An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server

Surajit Chaudhuri **Vivek Narasayya**
Microsoft Research, One Microsoft Way, Redmond, WA, 98052.
{surajitc, viveknar}@microsoft.com

Abstract

In this paper we describe novel techniques that make it possible to build an industrial-strength tool for automating the choice of indexes in the physical design of a SQL database. The tool takes as input a workload of SQL queries, and suggests a set of suitable indexes. We ensure that the indexes chosen are effective in reducing the cost of the workload by keeping the index selection tool and the query optimizer "in step". The number of index sets that must be evaluated to find the optimal configuration is very large. We reduce the complexity of this problem using three techniques. First, we remove a large number of spurious indexes from consideration by taking into account both query syntax and cost information. Second, we introduce optimizations that make it possible to cheaply evaluate the "goodness" of an index set. Third, we describe an iterative approach to handle the complexity arising from multi-column indexes. The tool has been implemented on Microsoft SQL Server 7.0. We performed extensive experiments over a range of workloads, including TPC-D. The results indicate that the tool is efficient and its choices are close to optimal.

1. Introduction

Enterprise-class databases require database administrators who are responsible for performance tuning. With large-scale deployment of databases, minimizing database administration function becomes important. We started the *AutoAdmin* research project at Microsoft to investigate new techniques to self-tune and self-administer database systems to achieve performance competitive with that of systems that are cared for by database administrators. One important task of a database administrator is selecting a physical database design appropriate for the workload of the system. An important part of physical database design is selecting indexes. In this paper, we present the novel technique that we have developed in the AutoAdmin project to automate the task of selecting a set of indexes.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 23rd VLDB Conference Athens, Greece, 1997

The index selection problem has been studied since the early 70's and the importance of the problem is well recognized. Despite a long history of work in this area, there are few research prototypes and commercial products that are widely deployed. Three basic approaches have been taken to the index selection problem. The "textbook solutions" [CG93] take semantic information such as uniqueness, reference constraints and rudimentary statistics ("small" vs. "big" tables) and produce a database design. Such designs may perform poorly because they ignore valuable workload information. The second class of tools adopt an expert system like approach, where the knowledge of "good" designs are encoded as rules and are used to come up with a design. Such tools can take into account workload information but suffer from being disconnected from the query optimizer [HE91]. This has adverse ramifications for two reasons. First, a selection of indexes is only as good as the optimizer that uses it. In other words, if the optimizer does not consider a particular index for a query, then its presence in the database does not benefit that query. Second, these tools operate on their own model of the query optimizer's index usage. While making an accurate model is itself hard, ensuring consistency between the tool and an evolving optimizer is a software-engineering nightmare.

In the third approach, the index selection tool uses the optimizer's cost estimates to compare *goodness* of alternative hypothetical designs. This approach avoids asynchrony between the index selection tool and the query optimizer. This is the approach we have adopted (also adopted in [FST88]).

1.1 Our Contributions

The AutoAdmin index selection tool that we describe in this paper is significant in several ways. First, we have recognized that index selection is more than just a difficult search problem. Therefore, we have taken an end to end systems approach in identifying the building blocks of an index selection tool, when the tool uses the optimizer's cost estimates. Implementing the index selection tool has given us a good understanding of the architectural and system level issues. Second, we use a novel search technique that filters out spurious indexes in an early stage and exploits characteristics of the relational query engine to reduce the cost of selecting indexes. Another novelty of our search technique is exploring the space of alternative indexes in an iterative way such that more complex alternatives (e.g., multi-column indexes) are generated from "good" simpler alternatives (e.g., single-column indexes). Finally, since we have fully implemented our tool on Microsoft SQL Server 7.0 (with necessary server changes), we are able to present extensive experimental results. These results demonstrate

that our proposed techniques result in an improvement by a factor of 4 to 10 in the search time, without significantly sacrificing the quality of the result. The experimental results complement our intuition on the effectiveness of the strategies proposed in the paper.

2. Overview of Our Approach

2.1 Problem Statement

Our goal is to pick a set of indexes that is suitable for a given database and workload. An index can be *single-column or multi-column.* An index may be either *clustered or non-clustered,* although over any single table, there can be at most one clustered index. A workload consists of *a set of SQL data manipulation statements,* i.e., Select, Insert, Delete and Update. In this paper we use the term **configuration** to mean a set of indexes. We define the *size* of a configuration as the number of indexes in the configuration.

To pick a configuration, we must be able to compare the relative *goodness* of any two configurations. Given a configuration and a workload, we use the sum of the optimizer estimated costs for all the SQL statements in the workload as the metric of goodness. We refer to this metric as the *total cost* of a configuration. Given a workload, a configuration that has the least value of total cost is called the *optimal configuration.* The goal of an index selection tool is to pick a configuration that is optimal or as close to optimal as possible. The index selection process is subject to constraints, e.g., an upper bound on the number of indexes or storage space. In this paper, we describe our technique to deal with the problem of picking a configuration subject to an upper bound on the number of indexes.

2.2 Architecture of the Index Selection Tool

An overview of the architecture of our index selection tool is presented in Figure 1. The dotted line in the figure denotes the process boundary between the tool and SQL Server. The index selection tool takes as input a workload on a specified database. The tool has a basic search algorithm, which iteratively picks more complex index structures. In the first iteration, the tool considers only single-column indexes; in the second iteration it considers single and two-column indexes, and so on. The search algorithm derives its efficiency from each of its three modules. First, the *candidate index selection* module (Section 4) eliminates from further consideration, a large number of indexes that provide little or no benefit for any query in the workload. The *configuration enumeration* module (Section 5) intelligently searches the space of subsets of candidate indexes and picks a configuration with low total cost. The *multi-column index generation* module selects multi-column indexes to be considered in the next iteration along with the indexes chosen by the configuration enumeration module. The tool has been designed so that any of these modules can be replaced with a newer or improved version, without having to change any of the remaining modules.

