Index Merging

Surajit Chaudhuri and Vivek Narasayya[1]
Microsoft Research, Redmond
Email: {surajitc, viveknar}@microsoft.com
http://research.microsoft.com/~{surajitc, viveknar}

Abstract

Indexes play a vital role in decision support systems by reducing the cost of answering complex queries. A popular methodology for choosing indexes that is adopted by database administrators as well as automatic tools is: (a) Consider poorly performing queries in the workload. (b) For each query, propose a set of candidate indexes that potentially benefits the query. (c) Choose a subset from the candidate indexes in (b). Unfortunately, such a strategy can result in significant storage and index maintenance cost. In this paper, we present a novel technique called index merging to address the above shortcoming. Index merging can take an existing set of indexes (perhaps optimized for individual queries in the workload), and produce a new set of indexes with significantly lower storage and maintenance overhead, while retaining almost all the querying benefits of the initial set of indexes. We present an efficient algorithm for index merging, and demonstrate significant savings in index storage and maintenance by virtue of index merging, through experiments on Microsoft SQL Server 7.0.

1. Introduction

Decision support systems and data warehouses rely on indexes for achieving good performance. Picking appropriate indexes for such systems is guided by *workload* information that is representative of the queries posted against the server. Such a workload may consist of customer benchmarks or queries logged by the system. Database administrators (DBA-s) and index selection tools exploit the workload information to create/add appropriate indexes for the database. One popular methodology for choosing indexes consists of the following steps: (a) Consider poorly performing queries in the workload (b) For each query, consider a set of candidate indexes that potentially benefits the query (c) Choose a subset of the indexes proposed in (b). Such a methodology is not only adopted by DBA-s, but is also at the heart of modern index selection tools [CNITW98, CN97].

The above methodology suffers from the key shortcoming that the space of indexes considered is generated by looking at the workload *one query at a time*. A set of indexes chosen using this methodology can therefore result in excessive storage requirements as well as index maintenance cost. Consider for example, the Q_1 and Q_3 queries of the TPC-D benchmark. Q_1 benefits significantly from an index I_1 on the lineitem table on columns (l_shipdate, l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax). I_1 is a *covering index* for Q_1, i.e., it contains all columns from lineitem required to answer the query. Similarly, a covering index I_2 on columns (l_shipdate, l_orderkey, l_extendedprice, l_discount) is beneficial for query Q_3. However, a single index I' on the columns (l_shipdate, l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax, l_orderkey) can reduce (a) the storage requirement by 38% and (b) the index maintenance cost for batch insertions by 22%, while the combined cost of queries Q_1 and Q_3 increases by only 3% compared to the case when both I_1 and I_2 exist.

Unfortunately, identifying candidate indexes by analyzing interactions among multiple queries in the workload at once is difficult since that could lead to an explosion in the space of candidate indexes considered, particularly for complex or large workloads. Therefore, in this paper we have pursued an alternative approach. We present a novel technique called *index merging* that uses a given set of indexes and derives a *new* set of indexes that significantly reduces the storage and maintenance overheads while retaining almost all querying benefits of the given initial set of indexes. Thus, index merging technique is able to take as input a set of indexes that has been optimized for individual queries in the workload and is able to generate a new set of indexes that is superior. To illustrate the benefits of index merging, consider for example the TPC-D database [TPC93] and the 17 queries defined in the benchmark. If we build indexes by tuning each query individually, then the storage space required for the indexes exceeds the data size by about a factor of 5. However, by applying index merging to the set of indexes generated by tuning queries individually, we can reduce the storage requirement to about 2.3 times the data size, even though the average cost of the queries increases only by about 5% compared to the initial set.

Naturally, an index merging component should be an integral part of an index selection tool to enable choosing indexes that have low storage and maintenance overhead. In fact, we have incorporated the index merging techniques into

[1] Also at the University of Washington Computer Science and Engineering Department.

the index selection tool that we built for Microsoft SQL Server 7.0 [CNITW98, CN97] and have obtained significant benefit. We note that index merging can be a valuable tool in itself, that can improve any given set of initial indexes.

1.1 Outline of the paper

To the best of our knowledge, this is the first paper to study the concept of index merging, implement index merging on a commercial DBMS system (Microsoft SQL Server 7.0) and present experimental results. The rest of the paper is organized as follows. In Section 2 we discuss related work. Section 3 is the heart of our paper. In this section, we present a formal framework for index merging and define the problem of finding a set of indexes that exploits index merging. We present an architecture that modularizes the above search problem, and also describe an efficient algorithm for finding a set of indexes that minimizes the storage overhead given an upper bound on the increase in the workload execution time that can be tolerated. In Section 4 we describe the results of experiments based on our implementation of a client utility for index merging on Microsoft SQL Server 7.0, which show that our algorithm produces good solutions on several databases and workloads. We present our conclusions in Section 5.

