Automatic Physical Design Tuning: Workload as a Sequence

Sanjay Agrawal
Microsoft Research
One Microsoft Way
Redmond, WA, USA
+1-(425) 705-3507
sagrawal@microsoft.com

Eric Chu*
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
+1-(608) 628-6941
ericc@cs.wisc.edu

Vivek Narasayya
Microsoft Research
One Microsoft Way
Redmond, WA, USA
+1-(425) 703-2616
viveknar@microsoft.com

ABSTRACT

The area of automatic selection of physical database design to optimize the performance of a relational database system based on a *workload* of SQL queries and updates has gained prominence in recent years. Major database vendors have released automated physical database design tools with the goal of reducing the total cost of ownership. An important assumption underlying these tools is that the workload is a *set* of SQL statements. In this paper, we show that being able to treat the workload as a *sequence*, i.e., exploiting the ordering of statements can significantly broaden the usage of such tools. We present scenarios where exploiting sequence information in the workload is crucial for performance tuning. We also propose techniques for addressing the technical challenges arising from treating the workload as a sequence. We evaluate the effectiveness of our techniques through experiments on Microsoft SQL Server.

1. INTRODUCTION

Database vendors such as IBM, Microsoft and Oracle offer automated physical design tuning tools. Database Tuning Advisor (DTA) in SQL Server [1], Design Advisor [16] in IBM DB2 and SQL Access Advisor [7] in Oracle 10g automate the task of finding the best *physical design structures* (e.g., indexes and materialized views) to optimize server performance. These tuning tools require a *workload* comprised of queries and updates to arrive at a physical design recommendation. All these tools are *set*-based – the workload is viewed as a set of statements and no ordering of statements is assumed during tuning.

The premise of this paper is that the ordering of statements can be important for performance tuning, specifically for physical database design. To illustrate this, we describe three scenarios in the context of physical database design where the set-based tuning approach falls short, and an alternative approach that exploits workload sequence information can lead to much superior workload performance.

Scenario 1: Data warehousing: "Query by day, update at night".

During the day there are multiple applications that issue complex queries against the warehouse. At night there is a batch window during which the warehouse data is updated, e.g., new data is inserted. Figure 1 below captures the ordering information in data warehouses.

Figure 1. Data warehousing scenario.

If we tune the workload that includes both queries and updates as a single set using a set-based approach, it is quite possible that we do not get any physical design structure to be recommended that benefit the workload as a whole. This is because, although such a tool may identify structures that speed up the queries (in the day), the update cost incurred (in the night) for the structures may far outweigh their benefit.

On the other hand, if we treat the workload as a sequence, we may recommend the following: create structures before the queries arrive and drop such structures before the updates arrive. Such a recommendation gives us the benefit of structures for queries but without the update overhead. If the benefit of such structures is greater than their creation cost, the data warehouse scenario can be optimized for performance as shown in Figure 2. Note that performance improvement arises from the fact that the structures incur no maintenance cost since they are dropped prior to updates .

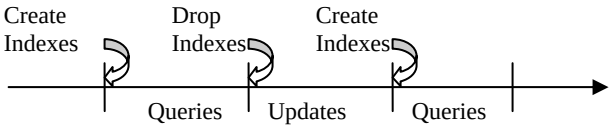

Figure 2. Optimized data warehousing scenario.

Observe that the obvious approach of breaking the workload into two workloads (the query workload and update workload respectively) and tuning each without awareness of the other can lead to sub-optimal performance. This is due to the fact that physical design recommendations for each workload can be very different and the cost of *transitioning* between the two recommendations can be significant. In the above example, indexes need to be created for the queries and dropped for the

Conference'06, June 26–29, 2006, Chicago, IL, USA.

*Work done when the author was visiting Microsoft Research.

updates. The creation and dropping of indexes can become very expensive at times (e.g., the drop of clustered index on a large table may internally lead to recreation of all non-clustered indexes on that table). Therefore, the cost of physical design transitions must be included in the analysis for achieving optimal performance. Similarly, a strategy that tunes physical design only for queries and ignores updates while tuning (thus there is no transition cost since the physical design does not need to change) can also be sub-optimal since the cost of updating physical design structures can be substantial.