Any index selection tool that bases its choices on optimizer cost estimates requires the ability to evaluate the workload for a given configuration. This service is provided in our architecture by the *cost-evaluation* module (Section 3). This module maintains the cost table information shown in Figure 2. The cost-evaluation module performs optimizations so that it needs to invoke the optimizer only for a selected subset of the configurations. Since an index considered by the search algorithm may not be present in the database, the tool also requires the ability to "simulate" the presence of the index in the database for the optimizer. The *"what-if" index creation* module provides the tool with an interface for specifying this requirement. We have modified SQL Server to support the creation and loading of a "what-if" index. A detailed discussion of the required changes to the server is beyond the scope of this paper.

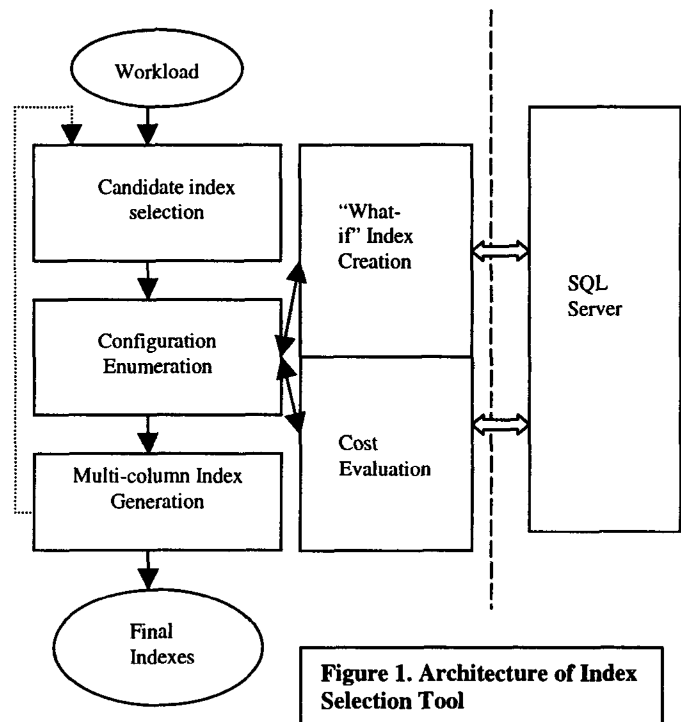

Figure 1. Architecture of Index Selection Tool

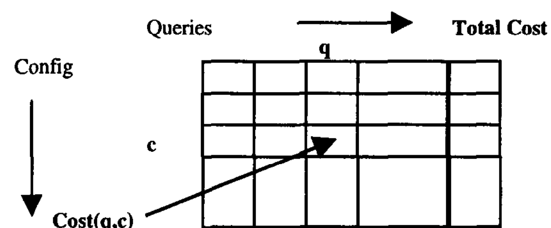

Figure 2. Table maintained by Cost Evaluation module.

There are three important measures of the *efficiency* of an index selection tool: (1) The *number of indexes* considered by the tool for a given workload. (2) The *number of configurations* that are enumerated by the index selection tool. (3) The *number of optimizer invocations* necessary to evaluate the total cost for each enumerated configuration. Any solution to the index selection problem has to be judged by its efficiency as well as by its *quality* (how close the solution is to the optimal). As we will show in this paper, an index selection tool can be made efficient through well-

designed algorithms, without significantly reducing the quality.

2.3 Preliminaries

Indexable columns identify columns in a query that are potentially useful for indexing. We present a simple definition of indexable columns here. Indexable columns form the basis for defining *admissible indexes of a workload* which identify the set of all potentially useful indexes for a workload and provide the starting point for the index selection tool which picks a subset of these indexes. These definitions can be easily modified without affecting the rest of the index selection tool. In the following definitions, we refer to SQL DML statement and "Query" interchangeably.

Definitions:

(i) *An indexable column for a query* in the workload is a column R.a such that there is a condition of the form *R.a operator Expression* in the WHERE clause. The operator must be among {=, <, >, <=, >=, *BETWEEN, IN}*.
Columns in GROUP BY and ORDER BY clauses are also considered indexable. For an Update query, the updated columns of the table are considered indexable.
(ii) *An admissible index for a query* is an index that is on one or more (in case of a multi-column index) indexable columns of the query.
(iii) *An admissible index for a workload* is an index that is an admissible index for one or more queries in the workload.

Example 1: Indexable Columns of a Query

Consider the following query $\mathbf{Q_1}$:
SELECT * FROM onektup, tenktup1
WHERE (onektup.unique1 = tenktup1.unique1)
AND (tenktup1.unique2 between 0 and 1000)
From the above definition, it follows that the indexable columns of $\mathbf{Q_1}$ are {onektup.unique1, tenktup1.unique1, tenktup1.unique2}.

3. Cost Evaluation

In the naïve approach to evaluating a configuration, the cost-evaluator asks the optimizer for a cost estimate for each query in the workload. Thus, if there are M configurations and Q queries in the workload, such estimation requires asking the SQL Server to optimize M*Q queries. Invoking the optimizer many times can be expensive (despite batching the invocations), since it requires communication across process boundaries between the tool and the server. In this section, we present techniques that result in significantly reducing the number of optimizer calls through the concept of *atomic configurations* (cf. [FST 88]).

Intuitively, a configuration C is *atomic* for a workload if for some query in the workload there is a possible execution of the query by the query engine that uses *all* indexes in C. In Section 3.1, we show that if a configuration is not atomic for the workload, then we can derive the cost of queries for that configuration accurately. Therefore, instead of having to evaluate all M configurations (over the space of admissible indexes of the workload), it is sufficient to *ask the optimizer to evaluate* only M' configurations among M, as long as all atomic configurations are included in M'. Identifying *which* configurations are to be included among M' is crucial for the accuracy and efficiency of the tool, and we present two techniques for this in Section 3.2.