2. Related Work

In the past, there has been lots of work in the area of index selection [CBC93,CN97,FON92,FST88,GHRU97,RS91]. In all prior work on index selection that we are aware of, every index considered by the algorithms is part of an "optimal" set of indexes for at least one query in the workload. Index merging however, is concerned with the space of indexes that is derived from an existing set of indexes that have been created for individual queries in the workload. Thus, the indexes explored are not necessarily optimal for any query in the workload. Moreover, unlike the focus of index selection, which is to minimize the cost of the queries under a given storage constraint, index merging aims to minimize storage under a given cost constraint.

The problem of finding the right set of "merged" indexes from an initial set of indexes is similar to finding appropriate vertical partitions for a given database and workload [C92,CY90,DP88,HN79,HS75,M83,NCWD84,NR89].
However, there are two key differences between these problems. First, an index is a *redundant* vertical slice of the base relation, whereas a vertical partition physically separates columns of the base relation. This distinction significantly affects query optimization since if an index is not available the optimizer can still scan the base relation, whereas if the columns required for a query are not available in a vertical partition, the optimizer must join multiple partitions to answer the query. Second, unlike a vertical partition, indexes can also be used for selection via lookup. Thus, algorithms for index merging must be sensitive to this issue.

Finally, we differ from most previous work in index selection and vertical partitioning except [FST88,CN97] in that (a) we use workload information to guide the index merging process (b) we rely on the optimizer's cost estimates of a query for evaluating the merit of a merged set of indexes, rather than using an approximate external cost model.

3. Index Merging

In this section we present a framework for index merging and define the problem (Section 3.1), outline our solution to the this problem (Section 3.2), and describe the three important components in our architecture for index merging: Merging a given pair of indexes (Section 3.3), Search Strategy (Section 3.4), and Cost Evaluation (Section 3.5).

3.1 Framework for Index Merging

In this paper, we use the term **configuration** to mean a set of indexes. We use the term **storage** of an index to mean the space required to store the index on disk. The storage of a configuration C is the sum of the storage of indexes in C. We now define what it means to *merge* two or more indexes. Intuitively, merging preserves the property that if prior to being merged, an index contained the columns required to answer a query, then the resulting index after merging also contains the columns required to answer the query.

Definition 1. Merging a Set of Indexes. A set of indexes $I = \{I_1, I_2, \ldots I_N\}$ are said to be merged to form index M if (a) Every column of each index in I is present in M and (b) M contains no columns that are not present in some index $Ij \in I$. The indexes $I_1, I_2, \ldots, I_N$ are referred to as the *parent indexes* and M is referred to as a *merged index*.

We note that if the indexes I_1 I_2, … I_N contain a total of k distinct columns, then k! different mergings of the N indexes are possible since each permutation of the columns determines a different index.

Example 1. Merging two indexes
Consider the lineitem table in the TPC-D benchmark. Let I_1 be a 4-column index on (l_shipdate, l_discount, l_extendedprice, l_quantity). Let I_2 be a 3-column index on (l_orderkey, l_discount, l_extendedprice). Then a total of 5! mergings of the two indexes are possible, and two such merged indexes are:
M_1 = (l_shipdate, l_discount, l_extendedprice, l_quantity, l_orderkey) and
M_2 = (l_orderkey, l_shipdate, l_discount, l_extendedprice, l_quantity)

Note that a merged index may not retain the indexing benefits of its parent indexes. For example, M_2 cannot be used efficiently for lookup (i.e. an *index seek* operation) to answer a query with an equality/range condition on l_shipdate. Although a set of indexes can be merged in many possible ways, a class of merges that is of particular interest is called *index preserving merges*. Intuitively the goal of an index preserving merge is to try to maintain (at least partially) the index seek benefit of the indexes being merged.

Definition 2. Index Preserving Merge
An index preserving merge of a given set of indexes I = {I_1, I_2 , … I_N} is an index M constructed using a succession of merges as follows:

(1) M consists of all columns of one of the indexes $I_j \in I$, in the same order as in I_j. $I = I - \{I_j\}$.
(2) Append to M all columns (that do not already appear in M) of an index $I_k \in I$ in the same order as in I_k. $I = I - \{I_k\}$.
(3) Repeat Step (2) until I is empty.