Scenario 2: SQL applications that use transient tables.

```
// Step 1: create table
CREATE TABLE #t (multiple columns TYPES)
// Step 2: populate table
INSERT INTO #t
SELECT columns FROM Y WHERE column = value
//Step 3: use table in multiple queries
SELECT X.*,#t.*
FROM    X INNER JOIN #t ON X.CUSTID = # t.CUSTID
WHERE X. PRODUCTID = value
ORDER BY #t.PRICE DESC
//Step 4: drop table
DROP TABLE #t
```

Figure 3. Transient table usage scenario

Many applications use transient or temporary tables in the manner as illustrated in Figure 3. Again one can recommend structures on transient tables by leveraging the knowledge of how such tables are used. In this scenario, after Step 2 and before Step 3, one can create an appropriate index on the transient table to improve the performance of the SELECT statement. This kind of tuning can not be achieved by set-based tuning tools since recognizing the sequence of CREATE TABLE (this marks the start of life of transient tables), followed by INSERT then SELECT and finally DROP (this marks the end of life of transient tables) is important. For example, creation of the index just before Step 2 would *not* be optimal since the index would incur update cost but would yield no benefit in Step 2.

Scenario 3: Periodic data change on production server.

A common scenario in applications is where data pertaining to a certain time period is available on the production server for a specified duration. For example, the sales table may contain data for the current quarter. The data gets populated as follows. At the end of the current quarter, all rows from the table are deleted. Subsequently data for the new quarter is inserted into the table. New data gets added at the end of each day from different sales sources. Meanwhile, there are queries that run against this data during the entire time period. Figure 4 captures this scenario.

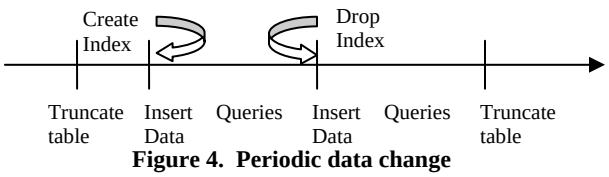

Figure 4. Periodic data change

Note that even though the set of queries and updates can remain the same, the same physical design may be totally ineffective for a significantly different data size and/or distribution. For example, non-clustered indexes may become ineffective for seeks if selectivity becomes large. Overheads of inserting rows and benefits of physical design structures are inherently tied to table sizes. A sequence-based approach in conjunction with additional database statistics that capture the dynamic nature of data, can make the better trade offs for suggesting create/drop of physical design structures to optimize performance as compared to a set-based approach.

The above scenarios highlight the fact that exploiting the order between statements can be crucial for improving performance. In practice, rarely is the workload either a single set or a single sequence of statements. A more general model of a workload is a *sequence of sets* of statements. Let us see how each of the above scenarios fits this model. For scenario 1 above, we may treat the workload as alternate sets of queries (during the day) and updates (during the night). Similarly for scenario 3, sets of queries and inserts alternate in the workload. Note that in both cases the set of queries alternate with the set of updates and this defines a *sequence*. Likewise each statement in scenario 2 can be viewed as a single statement set, the sets being ordered naturally as the order of steps above. In the rest of this paper, for simplicity of exposition, we assume that workload is a sequence of sets where each set is a single query/update. This makes it easier to understand the problem space and reason about our solution. In Section 7 we briefly discuss how our solution extends to the generalized model where the set contains multiple statements.

It is important to note that the output model of sequence-based tuning is different from set-based tuning solutions. Unlike a set-based approach where the output is a single SQL script with creates/drops of structures, for a sequence tuning tool the output contains create and drop of structures interleaved with the input workload. For example, in Scenario 2 above, the output would be "create index on transient table between the INSERT statement (Step 2) and the SELECT statement (Step 3)". Thus, implementing the recommendations may require changes in application code.

We summarize the key contributions of this paper below:

- We motivate the physical design tuning opportunities that arise by treating the workload as a sequence.
- We formally define the problem of physical design tuning for workload sequences (Section 2). Our goal is to add "create" and "drop" of physical design structures to the input sequence such that the overall performance of the generated sequence is maximized.
- We present an optimal solution to this problem by showing that the problem can modeled as finding the shortest path over a directed acyclic graph (DAG) constructed from the input (Section 3).
- We present two techniques *cost-based pruning* (Section 4) and *split and merge* (Section 5) that facilitate pruning of the search space.
- We describe a greedy heuristic (Section 6) that scales well for large workloads and many physical design structures.