Another way to reduce optimizer invocation is to exploit the fact that not every atomic configuration needs to be evaluated for every query in the workload. In particular, we may be able to estimate the cost of a query Q for an atomic configuration C by using the cost of the query for a "simpler" configuration C'. Section 3.3 describes this optimization.

3.1 Deriving Cost of a Configuration from Atomic Configurations

Let us assume that C is a configuration that is *not* atomic and Q is a Select/Update query in the workload. Consider all atomic configurations C_i of Q that are subsets of C. One of these configurations must be chosen by the optimizer while optimizing Q. Therefore, a well-behaved optimizer will choose the atomic configuration from the above set that has the minimal cost. Therefore, we can derive Cost (Q, C) = Min_i {Cost(Q, C_i)} *without invoking the optimizer*. Furthermore, since for a Select query, inclusion of an index in a configuration can only reduce cost, it will suffice to take the minimum cost over the *largest* atomic configurations of Q that are subsets of C.

If Q is an Insert/Delete Query, the analysis is more complex. The cost of an Insert/Delete query for a non-atomic configuration C, can be divided in three components. (a) Cost of selection (b) Cost of updating the table and the indexes that may be used for selection and (c) Cost for updating indexes that *do not* affect the selection cost. We note that the cost for updating each index in (c) is independent of each other and can be assumed to be independent of the plan chosen for (a) and (b). Therefore, the optimizer will pick a plan that minimizes costs for (a) and (b). As in a Select/Update query, we can derive T, the minimum of the costs over all atomic configurations of Q that are subsets of C to reflect the components (a) and (b) of Cost (Q, C) . To get the cost of updating an index I that is in (c), we look up (Cost (Q, {I}) – Cost (Q, { }). Thus we can estimate the total cost by: T + Σ_j (Cost (Q, {I_j}) – Cost (Q, { })) without invoking the optimizer for C.

3.2 Identifying Atomic Configurations

In this section we address the important issue of identifying atomic configurations for a workload. The total number of atomic configurations can be very large. In particular, for multi-table queries the number of atomic configurations is exponential in the number of tables. The next two sections describe techniques for heuristically reducing the number of atomic configurations that need to be considered for a workload without significantly sacrificing the accuracy of cost estimation.

3.2.1 Query Processor Based Restrictions

The characteristics of the query processor can influence what constitutes an atomic configuration. We

exploit two widely applicable considerations to restrict the set of configurations we consider atomic.

- In any given execution of a query, the query processor will only use no more than **j** indexes per table[1] (for some integer **j**). This restriction reflects the pragmatic that no more than a small number of indexes[2] may be intersected to identify tuples of one relation. For query engines that do not support index intersection, we can strengthen this condition by requiring that no more than one index per table appear in any atomic configuration.
- In any given execution of a multi-table query, indexes from at most **t** tables need to be considered for an atomic configuration. The intuition is that indexes used in the first few joins of a query have the most impact on the cost of the query.

In designing our cost evaluation module, we found that values of **j** = 2 and **t** = 2 provide good quality solutions while dramatically reducing the number of atomic configurations for complex workloads. We refer to these as *single-join atomic configurations.* It is the search strategy, discussed in Section 5, which determines how and when these atomic configurations are evaluated. In particular, they may be evaluated "on-demand", i.e., when we are asked to evaluate Cost (Q, C), we evaluate all atomic configurations that are subsets of C, which have not yet been evaluated.

It is possible for search strategies to use different values of **j** and **t** during enumeration. In fact, we have explored a two-tier search strategy where during the first phase single-join atomic configurations are used. Only indexes chosen in the first phase are considered in the second phase but no restrictions on atomic configurations are imposed during this phase. Due to space constraints, we don't discuss experimental results of this two-tier search strategy [CN97].

Example 2. Single-join Atomic Configurations

Consider a SELECT query with conditions T_1.A < 20, T_1.A = T_2.B, T_3.C BETWEEN [30,50], T_3.C = T_2.B. In this case, one 3-table atomic configuration is (T_1.A, T_2.B, T_3.C) since all three indexes may be used together to answer the query. However, due to the single-join atomic configuration based pruning step, the above atomic configuration is not evaluated. Rather, the cost of this query for the 3-table configuration is estimated by taking minimum of the costs of the atomic configurations: (T_1.A, T_2.B), (T_1.A, T_3.C), and (T_2.B, T_3.C).

3.2.2 Adaptive Strategy Based on Index Interaction

Our second technique does not make assumptions about the characteristics of the query processor. Instead it adaptively *detects* atomic configurations for a workload based on *interaction among indexes.* Assume that an atomic configuration C, of size n, has already been evaluated .The evaluated cost is compared with the derived cost (using techniques in Section 3.1). If the evaluated cost is significantly different from the derived cost, then it signals that indexes in C interact. In such cases, we will evaluate atomic configurations of size n + 1 that extend C. We formalize this intuition by the following algorithm:

1. n = 2; A = {*atomic* configurations of size <= 2}.
2. A'={}; Evaluate all configurations in A.
3. For each configuration C in A, determine if the indexes in C interact strongly. We do this by testing to see if the *evaluated* cost of the configuration is at least **x**% less than its *derived* cost.
4. If C meets the above condition, add all atomic configurations of size n+1 that are supersets of C to A'.
5. If A' = {}, then exit.
   Else A = A'; n = n + 1, Goto Step 2.

Figure 3. Adaptive Detection of Relevant Atomic Configurations.

The parameter **x** defines the threshold of index interaction. If the value of **x** is chosen too small, the risk of too many spurious atomic configurations being chosen in Step 4 is high. On the other hand, if **x** is too large, interaction among indexes can go undetected. Therefore, for the algorithm to perform well, choosing an appropriate value of **x** is important. Due to space constraints, we are unable to provide details on how to choose **x**; we defer this discussion and the experimental results to [CN97].