Example 2. Index preserving merge
In Example 1 above, M_1 is an index preserving merge since it has I_1 as its leading prefix followed by the (distinct) columns from I_2 appended in order. M_2 is not an index preserving merge since neither I_1 nor I_2 is a leading prefix of M_2. The only other index preserving merge possible in this case is (l_orderkey, l_discount, l_extendedprice, l_shipdate, l_quantity)

In Definition 2 we note that (a) the indexes from I are merged in some pre-determined order, and (b) one of the indexes, I_j, becomes a leading prefix of M. Thus the index lookup benefits of I_j are preserved in M. Furthermore, an index preserving merge has desirable behavior when we merge two indexes where one is a *prefix* of another. For example, if we merge an index on columns (A, B) with an index on (A,B,C) an index preserving merge will always produce the merged index (A,B,C), thereby preserving the index lookup benefit of both parent indexes. Finally, we observe that if we are merging p indexes, at most p! index preserving merges are possible.

Definition 3. Minimal Merged Configuration
A configuration C' = {J_1, J_2, …, J_k} is said to be a minimal merged configuration with respect to an initial configuration C = {I_1, I_2, … I_N} if

- Each J_i, $1 \le i \le k$, ($k \le N$), is either one of the indexes I_p, $1 \le p \le N$, or is the result of merging two or more indexes in C.
- For all p, q, J_p and J_q do not share a parent index from C (see Definition 1).

Thus, a minimal merged configuration C' is guaranteed to have no more indexes than the initial configuration C. When C' contains one or more merged indexes, the number of indexes in C' will be less than that of C. In such cases the storage required for C' can be significantly less than that of C. Although the Definition 3 does not require the use of index preserving merges, in this paper we restrict our study to the space of minimal merged configurations derived using only index preserving merges.

We now state the *index merging problem* as follows:
Input:

- An initial configuration C = {I_1, I_2, …I_N}
- A workload W of queries {Q_1, Q_2, … Q_P}
- An upper bound U , on the cost of the workload, referred to as the *cost-constraint.*

Goal: Find a minimal merged configuration C' = {J_1, J_2, … J_k} , k<= N with respect to C, such that

- Cost(W, C') ≤ U, where Cost(W, C') is the cost (or estimated cost) of executing the workload when the database configuration is C'.
- C' has the lowest storage among all merged configurations, derived using only index preserving merges, satisfying the cost- constraint U.

We call the above problem the *Storage-Minimal Index Merging* problem. In this paper, we focus on solutions to the Storage-Minimal Index Merging problem. Note that this formulation has the attractive property that it guarantees an upper bound on performance degradation due to index merging. Although not explored in this paper, we note that a dual formulation of the problem is possible where the goal is to minimize the cost of the workload subject to a maximum storage constraint. We refer to the second formulation of the problem as the *Cost-Minimal Index Merging* problem.

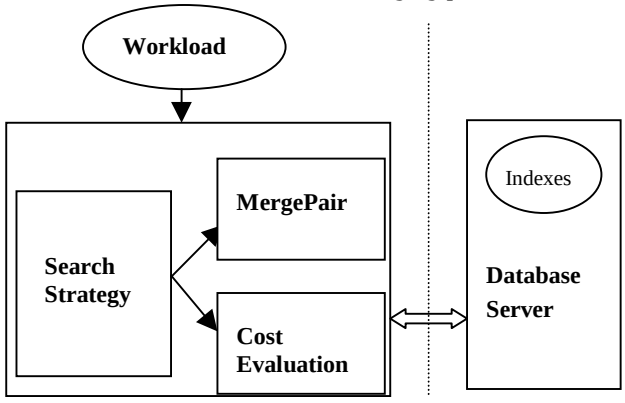

Figure 1. Architecture of index merging algorithm.

3.2 Our Approach

The architecture of our solution to the index merging problem is presented in Figure 1. As explained in the introduction, our approach is to take into account the workload information faced by the system. Although determining a representative workload for a system is itself a challenging task, there are several reasonable solutions. For example, the database administrator may use organization specific representative workloads. Another approach is to generate a log of SQL queries (and updates) at the server. Most database systems today offer a mechanism to log events at the server. For example, in Microsoft SQL Server, the SQL Server Profiler provides this capability.

We explore the space of possible merged configurations carefully by using three important components in our architecture. First, we approach the index merging process by a sequence of pair-wise merging of indexes. Therefore, in the context of index preserving merges, the important issue is

how the indexes are ordered during merging. This functionality is provided in our architecture by the *MergePair* module (Section 3.3), which takes as input a pair of indexes and produces a merged index. The *Search strategy* module is responsible for deciding the sequence of pair-wise mergings required to arrive at a good overall merged configuration. We describe our search strategy in Section 3.4. The search strategy uses the *Cost Evaluation* module, which takes as input a workload W and a configuration M, and returns the cost of the workload W for the configuration M. We denote this function as Cost (W, M). The cost evaluation module must be able to provide the cost even when M contains merged indexes which do not physically exist in the database. We describe alternative implementations of Cost (W, C) in Section 3.5. In one implementation, the cost evaluation module needs to communicate with the database server to obtain the cost of a query. The block arrow from the cost evaluation module to the server represents the interfaces in the server for obtaining this cost. The choices made in these three modules are responsible for the quality as well as running time of the algorithm for index merging.