Finally, we note that the techniques developed in the paper are general in the sense that they apply to any physical design structure (indexes, materialized views, etc.) that may be supported by the underlying database.

2. PROBLEM DEFINITION

2.1 Preliminaries

We model the ***workload*** as a *sequence* of SQL statements, i.e., SELECT, INSERT, DELETE and UPDATE statements. All the statements are ordered by monotonically increasing ID (a timestamp is an example). We represent a statement in a sequence as $\mathbf{S_k}$ where **k** denotes its ID. [S_1, S_2... S_N] denotes a sequence of N statements S_1 through S_N. The workload can be gathered using tracing tools (for example, Profiler tool in Microsoft SQL Server) that are available on today's database systems.

A ***physical design structure*** can be any access path supported by the database server, e.g., index, materialized view, multidimensional clustering of tables, etc.. A ***configuration*** is a valid set of physical design structures that can be realized in a database. A structure is considered ***relevant*** for a statement if it could potentially be used in an execution plan for answering the statement (even if the structure is not actually chosen by the query optimizer in the final plan).

We use the following notations in the paper. ***COST***(S, C) denotes the cost of executing statement S for configuration C. We rely on *optimizer estimated costs* and *what-if extensions* that are available in several commercially available database servers [1, 7, 16]. For the reasons described in [2], this allows our solution to be robust and scalable; we can try out numerous alternatives during search very efficiently without disrupting the normal database operations. ***TRANSITION-COST***(C_1,C_2) denotes the minimum cost of realizing configuration C_2 in the database starting from configuration C_1, i.e. cost of creating and dropping structures to get from configuration C_1 to configuration C_2. For example, suppose configuration C_1 contains a single index $\{I_1\}$ and C_2 contains a single index $\{I_2\}$. C_2 can be realized from C_1 by executing the following statements - CREATE INDEX I_2 followed by the statement *DROP* INDEX I_1. *TRANSITION-COST*(C_1,C_2) would be the cumulative cost of creating I_2 and dropping I_1. In general, this function could be provided as an input.

We represent the ***execution*** of a sequence [S_1, S_2... S_N] as [C_1, S_1, C_2, S_2… C_N, S_N, C_{N+1}] where C_i is the configuration that is realized in the database prior to executing S_i and C_{N+1} denotes the configuration after statement S_N is executed. Note that there is an implicit ordering between the configurations: $\forall i$ C_i is realized before C_{i+1}. Let C_0 denote the configuration prior to the sequence execution. Note that C_0 is an input and could be implicit (the current configuration in the database) or optionally specified through what-if interface [1]. We define the ***sequence execution cost*** of [C_1, S_1, C_2, S_2… C_N, S_N, C_{N+1}] as *TRANSITION-COST* (C_N, C_{N+1}) + $\sum_{k=1}^{N}$ (COST (S_k, C_k) + *TRANSITION-COST* (C_{k-1}, C_k)). Note that this includes the cost of changing the configurations during sequence execution through the *TRANSITION-COST* component.

2.2 Physical Design Problem for a Workload Sequence

Problem Statement: Given a database **D**, a sequence workload **W**=[S_1,S_2... S_N], initial configuration $\mathbf{C_0}$ and a storage bound **M**, find configurations $\mathbf{C_1,C_2,…C_{N+1}}$ such that storage requirement of C_i ($1 \le i \le N+1$) does not exceed **M,** and ***sequence execution cost*** of [C_1, S_1, C_2, S_2… C_N, S_N, C_{N+1}] is minimized.

There are few important points to observe about this problem formulation. First, the output of any solution to the above problem is itself another sequence where statements corresponding to create and drop of physical design structures (DDL statements) are inserted to the input sequence [S_1, S_2... S_N] such that configurations C_i for i=1 to N+1 are realized as above. Second, if the workload contains *inserts/updates/deletes*, the cost of updating the physical design structures are accounted for automatically as part of our optimization problem – this is captured as part of COST(S_i,C_i). Third, the cost of transitioning from one configuration to the next is also accounted for in the optimization problem – this is done via TRANSITION-COST(C_{i-1}, C_i). Finally, observe that the storage bound **M** is required to hold at *all* points in the sequence.