3.3 Reducing the Cost of Evaluating Atomic Configurations

When evaluating an atomic configuration, substantial savings are possible in the number of optimizer calls by the following optimization. The idea is that when asked to evaluate Cost (Q, C), we find a "simpler" atomic configuration C' such that Cost (Q, C) = Cost (Q, C'). Assume that the set of indexable columns for the Select/Update query Q[3] is P. Then, only indexes in C that are on one or more columns in the set P have effect on the cost of the query Q. If C' is the configuration consisting of only such indexes then Cost (Q, C) = Cost (Q, C'). If Cost (Q, C') has already been evaluated, we can simply reuse the cost. We refer to this step as the *relevant index set optimization.* An extreme case of relevant index set optimization occurs where the relevant set is empty. In this case, the estimated cost for the query is the same as that over a database with no indexes. We call this the *irrelevant index set optimization.*

Example 3: Reducing calls to the optimizer

Consider the following query Q_2:

SELECT * FROM onektup WHERE unique1 < 100

Assume that I_1 is an index on onektup.unique1 and I_2 is an index on tenktup1.unique1; we wish to evaluate the configuration C = {I_1, I_2}. The indexable columns of Q_2 are

[1] A *correlation* to be precise.

[2] Each index can either be clustered or non-clustered.

[3] Similar arguments apply for Insert/Delete queries. Details are available in [CN97].

{onektup.unique1}. Following the above optimization technique, Cost (Q_2, C) = Cost (Q_2, {I_1}). If Cost (Q_2, {I_1}) has been evaluated previously, then we save an optimizer invocation.

4. Candidate Index Selection

If we consider *every* admissible index of the workload, then too many spurious indexes will be considered, resulting in a blow up of the space of configurations that must be enumerated. We now describe our algorithm for picking the set of *candidate* indexes from the space of admissible indexes.

The idea is to determine the best configuration for each query *independently*, and consider all indexes that belong to one or more of these best configurations as the candidate index set. The intuition behind this algorithm is that an index that is not part of the best design for even a single query in the workload, is unlikely to be part of the best design for the entire workload. We refer to this technique as the *query-specific-best-configuration* candidate index selection algorithm. The challenge, therefore, is to determine the best configuration for each query Q in the workload.

We now make a key observation about choosing the best configuration for a query: the problem is no different than the overall problem of index selection itself, the difference is that *the workload consists of exactly one query.* We can therefore obtain the best configuration for each configuration by using the index selection tool itself! We now elaborate on this technique.

Let the workload **W** be {Q_1, .., Q_n}. The enumeration step (described in Section 5) picks the final set of **k** indexes by enumerating configurations over a set, **CI**, of candidate indexes. We denote this step by *Enumerate*(**k**, **CI**, **W**) where **CI** is the set of candidate indexes. When there are no bounds on **k**, we designate the step by *Enumerate* (**CI**, **W**). Let I_j denote the indexable columns of the query Q_j. We describe the algorithm in Figure 4 using these notations.

Step 2 of the algorithm shows that for the purpose of determining candidate indexes, we consider *all* indexable columns of the query as candidates. This is how we avoid "cyclic dependence" on the candidate index selection module.

1. For the given workload W that consists of n queries, we generate n workloads W_1..W_n each consisting of one query each, where Wi = {Q_i}
2. For each workload W_i, we use the set of indexable columns I_i of the query in W_i as starting candidate indexes.
3. Let C_i be the configuration picked by index selection tool for W_i, i.e., C_i = Enumerate (I_i, W_i).
4. The candidate index set for W is the union of all C_i's.

Figure 4. Candidate index selection algorithm.

We comment on several properties of this algorithm. First, if the workload is free of updates (including insert and delete statements), and if we do not impose any bound on the number of indexes that may be chosen, then the configuration chosen by this algorithm has less or equal total cost for the workload compared to any other configuration. However, if the workload contains updates or if there is a bound on the number of indexes to be chosen by the algorithm, then this guarantee does not hold. For example, say that the workload consists of a query Q with two indexable columns $T_1.C_1$ and $T_2.C_2$, and an insert query U on T_1 such that the best configuration for Q consists of an index on $T_1.C_1$ only. In such a case, if the index maintenance cost predicted for U is high, then it is possible for the index selection tool to not recommend any index since the enumeration phase will not consider $T_2.C_2$. Such situations result in inappropriate pruning. A similar observation may be made when we restrict the number of indexes to chosen by the above candidate index selection algorithm.

Although generalizations of the algorithm are possible to account for these observations (e.g. modifying Step 3 to pick the "next best" or the top few configuration(s) as well), the basic scheme did very well on a variety of workloads, including those with updates. This is due to a couple of factors. First, indexes that are part of the "next best" configuration of a query may appear as the best configuration for another query in the workload. Also, when there are multiple indexes selected in the best configuration, there may be considerable overlap between indexes in the best and the "next best" configurations. Finally, since Step 3 of the algorithm picks configurations without a bound on the number of indexes, we expect that most indexes from the "next best" configurations will also find their way into the set of candidate indexes.

5. Configuration Enumeration

If there are **n** candidate indexes, and the tool is asked to pick **k** indexes, a *naïve enumeration algorithm* would enumerate all subsets of the candidate indexes of size **k** or less, and pick the one with lowest total cost. The naïve enumeration algorithm guarantees an optimal solution; however, it is not practical, since for realistic values of **n** and **k** (e.g., **n**=40 and **k**=10) the number of configurations enumerated is too large to make exhaustive enumeration feasible.

1. Let S = the best **m** index configuration using the *naïve enumeration algorithm.* **If m = k then exit.**
2. Pick a new index I such that Cost (S U {I}, W) <= Cost(S U{I'}, W) for any choice of I' != I
3. **If** Cost (S U {I}) >= Cost(S) **then exit**
Else S = S U {I}
4. If |S| = **k** then exit
5. **Goto 2**

Figure 5. The Greedy(m, k) enumeration algorithm.