3.3 Merging a Pair of Indexes

In this section we present two alternatives for merging a pair of indexes using an index preserving merge. The first alternative uses the cost of queries and the usage of indexes in the initial configuration C to determine the order of indexes, whereas the second alternative relies only on syntactic information in the workload. For purposes of comparison in our experimental evaluation, we also use the **MergePair-Exhaustive** procedure, which considers all possible ways[2] of merging a given pair of indexes, i.e. is not restricted to index preserving merges.

The MergePair module also needs to compute the expected storage space taken up by a merged index, i.e. its size. The size of an index can be accurately predicted if we know the on-disk structure used to store the index. For example, if indexes are stored as B+-Trees, we can estimate the number of pages required for the index, given the width of columns in the index and the number of tuples in the relation on which the index is defined.

3.3.1 Merging based on Cost and Index Usage

Our approach to merging a pair of indexes is based on the observation that indexes can be used in two fundamental ways: *index seek* and *index scan*. Index seek is used to retrieve a subset of the rows from a table. For example, if a query has a condition in the WHERE clause such as T.a = 10, then an index on column T.a can be used for index seek to answer the query by retrieving all rows from T that match the condition. Index scan is used to read in a subset of the columns of a table, i.e. a vertical slice, that contains the columns referenced in a query. For example, if a query only references columns A and B from a table, then an index on (A,B) can be used to scan for the data rather than scanning the entire table. It is important to differentiate between these two types of index usage since, when merging indexes, the order of the columns in the merged index affects its usage for index seek but *not* index scan. For example, consider indexes I_1= (A,B) and I_2=(B,C). If we generate the merged index M = (B,C,A), then M is not useful for answering queries with conditions on A using index seek. However, M can still be used to answer queries that reference only the columns A and B using an index scan (although at a slightly higher cost due to the additional column C).

Figure 2 presents the procedure MergePair-Cost, for merging a pair of indexes I_1 and I_2. Seek-Cost (W, I) denotes the cost of all queries in the workload W where I was used for index seek. The MergePair-Cost procedure always generates an index preserving merge by making the index with the *higher* Seek-Cost the leading prefix of the merged index. This heuristic attempts to minimize the increase in the cost of queries due to merging. The underlying intuition is that the absence of an index for lookup typically has a multiplicative effect on the cost of a query. Since the leading columns of M are the same as those of *Leading*, M can be used for index seek in queries where *Leading* was previously being used for index seek. M can also be used for index scan in queries in which *Leading* and *Trailing* were being used for index scan, since the order of columns in M is *not relevant* for index scan. Thus the only queries that are likely to increase in cost significantly are those where *Trailing* was being used for index seek.

1. **If** Seek-Cost(W, I_1) ≥ Seek-Cost(W, I_2)
2. **Then** *Leading* = I_1, *Trailing* = I_2.
3. **Else** *Leading* = I_2, *Trailing* = I_1.
4. Generate M by performing an index preserving merge of I_1 and I_2 where the leading prefix of M is *Leading*.
5. **Return** M

Figure 2. The MergePair-Cost procedure.

The MergePair-Cost procedure requires computing the Seek-Cost of an index in the initial configuration for the given workload. This information can be gathered by examining the plan and cost of each query in W for the initial configuration. Most database systems support such a mechanism to obtain the plan and cost of a query, without having to execute the query. In Microsoft SQL Server, this functionality is provided by the *Showplan* mechanism.

1. Assign a frequency to each index I_1 and I_2 based on the number of appearances of the leading column of the index in (a) A condition (selection/join) (b) An Order By clause (c) A Group By clause (d) A Select clause.
2. Generate an index preserving merged index M of I_1 and I_2 where the index with the higher frequency is the leading prefix of M
3. **Return** M

Figure 3. The MergePair-Syntactic procedure.

[2] If there are altogether k distinct columns in the two indexes, then there are k! ways of merging the two indexes.

3.3.2 *Merging based on Query Syntax*

In order to evaluate the importance of cost and index usage information in merging a pair of indexes, we present a third implementation of MergePair that we refer to as **MergePair-Syntactic** (shown in Figure 3). The decision of which index precedes the other during a merge is not made based on index usage information, but only on the parsed information of queries in the workload. Our experiments show that such a procedure that ignores the cost and index usage performs relatively poorly in practice.

3.4 Search Strategy

Given an initial configuration C consisting of N indexes, an obvious search strategy is to exhaustively enumerate every possible merged configuration with respect to C derived using MergePair, and pick the solution that results in maximum storage savings that also meets the given cost constraint U. Although such an *Exhaustive* algorithm is guaranteed to give the optimal solution, it is infeasible in practice (e.g. N=20) since the number of possible merged configurations grows rapidly as we increase N, even when we restrict ourselves to minimal merged configurations.