Our problem formulation is general enough to handle some other common constraints. Some important constraints include:

- Consider the case where the sequences are generated by individual applications (for example, Scenario 2 in the Introduction). An application's impact on the underlying databases physical design is limited to the duration when the corresponding workload sequence executes. This can be incorporated by constraining $\mathbf{C_{N+1}=C_0}$, which ensures that the physical design is restored to the same state as it was prior to the workload sequence. We refer to this as *transparency* constraint. A variation of this includes the case where $\mathbf{C_{N+1}}$ is constrained to be an explicitly provided configuration (need not be C_0) by the user.
- Physical design changes are allowed only at specific points in the sequence. For example, in Scenario 1 in the introduction, the user (e.g., DBA) may allow physical design changes to happen *only* at two specific points during the day. More generally, in the sequence [S_1..S_p..S_q..S_N] if we want to allow configuration changes only between statements S_p and S_q, this can be specified via the constraint: For $1 \le i < p$, $\mathbf{C_i=C_{i-1}}$ and for $q < i \le N+1$, $\mathbf{C_i = C_q}$. Thus we only need find configurations C_p... C_q.
- Only allow physical design changes that complete within a user specified cost bound. This can be represented as *TRANSITION-COST*(C_{i-1}, C_i) $\le$ t for all $1 \le i \le N+1$.

We now define the set of physical design structures over which we perform the sequence optimization described above. A naïve way to generate such a set from the input sequence is to union the set of all relevant structures for each statement in the input sequence. However as detailed in [1, 2, 15, 16] the space of relevant structures for a single statement can become prohibitively expensive to compute, let alone for the entire sequence. There are a number of existing techniques that allow us to efficiently generate a much smaller set of structures for the purposes of physical database design tuning. One example is the IBM DB2 [15] approach where the optimizer recommends its own structures for a statement. An alternative approach is discussed in [2] where very good structures are generated using "candidate selection" and "merging" steps keeping the query optimizer in the loop. We note that the focus of this paper is orthogonal to the specific method used for the purpose. We refer to the set of structures generated using such a technique as "candidate structures". For

the rest of paper, we assume that a set of candidate structures can be obtained, given the input workload.

Finally, we comment briefly on the search space for the optimization problem. If we are provided as input a sequence of N statements, and there are M candidate structures, the number of possible *configurations* is 2^M, since each subset of structures defines a unique configuration. Hence we have a total of $2^{M*(N+1)}$ choices as we need to find N+1 configurations $C_1,C_2,\ldots C_{N+1}$. We contrast this with the set-based tuning problem where there are "only" 2^M possible configurations. Thus, we can view the set-based tuning problem as a constrained version of sequence tuning problem, where physical design changes are only allowed at the beginning of the sequence.

3. OPTIMAL ALGORITHM

In this section, we describe an algorithm to generate an optimal solution to the physical design problem for workload sequences (defined in Section 2.2). The key observation is that given an instance of the problem, we can construct a graph such that the shortest path in that graph is an optimal solution for that instance.

We illustrate our solution through a simple example. The input workload is a sequence of N SQL statements. The set of candidate structures is a single index (referred to as I). The goal is to find N+1 configurations ($C_1 \ldots C_{N+1}$) as described in Section 2.2. We observe the following. In this example, there are two possible configurations: (1) The empty configuration {} and (2) The configuration {I}. Thus, with any statement S_i, there are two possible costs: COST (S_i,{}) and COST (S_i,{I}).

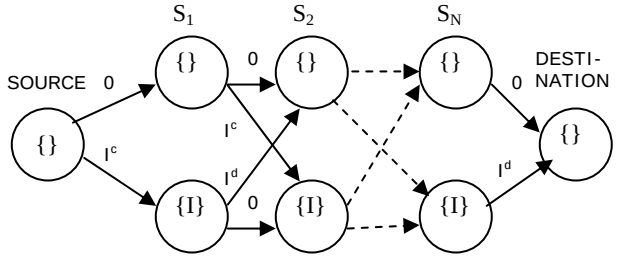

Figure 5. Graph for single index, N statements

Figure 5 shows the graph that is generated for single index, N-statement case. The graph is constructed as follows:

1. For every statement in the input sequence and for every possible configuration generated from the input set of structures we generate a node, i.e., a node **n** represents a (statement S, configuration C) pair with a node cost = COST (S,C). The two nodes SOURCE and DESTINATION representing the initial and final configuration are added to the graph and have a node cost of 0. SOURCE precedes the first statement in the input sequence and DESTINATION succeeds the last statement in the input sequence. Thus in our example, there are 2*N+2 nodes in the graph since there are only two configurations {} or {I} possible for each statement.
2. The graph has N+2 ***stages***; a stage for each statement, SOURCE and DESTINATION. We refer to SOURCE as 0-th stage and DESTINATION as (N+1)-th stage.
3. The edges in the graph are directed, and exist only between nodes in stage k and nodes in stage k+1 ($0 \le k \le N$). Let edge **e**=(n_1, n_2) represent the edge from node n_1=(*,C) to node n_2=(*,C'); * denotes that it could be any statement. Then cost of **e**=TRANSITION-COST(C,C'). In our running example, there are 4*N edges and the costs of the edges can be: (i) 0 when there is no change in configuration between nodes that define the edge (ii) cost of creating the index (denoted in the figure by (I^c)) when transitioning from {} to {I} and (iii) cost of dropping the index (denoted by I^d) when transitioning from {I} to {}.
4. If the final configuration denoted by C_{N+1} is constrained to be the same as initial configuration denoted by C_0 or some other user provided configuration, we assign the appropriate edge cost between the nodes in stage N and DESTINATION node. However if we do not have any constraints on the final configuration, we assign a cost of 0 to all the edges between the nodes in the stage N and DESTINATION node. In our example, we assign edge costs of 0 between all nodes in S_N and DESTINATION and C_{N+1} will be the same as C_N.

Once the graph is constructed as above, any path from SOURCE to DESTINATION represents a valid sequence execution. Note that the path cost includes the cost of nodes as well as edges. The optimal output sequence is the **shortest path** in this graph. The equivalence between the shortest path in the graph above and sequence execution cost is straightforward as the node costs represent COST (S_k, C_k) and edge costs represent TRANSITION-COST (C_{k-1}, C_k). We note the following properties of this graph. (1) The cost of nodes and edges are non negative. (2) The shortest path in this graph can be computed very efficiently using single–source shortest path technique for DAGs described in [6] in linear complexity as number of both edges and nodes are O (N); the intuition behind the linear complexity is that each edge needs to be examined exactly once to arrive at the solution.

Generalizing the graph to N-statement sequence and M structures to generate an optimal solution is conceptually straightforward. In each stage 1 through N, there are 2^M nodes, each representing a configuration. This is because each subset of input structures defines a configuration. We refer to the solution that enumerates all 2^M configurations exhaustively at each stage as EXHAUSTIVE. It is important to note that the graph has O ($N*2^M$) nodes and O ($N*2^{2M}$) edges, i.e., the graph is exponential in the number of input structures. Though the shortest path can be solved in linear complexity of number of nodes and edges as above, these however are exponential in number of structures.

Thus, if the input workload has only a small number of candidate structures, we can apply EXHAUSTIVE to get an optimal solution. However, it becomes impractical for large workloads that can typically have hundreds or more candidate structures. Note also that each node represents a (statement, configuration) pair and has an associated node cost. Thus, we have to compute COST(S,C) corresponding to that node. Using the costing technique described in Section 2.1 may involve invoking the query optimizer, which can also contribute to making graph construction expensive.

In the next two sections we discuss two techniques that can improve the efficiency of the above algorithm by reducing the number of nodes in the graph significantly. The first technique (discussed in Section 4) is an optimality preserving *cost-based* pruning. The second technique (discussed in Section 5) is

effective for large workloads where different statements touch different tables/columns of the database.

4. COST-BASED PRUNING

This optimization allows us to prune configurations at a given stage in the input sequence while preserving optimality. It relies on the observation that in many cases we can *efficiently* compute a lower bound on the cost of a statement for a given configuration. The intuition behind this technique is that we can leverage optimal solutions of individual structures as described in Section 3 to significantly prune nodes at various stages in the graph that would otherwise be generated by EXHAUSTIVE above. We first describe the pruning technique under the assumption that the benefits of the structures are independent. We subsequently describe how the technique can be leveraged when the structures interact, i.e., the benefits of the structures are not independent; [2] provides a detailed description of such interactions. One example of such an interaction is when both indexes used for a Merge Join are sorted on join key, join proceeds much faster compared to the case when only one index is sorted on the join key.