Our approach to the configuration enumeration problem is to use the Greedy (**m**, **k**) algorithm, shown in

Figure 5. The algorithm picks an optimal configuration of size **m** (where **m** <= **k**) as the "seed". This seed is then expanded in a greedy fashion until all **k** indexes have been chosen, or no further reduction in cost is possible by adding an index. Each greedy step considers all possible choices for adding one more index and adds the one resulting in the highest cost reduction. Note that at one extreme, if the parameter **m** = 0, then the algorithm takes a pure greedy approach. On the other hand, if **m** = **k**, the algorithm is identical to the naïve enumeration algorithm. Therefore, the use of the algorithm is computationally efficient only if **m** is small relative to **k**. In such a case, the enumeration exhibits near greedy behavior. The value of **m** relative to **k** reflects the desired degree of completeness of enumeration. The issue of heuristically determining an appropriate value of **m** depending on the index interactions among queries in the workload is discussed in [CN97]. This measure can also be adjusted by the user of the tool explicitly to vary the nature of enumeration from quick and heuristic to exhaustive. Despite the fact that a greedy algorithm can be in principle arbitrarily bad for configuration enumeration [CN97], our experimental results seem to indicate that a relatively low value of **m** produces near-optimal results.

The key reason why a greedy algorithm performs well for enumeration is that in many cases, despite interaction among indexes, the largest cost reductions often results from indexes that are good candidate indexes by themselves. Nonetheless, it is important to capture significant interactions, e.g., merge join using two clustered indexes, single table index intersection. This justifies the exhaustive phase of Greedy (**m**, **k**) since it helps capture interactions that have the most significant effect on cost. For example, observe that for any single query, the join order is often determined primarily by sizes of intermediate relations and presence (or absence) of a few important indexes. Once the join order has been determined, additional indexes may come into play, but such indexes only help reduce the cost of a join locally, and do not strongly interact with indexes used in other operations in the execution tree. In such cases, the exhaustive phase of Greedy (**m**, **k**) chooses the important interacting indexes that affect the join order and subsequently picks the remaining indexes greedily.

A variant of the enumeration algorithm described above uses branch-and-bound. The algorithm uses Greedy (**m**, **k**) with a low predetermined **m** to generate a configuration that serves as a first-cut solution. Subsequently, configurations are enumerated exhaustively with the constraint that the cost of each partial configuration[4] must be within a certain factor of the cost of the corresponding partial configuration of the first-cut (greedy) solution. This algorithm is explained in more details and compared with Greedy (**m**, **k**) in [CN97]

6. Multi-Column Index Generation

Index selection tools of the past have failed to take into account the complexity arising from the inclusion of multi-column indexes. For a given set of k columns on a table, k! multi-column indexes are possible, and considering all permutations can significantly increase the space of configurations that must be considered by the tool. In this section, we present a technique for dealing with this complexity.

We adopt an iterative approach for taking into account multi-column indexes of increasing width. In the first iteration, we only consider single-column indexes. Based on the single-column indexes picked in the first iteration, we select a set of admissible two-column indexes. This set of two-column indexes, *along with the "winning" single-column indexes*, becomes the input for the second iteration[5]. We use the notation M (*a*, *b*) to represent a two-column index on the columns *a* and *b* where *a* is the leading column of the two-column index.

Our strategy for selecting the set of two-column indexes is based on the intuition that for a two-column index to be desirable, a single-column index on its *leading* column must also be desirable. If S is the set of single column indexes picked in the first iteration, and *a* is a column such that an index on this column is in S, then we consider the set S' of all admissible multi-column indexes of the form M (*a*, *b*). Note that *b* must bean indexable column but there may not be any index in S that is on *b*. We call this algorithm MC-LEAD.

We also considered a more aggressive variant of MC-LEAD. In this variant, a two-column index M (*a*, *b*) is considered only if the set S contains an index on *a* as well on *b*. This algorithm is based on the assumption that for a multi-column index M (*a*, *b*) to be important, single-column indexes on both *a* and *b* must be important. We refer to this algorithm as MC-ALL. Note that the set of indexes considered by MC-ALL is a subset of the indexes considered by MC-LEAD. In Section 7, we present a comparison of the performance of these algorithms with the scheme where single-column and multi-column indexes are searched together in one pass (i.e., no iterative step). We refer to this algorithm as MC-BASIC. The results demonstrate that MC-LEAD is significantly superior.

These strategies for selecting candidate multi-column indexes are closely related to the technique used by our tool to consider indexes that help "index-only" access. In such cases, while considering M (*a*, *b*) above, *b* need *not* be an indexable column, but may be part of a projection list in the workload that is strongly correlated with the column *a*. These details are presented in [CN97].

7. Implementation and Experiments

7.1 Implementation

We have implemented our index selection tool for Microsoft SQL Server 7.0. The tool takes as input a workload of SQL DML statements (generated using the SQLTrace utility), and generates a set of indexes as output. On startup, the tool gathers schema information from the

[4] For a configuration consisting of a set S of indexes, all subsets of S are its partial configurations

[5] The above technique generalizes to the case where multi-column indexes of width less than or equal to k are needed, by performing k-1 iterations.

server and analyzes the given workload to determine a set of admissible indexes. We have modified SQL Server to support creation of what-if indexes, i.e., for a given what-if (hypothetical) index, the server makes available to the optimizer the distribution information of the column(s) over which the index is defined. This enables the optimizer to estimate costs of queries over configurations containing hypothetical indexes. When the tool needs to evaluate the workload for a configuration C, it "simulates" the presence of C for the query optimizer by loading the catalog tables of SQL Server with the distribution information of indexes in C. It then submits the queries in the workload to the optimizer in a batch. Batching not only reduces the communication overhead per query, but also ensures that the loading of distribution information for a given configuration needs to be done exactly once. The queries are optimized in the "no-execute" mode, and the optimizer returns a plan and a cost estimate for each query.

