix and asks for a breakdown of the queries by SQL Type. The result of this analysis looks like Figure 8. The DBA may then decide to focus on the most expensive queries in the workload for the existing configuration, by requesting the top 25 queries ordered by cost. To decide which tables are good candidates for indexing, the DBA may wish to see the distribution of conditions in queries on tables (Figure 9). Having picked a table that has many conditions on it, the DBA may decide to further "drill-down" and look at the distribution of conditions in queries over columns of that table (Figure 11). This gives a good idea of which columns on the table are likely candidates for indexes. The DBA finds that columns A and B of table $T_2$ look promising.

### *4.5.2 Analyzing the current configuration*

The DBA may then wish to see if indexes on A and B of table $T_2$ already exist in the current configuration. He does this by requesting to see all indexes on $T_2$ in the current configuration ordered by their storage requirement. In this case, there is an index on A, but no index on B. So the DBA decides to explore hypothetical configuration scenarios that include an index on B.

### *4.5.3 Exploring "what-if" scenarios*

The DBA then decides to explore two "what-if" scenarios and evaluate each relative to the current configuration. He first proposes a hypothetical configuration consisting of the current configuration with an additional non-clustered single-column index on column B of $T_2$. For this configuration he compares the cost of the workload with the cost of the workload in the current

configuration. Adding a single-column index on B produces a 5% improvement in total cost of the workload (Figure 13). By studying index usage in the proposed configuration (Figure 10), the DBA sees that the new index was used in three queries. Not being impressed with the improvement in performance, the DBA decides to explore a different hypothetical configuration. This time he proposes a two-column index (B, A) on table $T_2$ in addition to the current configuration. Once again, the DBA compares the cost of this configuration with the current configuration and sees an 18% improvement for the workload. He then looks at the top five queries in the workload that are affected by adding the index and notices that two of the most expensive queries under the current configuration were positively affected and that there were no queries that were negatively affected. He then decides to build the two-column index (B,A) and schedules the index to be built at midnight.

## 5. Conclusion

In this paper we have shown how an index analysis utility can help the DBA of an enterprise-class database to select indexes for the database. We have presented the interfaces supported by a hypothetical configuration analysis engine and shown how this functionality can be used to conduct interesting and powerful analysis studies. We have described the implementation of the hypothetical configuration analysis engine for Microsoft SQL Server 7.0, including the necessary server extensions. In the future, we will extend our current framework to incorporate other aspects of physical database design.

## 6. Acknowledgments

We would like to acknowledge Nigel Ellis from the SQL Server relational engine group for helping incorporate our extensions into the server.

## 7. References

[1] AutoAdmin Project, Database Group, Microsoft Research, http://www.research.microsoft.com/db.

[2] Choenni S., Blanken H. M., Chang T., “Index Selection in Relational Databases”, Proceedings of 5th IEEE ICCI 1993.

[3] Chaudhuri, S., Motwani, R., Narasayya, V., “Random Sampling for Histogram Construction: How Much Is Enough?”. Proceedings of ACM SIGMOD ’98.

[4] Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. ". Proceedings of the 23rd VLDB Conference, Greece, 1997.

[5] Frank M., Omiecinski E., Navathe S., “Adaptive and Automative Index Selection in RDBMS”, Proceedings of EDBT 92.

[6] Finkelstein S, Schkolnick M, Tiberio P.“Physical Database Design for Relational Databases”, ACM TODS, Mar 1988.

[7] Gupta H., Harinarayan V., Rajaramana A., Ullman J.D., “Index Selection for OLAP”, Proceedings of ICDE97.

[8] Harinarayan V., Rajaramana A., Ullman J.D., “Implementing Data Cubes Efficiently”, Proceedings of ACM SIGMOD 96.

[9] Labio W.J., Quass D., Adelberg B., “Physical Database Design for Data Warehouses”, Proceedings of ICDE97.

[10] Olken F., “Random Sampling in Databases”, Technical Report, 1993.

[11] Rozen S., Shasha D. “A Framework for Automating Physical Database Design”, Proceedings of VLDB 1991.

[12] Stonebraker M., Hypothetical Data Bases as Views. Proceedings of ACM SIGMOD 1981.