1. C' = C /* C is the initial configuration */
2. **Do** /* Loop forever */
3. Let S = Set of all merged pairs of indexes in C' using the MergePair procedure
4. **If** S is empty **Return** C'
5. **For** each merged index M in descending order of storage reduction (Let I_a and I_b be the parent indexes of M.)
6. C'' = C' – {I_a} –{I_b} + {M}
7. **If** (Cost (W, C'') ≤ U) C' = C''; **Break**
8. **End For**
9. **If** no merged configuration satisfies cost-constraint in Step 7, **Return** C'
10. **End Do**

Figure 4. The Greedy Algorithm

3.4.1 *Greedy Search Strategy*

Our solution to the search problem is to use a *greedy search strategy* for enumerating the space of merged configurations. The greedy algorithm, outlined in Figure 4, iteratively finds better merged configurations. It uses the function MergePair, which takes as input a pair of indexes and returns a merged index and its expected storage requirement. In each execution of the outer loop (Steps 2-10) the algorithm considers all merged configurations obtained by replacing a pair of indexes in the current configuration (C') with a single merged index. Among all such merged configurations considered, the algorithm picks the merged configuration C'', that results in the largest storage reduction compared to C', that also satisfies the cost constraint U. The algorithm then repeats the outer loop with C'' as the current configuration. The algorithm terminates when no merged configurations satisfying the cost constraint are found in an iteration.

In practice this strategy performs extremely well and our experiments show that it produces reduction in storage comparable to the *Exhaustive* algorithm (which considers all possible minimal merged configurations derived using MergePair). Furthermore, unlike the *Exhaustive* algorithm, the greedy algorithm runs in polynomial time in the number of indexes in the initial configuration.

3.4.1.1 *Analysis of Greedy Algorithm*

If the number of indexes in the initial configuration is N, the work done in the first iteration is O (N^2 log(N)), which is the time to order the merged pairs by storage improvement (Step 5). In all iterations after the first, the work done is O (N^2) since the only new pairs considered are those resulting from the newly added merged index from the previous iteration. Since the outer loop is executed at most (N-1) times, the overall running time of the algorithm is O (N^3).

We note that the function Cost (W, C'') can potentially be expensive to compute if (a) the workload is large or (b) an accurate costing technique such as the optimizer-estimated cost is used (see Section 3.5). In such cases, checking whether the cost constraint is satisfied in Step 7 can dominate the running time of the algorithm.

In practice we observe that the algorithm executes much faster than the expected worst case running time of O (N^3). One reason is that the number of configurations for which we invoke Cost (W, C'') per iteration is relatively small since a merge that results in large storage reduction is also likely to result in a small increase in workload cost; thereby making it likely to satisfy the cost-constraint in Step 7. This is true since a large storage reduction by merging two indexes implies that the indexes have a high degree of overlap in their columns (e.g. when one index is a prefix of another).

3.5 Cost Evaluation

The search strategies for index merging (described in Section 3.4) require availability of the function Cost (W, C), which returns the total cost of all queries in the workload when the database configuration is C. In this section, we present possible alternatives for implementing this cost function. A good implementation of the cost evaluation module is crucial since an inaccurate cost estimate can result in poor quality of the resulting merged configuration. An important requirement on the cost evaluation module is that it must be able to estimate the cost even though the merged indexes in C do not physically exist in the database.

The obvious solution is to implement Cost (W, C) by materializing the indexes in C and executing the queries in the workload to obtain the cost. While the execution cost provides accurate information to the algorithm, it is prohibitively expensive to obtain and can cause serious disruption to operational queries since it requires creating (and dropping) indexes. Therefore, in practice, using the execution cost is infeasible.

3.5.1 *No-Cost Model*

The purpose of the cost evaluation module is to be able to determine if a given merged configuration satisfies the cost-

constraint (see Section 3.1 for definition). Therefore a simplistic model (which we refer to as the *No-Cost* model) is to assume that the cost constraint is met by a merged configuration if and only if merged indexes in the configuration satisfy certain syntactic constraints. The No-Cost model requires that (a) the width of each merged index in the configuration not exceed a certain percentage **f**, of the width of the base relation on which the index is defined, and (b) the width of a merged index not exceed the width of either of its parent indexes by a percentage higher than **p**. We note that such syntactic properties on the indexes can be enforced by the *MergePair* module. While this model is simple and requires little computational effort, it introduces new thresholds like **f** and **p** that need to be determined appropriately for a given system. Another drawback of the *No-Cost* model is that it cannot guarantee that the final merged configuration will satisfy the given cost-constraint.