The cost of gathering distribution information can be amortized over multiple executions of the tool and multiple workloads, since the statistics need to be gathered only once per index. Details of creating and simulating "what-if" indexes, and the necessary changes to the server, will be described in a future paper. The tool must be run with administrator privileges since it needs to update system catalogs. The index selection tool is accompanied by a design assistant tool which can be used to analyze the current design using cost based techniques similar to those used by the index selection tool [CN97].

7.2 Experimental Setup

We have tested our tool on several schemas and workloads including synthetically generated schemas and workloads. Due to space constraints, we report results of our experiments on five representative workloads of the TPC-D schema only. Table 1 summarizes relevant characteristics of the workloads.

Workload	Number of Joins	% Pure Queries	% Update Stmts.	# admissible Indexes
TPCD_1	0, 1, 2	70	30	75
TPCD_2	3, 4, 5	70	30	143
TPCD_3	0-5	100	0	123
TPCD_4	0-5	50	50	108
TPCD_O	-	100	0	124

Table 1. Summary of workloads.

The first four workloads were generated using a synthetic workload generation program. This program has the ability to vary a number of parameters including percentage of update statements, number of joins in a query, selectivity of conditions, and frequency of conditions on a column. TPCD_1 consists of 0, 1, and 2-join queries only, whereas TPCD_2 consists of 3, 4, and 5-join queries. TPCD_3 and TPCD_4 differ in the percentage of update statements in the workload. TPCD_O is a workload consisting of the set of 17 TPC-D queries as specified in the benchmark. In our experiments, we considered multi-column indexes (of width two) along with single column indexes as possible structures of indexes. The last column of Table 1 shows the total number of *admissible* indexes for each workload. Unless otherwise stated, the numbers we report are for the case when the tool was asked to select ten indexes.

7.3 Experiments

7.3.1 Candidate Index Selection

To evaluate the performance of the *query-specific-best-configuration* algorithm (denoted as BEST-CONF) for picking candidate indexes, we compared the total number of *candidate* indexes picked by the algorithm with the number of *admissible* indexes[6] considered by our algorithm for the workload. Figure 6 shows that for each workload, the number of candidate indexes chosen by the BEST-CONF algorithm is significantly smaller than the number of admissible indexes for that workload.

We found no degradation in quality of the final configuration picked by the index selection tool when the BEST-CONF algorithm is used (even for the update intensive workload TPCD_4). To provide insight into the characteristics of BEST-CONF, we considered two variants, BEST-CONF-1, and BEST-CONF-2, that choose at most one (and respectively two) candidate index(es) for each query in the workload. Table 2 underscores the importance of the strategy used in BEST-CONF of including indexes in the "best" configuration for each query without any predetermined bound for reasons explained in Section 4. Overall, the results of these experiments clearly bring out the effectiveness of our candidate index selection algorithm.

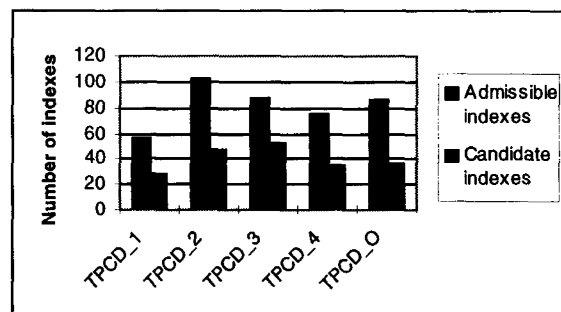

Figure 6. Performance of candidate index selection algorithm.

	BEST-CONF-1	BEST-CONF-2	BEST-CONF
TPCD_1	1%	0%	0%
TPCD_2	82%	4%	0%
TPCD_3	47%	0%	0%
TPCD_4	0%	0%	0%
TPCD_O	15%	2%	0%

Table 2. Drop in quality of final configuration.

[6] Note that the number of admissible indexes passing through the candidate index selection module is less than the total number of admissible indexes for the workload, since not all two-column indexes are selected for the second iteration.

7.3.2 Effectiveness of Query Processor Based Restrictions on Atomic Configurations

In this experiment we show the effectiveness of using the cost estimates based on single-join atomic configurations described in Section 3.2.1. We refer to this pruning technique as **SJ** in the figures below. We compare **SJ** against the cost estimations that do not use any query processor constraints. We refer to the latter as **MJ**. Figure 7 shows the comparison in the number of atomic configurations evaluated by **SJ** and **MJ**. For TPCD_1, the reduction in atomic configurations using **SJ** is not very large since the workload consists of only 0, 1, and 2-join queries. However, the benefits of **SJ** for TPCD_2, which consists of 3, 4 and 5-join queries, and for TPCD_O which consists of many complex queries are 50% and 61% respectively. We conclude that for workloads with large join queries, **SJ** can significantly reduce the number of configurations evaluated.

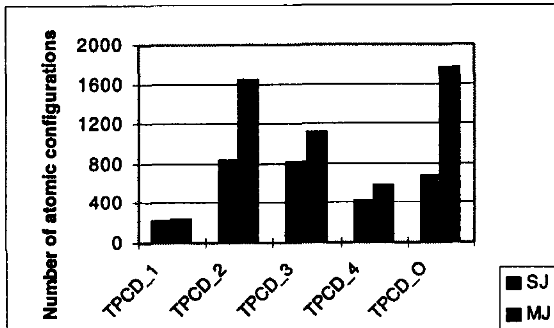

Figure 7. Number of atomic configurations for SJ vs. MJ.

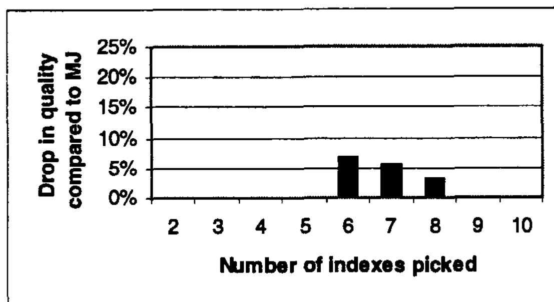

Figure 8. Drop in quality when using SJ as compared to MJ for TPCD_2.

We now compare the *quality* of the configuration picked for each workload when using **SJ** and **MJ** respectively. Our results show that when asked to pick 10 indexes (**k** = 10), the configuration picked when using **SJ** was identical to that picked using **MJ**. To provide intuition on why **SJ** performs well, we designed an experiment where the tool was invoked repeatedly with the number of indexes picked (**k**) being varied from 2 to 10 for the TPCD_2 workload. Figure 8 shows that for 2 <= **k** <= 5, the resulting configurations using **SJ** and **MJ** were the same. However, the *sixth* index that is picked when using **MJ** is the *ninth* index picked when using **SJ**. Hence for 6 <=**k** <=8, using **SJ** caused a drop in quality. The intuitive explanation for this behavior is that when k is small, the "obviously" good indexes are selected in both cases. When **k** is large, most indexes that provide *any* benefit will be included. It is in the "middle" range of **k** that we can expect deviation between **SJ** and **MJ**. We note however, that the maximum drop in quality is quite small (7%). Similar results were observed for other workloads. We conclude that query processor based restrictions provide a reasonable method of restricting the set of atomic configurations that need to be evaluated for a workload.

7.3.3 Reducing calls to the optimizer.

This experiment shows the extent to which the number of calls to the optimizer can be reduced through the optimizations discussed in Section 3.3. We found that the total reduction in optimizer calls across the workloads varied from 79% to 92%. While most of the savings were due to *irrelevant index set optimization* (58%-84%), *relevant index set optimization* further reduced the number of optimizer calls by 9%-29%. The experiment shows that these optimizations are truly significant.

7.3.4 Performance of Greedy vs. Optimal

In this experiment we compare Greedy (2, **k**) with the naïve enumeration algorithm (discussed in Section 5). The naïve enumeration algorithm does an exhaustive search of the configuration space and therefore finds the optimal configuration of a given size (we refer to this as Optimal). In each case Greedy and Optimal were asked to pick 4 indexes (**k**=4). The experiments were run *without* the query-processor-based-restrictions on atomic configuration, to ensure that accurate information of atomic configuration costs were available.

We observed that the fraction of configurations enumerated by Greedy as compared to Optimal varied from 2% to 7% for the different workloads. Recall that Greedy (2, k) starts by considering an optimal configuration of size two. Nonetheless, when asked to pick four indexes, the exhaustive nature of the Optimal causes it to explore a much larger space of configurations than Greedy. In all but one workload, using Greedy did not compromise optimality. The one counter-example was TPCD_4 (an update intensive workload) in which Optimal chose a different clustered index for the *lineitem* table than Greedy. However, the absolute difference in cost between the optimal configuration and the one chosen by Greedy was very small (< 1%).

Our conclusion based on this experiment (and our experience with Greedy on other databases and workloads) is that Greedy (2, k) does very well for a large variety of workloads over Microsoft SQL Server.

7.3.5 Multi-Column Index Generation

We compared the algorithms for selecting admissible multi-column indexes that are described in Section 6. Figure 9 shows the number of admissible multi-column indexes picked by each algorithm.

We note that for most workloads, both techniques substantially reduce the number of admissible multi-column indexes picked when compared to MC-BASIC. Table 3 shows that MC-LEAD performs very well when compared with MC-BASIC, whereas MC-ALL suffers a noticeable

drop in quality for two workloads. The success of MC-LEAD confirms the intuition that even if a single column index on column *a* is not a "winner", its interaction with another (winning) column *b* can be significant enough to make a multi-column index M (*b*, *a*) attractive. MC-LEAD fails only when neither *a* nor *b* is important as a single column index, but M (a, b) is important. We expect that such cases are relatively rare; and even when they do occur, their impact on the total cost is fairly small in practice as can be seen for TPCD_2 and TPCD_O.

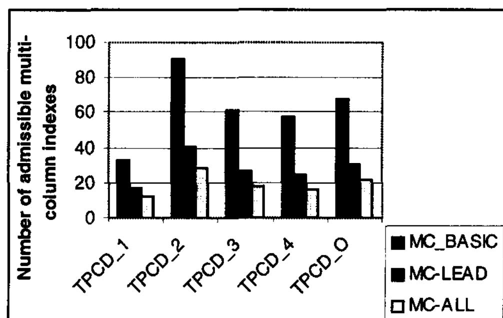

Figure 9. Number of admissible multi-column indexes.

	MC-LEAD	MC -ALL
TPCD_1	0%	0%
TPCD_2	6%	18%
TPCD_3	0%	2%
TPCD_4	0%	5%
TPCD_O	2%	14%

Table 3. Drop in quality of configuration chosen compared to MC-BASIC

7.4 Putting It All Together

In this section we summarize our algorithm for index selection:

1. Run the *query-specific-best-configuration algorithm* for identifying candidate indexes. This requires splitting the given workload of *n* queries into *n* workloads of one query each, and finding the best configuration for each query. The union of these configurations is the candidate index set for Step 2.
2. Run the Greedy (**m**, **k**) algorithm to enumerate configurations subject to the *single-join atomic configuration pruning* technique and select a set of indexes until the required number of indexes have been picked or total cost can be reduced no further. We found that **m**=2 produced very good solutions.
3. Select a set of admissible multi-column indexes using the technique MC-LEAD, based on single-column indexes picked in Step 2.
4. Repeat steps 1 and 2 starting with the admissible index set consisting of single-column indexes that were chosen in Step 2, and multi-column indexes selected in Step 3.