3.5.2 External Cost Model

Much of the previous work in vertical partitioning and index selection relies on an external cost model for implementing the Cost function, i.e. they are not "in-sync" with the optimizer. An external cost model is attractive since, it can lead to a very inexpensive implementation of the Cost function. However, there are two reasons why an external cost model is not desirable. First, building an accurate external cost model to account for the cost of complex SQL queries is hard. Modern query processors use innovative techniques such as index intersection, and join techniques that use indexes, which make it hard to model externally. Second, since the query optimizer itself evolves over time, the cost model has to be updated and maintained to remain faithful to the query optimizer, which incurs a substantial software-engineering overhead.

3.5.3 Optimizer-Estimated Cost

The query optimizer component of a database system routinely predicts the cost of a query for a configuration C without actually executing the queries. Most database systems support interfaces for exposing this cost information, e.g. Showplan in Microsoft SQL Server, EXPLAIN in IBM/DB2. However, we are still faced with the problem that not all indexes in C exist in the database. Thus, we have to ensure that the query optimizer is able to predict Cost (W, C) even when one or more indexes in C are non-existent. Fortunately, the optimizer does not rely on physical existence of indexes, but merely on statistical information on the columns of the indexes. Such statistical information consists of a histogram on the column(s) of the indexes and density information. This statistical information can be gathered inexpensively using sampling, without compromising accuracy significantly [CMN98]. We refer to the statistical information for non-existing indexes as hypothetical (or "what-if") indexes. Therefore, when presented with a configuration C consisting of one or more hypothetical indexes, the optimizer can still predict Cost (W, C) by first constructing the necessary hypothetical indexes, and then optimizing the query. More details on this mechanism are available in [CN98].

For large workloads, obtaining the optimizer cost estimates can be expensive since the query optimizer must be invoked once per query. However, several techniques can be used to alleviate this problem. First, for a given configuration C, the cost needs to be obtained only for *relevant* queries in W (i.e. queries for which indexes in C are potentially relevant). Second, *workload compression* techniques can be used to reduce the number of optimizer invocations. The simplest form of workload compression is to detect (syntactically) identical queries in the workload and replace all identical queries by a single query (with an adjusted frequency). Another effective method of workload compression is to only consider the **k** most expensive queries in the workload such that a significant fraction of the total workload cost is covered by these queries. Finally, a search algorithm that uses the optimizer's cost estimates can be partially supplemented by an external cost model to reduce the number of optimizer invocations and the number of hypothetical indexes that must be built. Thus, before invoking the optimizer for evaluating Cost (W, C), the search algorithm may consult an external cost model for an estimated cost, and invoke the optimizer only if the external cost model predicts that the cost-constraint can be met. The exploration of more sophisticated external cost models to supplement optimizer cost estimates is part of our future work.

4. Experiments

The goal of the experiments is to evaluate each of the three components of index merging presented in Section 3: (a) MergePair (b) Search strategy and (c) Cost Evaluation strategy. In this section we present a set of experiments that shows the effectiveness of our proposed techniques for index merging.

4.1 Implementation

We have implemented the index merging algorithm as a client of Microsoft SQL Server 7.0. The implementation mirrors the architecture presented in Figure 1, i.e., consists of three modules (a) Search module (b) MergePair module and (c) Cost Evaluation Module. Modularizing the components allows us to easily substitute one implementation of a module with another and observe the difference in the behavior of the algorithm. We had implemented the interfaces in the server for optimizing a query for a hypothetical configuration in the context of our work on index selection. We refer the reader to [CN98] for further details on these interfaces.

4.2 Experimental Setup

4.2.1 Databases

We experimented with three databases: TPC-D 1GB, and two synthetically generated databases Synthetic1 and Synthetic2. Synthetic1 had 5 tables with the number of columns in the each table being varied between 5 and 25. Synthetic2 had 10 tables with the number of columns in each table being varied between 5 and 45. In both schemas, columns of varying width (between 4 and 128 bytes) were included. The total size of Synthetic1 was approximately

200MB and Synthetic2 was approximately 1.2 GB. The data values in each column were randomly generated based on a Zipfian distribution. The Zipf value for the distribution was picked randomly from the values 0, 1, 2, 3, 4. (0 implies uniform distribution, whereas 4 is highly skewed data).

4.2.2 Workloads

For each database, we generated two classes of workloads. The first class consists of randomly generated projection-only queries where indexes are predominantly used as covering indexes (see Section 1 for examples). The second class of workloads consisted of randomly generated complex queries (containing joins, aggregations etc.) using the automatic query generation tool Rags [S98]. In each class, we randomly generated two workloads with 30, and 50 queries respectively.

4.2.3 Initial Set of Indexes

The initial set of indexes determines the space of possible merged configurations proposed by the algorithms. We varied this parameter by considering initial configurations of size 5, 10, 15, 20, 25, 30 indexes. Indexes were selected using the Index Tuning Wizard [CNITW98] that is part of Microsoft SQL Server 7.0, which can be used to pick indexes for an individual query. We picked a query at random from the workload and created indexes recommended by the Index Tuning Wizard for optimizing the performance of that query. This process was repeated until the required number of indexes were generated.

4.3 Results

In this section we show through our experiments that:

- The greedy search strategy produces substantial reduction in storage.
- The optimizer-cost based strategy for cost evaluation is significantly better than the No-Cost model approach.
- The quality of index merging depends on the use of cost and index usage information in MergePair.
- The reduction in index maintenance cost due to index merging is substantial.

4.3.1 Comparison of Search Algorithms

(A) Quality of solution

In our first experiment we compare the quality of the solution produced by (a) *Exhaustive* algorithm which uses the optimizer-estimated cost, (b) Greedy search algorithm which using the optimizer-estimated cost (*Greedy-Cost-Opt*) and (c) Greedy search algorithm using the No-Cost model described in Section 3.5.1 (*Greedy-Cost-None*). We used **f** = 60%, and **p** = 25% for the No-Cost model since that yielded best results for Greedy-Cost-None. Due to the exponential increase in the number of merged configurations explored by the *Exhaustive* algorithm we present results for the initial number of indexes (N) = 5 for each database. We used the complex query workload consisting of 30 queries in each case. Figure 5 shows the percentage reduction in storage achieved by each algorithm. The greedy algorithm that uses optimizer-estimated cost information of queries to guide the search (Greedy-Cost-Opt) performs almost as well as the exhaustive algorithm in each case, yielding only less than 4% storage reduction compared to *Exhaustive* for the Synthetic2 database. The figure also shows that the greedy algorithm that uses no cost model (Greedy-Cost-None) suffers a significant drop in quality. This experiment shows that the greedy approach performs well when cost evaluation uses optimizer-estimated cost. We observed similar results when we varied the cost-constraints (see Section 3.1). However, we do not present these additional results due to lack of space.

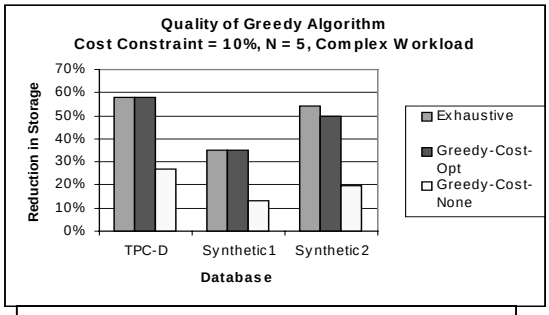

Figure 5. Quality of Greedy algorithm

(B) Running time

Figure 6 shows that the running time of the greedy algorithms is a small percentage of the running time of the *Exhaustive* algorithm. The figure also shows that Greedy-Cost-Opt incurs a moderate overhead in running time as compared to Greedy-Cost-None. We note that in most cases, Greedy-Cost-Opt explored a very small number of merged configurations in the inner loop (Steps 5-8 in Figure 4) of the algorithm. Thus the number of hypothetical indexes created and number of optimizer invocations were relatively small. We conclude that the increased running time of the Greedy-Cost-Opt algorithm is a price worth paying for the high quality recommendations that it yields.

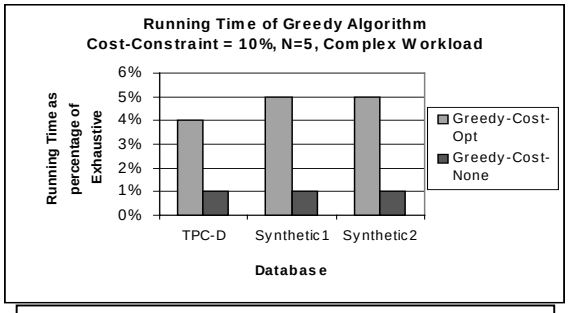

Figure 6. Comparison of Running Time

4.3.2 Comparison of MergePair procedures

In this experiment we compare the quality of solutions produced by the Greedy search algorithm when using the MergePair-Cost, MergePair-Syntactic and MergePair-Exhaustive procedures respectively (presented in Section 3.3). Due to the exponential nature of MergePair-Exhaustive, we present the results for the initial number of indexes (N) = 5, using the Greedy-Cost-Opt algorithm, for a cost constraint of

10%. Figure 7 shows that despite restricting itself to the space of index preserving merges, the MergePair-Cost procedure gives almost as much reduction in storage as MergePair-Exhaustive. This is due to the fact that it takes into account cost and index usage information. On the other hand, MergePair-Syntactic, which also uses index preserving merges but relies only on syntactic information of queries in the workload, is substantially worse in quality. This experiment shows that restricting the space of possible merges to index preserving merges works well when combined with query cost and index usage information.