Figure 10. Summary of index selection algorithm.

We now compare the performance of our algorithm described above with a "baseline" algorithm. The baseline algorithm differs from our algorithm in that is non-iterative and it does not execute the candidate index selection step. It therefore considers all admissible indexes for the workload (including multi-column indexes) during enumeration. It runs the Greedy (2, **k**) algorithm, but *without* imposing any query processor based restrictions on atomic configurations. Table 4 shows that in every important category (number of candidate indexes, number of optimizer calls, number of configurations enumerated), our algorithm does significantly less work than the baseline algorithm. The overall running time of our algorithm improves by a factor of 4 to 10 over the baseline algorithm. Moreover, as the final column of Table 4 shows, the drop in quality of the final configuration picked by our algorithm is very small[7].

	Num. Candidate indexes	Num. optimizer calls	Num. Configs. Enum.	Running time	Drop in quality
TPCD_1	38%	52%	51%	24%	0%
TPCD_2	33%	8%	28%	12%	7%
TPCD_3	43%	11%	57%	14%	6%
TPCD_4	32%	11%	39%	18%	0%
TPCD_O	30%	3%	33%	10%	1%

Table 4. Comparison of index selection algorithm to "baseline" algorithm.

8. Related Work

There is a substantial body of literature on physical database design dating back to the early 70's. In this section, we briefly review representatives of the major directions of work. Previous work in index selection that takes into account workload information can be divided into the following two categories: (1) identifying a set of possible indexes and configurations for the given workload (2) searching over the space of possible indexes.

The problem of identifying a set of possible indexes has been looked at from two angles. Syntactic analysis of the workload is used to identify potentially useful indexes. This is the approach taken in [FST88, HC76]. In [FON92], an alternative approach is to generate the set of all configurations for each query that may be potentially used by the optimizer and then choose among the union of all such configurations over the queries in the workload. This technique is not scalable for large workloads. The framework by Rozen and Shasha [RS91] suggests generation of a set of candidate configurations for each query based on a knowledge-based approach. The set of configurations explored is the union of only those configurations. This idea has been further pursued by [CBC, CBC93] who propose a rule-based framework. The knowledge-based approach has also been taken in the commercial product RdbExpert [HE 91]. Our approach to candidate index selection (Section 4) is distinct from these approaches.

[7] The observed deviation from the baseline algorithm occurred because of our approximation for dealing with multi-column indexes.

The problem of selecting a configuration from the set of candidate indexes has two aspects. First, a cost function to characterize the goodness of a configuration is needed. In our approach and in [FST88], optimizer-cost driven estimates are used. An approximate "stand-alone" cost model is used in [HC76, CBC]. Next, an efficient search technique that does not compromise the quality of the solution is needed. Several greedy-like search algorithms have been proposed in the past. More recently, several variants of greedy algorithms have also been recommended in the context of the materialized view and index selection problem. In particular, [HRU96, GHRU97] show bounds on the deviation of the greedy from the optimal. However, their results assume monotonicity, i.e., inclusion of one index does not have any impact on the effectiveness of another. By exploiting the exhaustive phase of Greedy (**m**, **k**), we have been able to capture some of the significant index interactions that a traditional greedy algorithm is unable to capture.

9. Conclusion

Index selection is one important aspect of physical database design. In this paper we have described our work on the design and implementation of an index selection tool for Microsoft SQL Server. The effectiveness of our algorithm for index selection can be attributed to three novel techniques that we present in this paper:

- Query-specific-best-configuration algorithm for choosing candidate indexes.
- An algorithm to reduce the number of atomic configurations (and hence number of optimizer calls) that must be evaluated for a workload.
- An iterative technique for handling multi-column indexes.

Our experimental results show that using these techniques can increase the overall efficiency of the tool by a factor of 4 to 10 without significantly compromising the quality of indexes selected. Our future work will explore techniques to choose other structures (e.g., join indexes and materialized views) for physical database design in addition to indexes.

References

[CBC] Choenni S., Blanken H. M., Chang T., "On the Automation of Physical Database Design" , Proc. of ACM-SAC, 1993.

[CBC93] Choenni S., Blanken H. M., Chang T., "Index Selection in Relational Databases", Proc. of 5th IEEE ICCI 1993.

[CG93] Peter C., Gurry M., "ORACLE Performance Tuning", O'Reilly & Associates, Inc. 1993.

[CN97] Chaudhuri S., Narasayya V., Physical Database Design Assistant and Wizard for SQL Server, Microsoft Research Technical report, *in preparation,* 1997

[FON92] Frank M., Omiecinski E., Navathe S., "Adaptive and Automative Index Selection in RDBMS", Proc. of EDBT 92.

[FST88] Finkelstein S, Schkolnick M, Tiberio P."Physical Database Design for Relational Databases", ACM TODS, Mar 1988.

[GHRU97] Gupta H., Harinarayan V., Rajaramana A., Ullman J.D., "Index Selection for OLAP", Proc. of ICDE97.

[HC76] Hammer M., Chan A., "Index Selection in a Self-Adaptive Data Base Management System", Proc. of SIGMOD 76.

[HE91] Hobbs L., England K., "Rdb/VMS A Comprehensive Guide", Digital Press, 1991.

[RSS96] Ross K. A., Srivastava D., Sudarshan S., "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time", Proc. of SIGMOD 96.

[HRU96] Harinarayan V., Rajaramana A., Ullman J.D., "Implementing Data Cubes Efficiently", Proc. of SIGMOD 96.

[LQA97] Labio W.J., Quass D., Adelberg B., "Physical Database Design for Data Warehouses", Proc. of ICDE97.

[RS91] Rozen S., Shasha D. "A Framework for Automating Physical Database Design", Proc. of VLDB 1991.