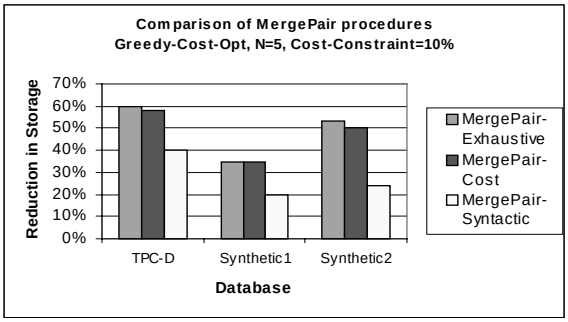

Figure 7. Comparison of alternative implementations of MergePair module.

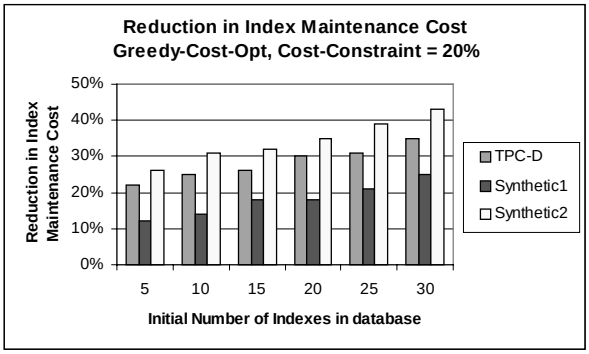

Figure 8. Reduction in cost of data insertions.

4.3.3 Comparison of Index Maintenance Cost

In our final experiment we demonstrate the reduction in index maintenance cost due to index merging. To study the effect of batch updates in a decision support environment, we inserted 1% of the tuples into the two largest tables in each database. We did this when the database consisted of (a) the initial configuration, and (b) the merged configuration produced by the Greedy-Cost-Opt algorithm. We repeated this for different number of indexes in the initial configuration. Figure 8 shows that index merging results in substantial savings in index maintenance cost for each database. This experiment shows that in addition to reducing the storage requirements, index merging can also help significantly reduce the batch insertion time in a typical decision support system.

5 Conclusion

In this paper, we have introduced the concept of index merging and identified the important components of the index merging problem. Index merging helps identify indexes that can significantly reduce storage, index maintenance and administrative costs in a database. Our experiments indicate that a greedy search strategy that uses optimizer-estimated cost of workload for evaluating the merit of a merged configuration (i.e., the *Greedy-Cost-Opt* algorithm), is an excellent algorithm for index merging.

References

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

[**C92**] Chu P., A Transaction Oriented Approach to Attribute Partitioning. Information Systems 17(4), 329-342, 1992.

[**CMN98**] Chaudhuri, S., Motwani, R., Narasayya, V., "Random Sampling for Histogram Construction: How Much Is Enough?". Proc. of ACM SIGMOD '98.

[**CN98**] Chaudhuri, S., Narasayya, V., AutoAdmin "What-If" Index Analysis Utility. Proc. of ACM SIGMOD '98.

[**CNITW98**] Chaudhuri, S., Narasayya, V., Index Tuning Wizard For Microsoft SQL Server. Microsoft White Paper. http://research.microsoft.com/db/AutoAdmin/

[**CN97**] Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server.". Proceedings of the 23rd VLDB Conference, Greece, 1997.

[**CY90**] Cornell D.W., Yu, P.S, A Vertical Partitioning Algorithm for Relational Databases. Proc. of ICDE'87.

[**DPP88**] De P., Park J.S., Pirkul H., An integrated model of record segmentation and access path selection for databases. Information Systems 13(1), 13-30, 1988.

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

[**FST88**] Finkelstein S, Schkolnick, 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 ICDE'97.

[**HN79**] Hammer M., Niamir B. A heuristic approach to attribute partitioning. Proc. of ACM SIGMOD'79.

[**HS75**] Hoffer J.A., Severance D.G., The use of cluster analysis in physical data base design. Proc. of VLDB'75.

[**M83**] March S.T., Techniques for structuring database records. ACM Computing Surveys. 15(1) 45-79, 1983.

[**NCWD84**] Navathe S., Ceri G., Wiederhold G., Dou J., Vertical Partitioning algorithms for database systems. ACM TODS, 9(4), 680-710, 1984.

[**NR89**] Navathe S., Ra M., Vertical Partitioning for Database Design: A Graphical Algorithm. ACM SIGMOD'89.

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

[**S98**] Slutz, D., Massive Stochastic Testing of SQL. Proc. of VLDB'98.

[**TPC93**] TPC. TPC Benchmark D. (Decision Support). Working Draft 6.0. August 1993.