We define the following. SPS (***s***) for a physical design structure ***s*** refers to the shortest path solution that we get by optimizing the entire sequence for ***s*** alone using the algorithm presented in Section 3. Let $[b_1, S_1, b_2, S_2 \ldots b_N, S_N, b_{N+1}]$ represent the solution SPS(s) where b_i represents configurations. Let $[C_1,S_1,C_2,S_2 \ldots C_N,S_N,C_{N+1}]$ denote the solution where C_i represents configurations that we get by running EXHAUSTIVE over the graph that has *all* the configurations (the power set of all structures). If there are no interactions across structures (i.e., for every statement, the benefits of structures are independent of the presence of other structures) then the following claim holds.

Claim: $\boldsymbol{s} \notin b_k$ in SPS(***s***) $\Rightarrow \boldsymbol{s} \notin C_k$ for every structure s and stage k (k = 1 to N).

Proof: We prove it by contradiction. Assume that there exists a structure ***s*** where *k* is the first stage such that $\boldsymbol{s} \in C_k$ but $\boldsymbol{s} \notin b_k$. All the costs are non negative and $C_0=b_0$ (both SPS and EXHAUSTIVE are solved with the same initial configuration). We use the following notations:

- I^c as the creation cost for ***s***.
- I^d as the drop cost for ***s***.
- $p_1 = COST(S_k, C_k)$
- $p_2 = COST(S_k, C_k - \{\boldsymbol{s}\})$ where $C_k - \{\boldsymbol{s}\}$ is the configuration one gets by removing s from C_k.
- $q_1 = COST(S_k, \{\boldsymbol{s}\})$
- $q_2 = COST(S_k, \{\})$

Since benefits are independent $q_1-p_1=q_2-p_2 \Rightarrow q_1-q_2=p_1-p_2$. We enumerate all possible cases as following.

- $\boldsymbol{s} \in C_{k-1}$. Then $p_1<I^d+p_2$ must hold as EXHAUSTIVE would not pick C_k otherwise. If $\boldsymbol{s} \in b_{k-1}$, then $q_1>I^d+q_2 \Rightarrow q_1-q_2> I^d \Rightarrow p_1-p_2>I^d$ results in a contradiction. The other case $\boldsymbol{s} \notin b_{k-1}$ can not happen as *k* is the first stage where violation occurs.
- $\boldsymbol{s} \notin C_{k-1}$. Then $I^c+p_1<p_2$ must hold. If $\boldsymbol{s} \in b_{k-1}$, then $q_1>I^d+q_2 \Rightarrow q_1 > q_2 \Rightarrow p_1> p_2$ results in a contradiction. If $\boldsymbol{s} \notin b_{k-1}$, then $q_1 > q_2 \Rightarrow p_1> p_2$ again results in a contradiction.

We also note that $C_{N+1}= C_N$ in our original optimization problem (or equals C_0 in *transparency* constrained problem) and the proof holds for stage N+1. This allows us to eliminate structures (and configurations that contain these) at a given stage as follows. We run the SPS for all structures and analyze their respective solutions at each stage. For a given stage *k*, we construct a set of structures $R=\{s \mid s \in b_k \text{ in SPS(s)}\}$. Note that every subset c of R defines a unique configuration and is added to the graph as node (S_k,c) if configuration c obeys the storage bound.

Let's apply this to following example. Assume that the input sequence has 4 statements $[S_1,S_2,S_3,S_4]$. Also for statement S_i index I_i (i= 1, 2, 3 and 4) and no other index is relevant and the drop cost of every index is 0. Using EXHAUSTIVE we will enumerate all 2^4=16 configurations to find the optimal solution. By looking at optimal solutions on a *per structure* basis, i.e., $SPS(I_1)=[\{\},S_1,\{\},S_2,\{\},S_3,\{\},S_4,\{\}]$, $SPS(I_2)=[\{\},S_1,\{I_2\},S_2,\{\},S_3,\{\},S_4,\{\}]$ and so on, we know that only $\mathbf{I_i}$ is present in stage **i**. Thus we can construct the "reduced" graph with 5 configurations as shown in Figure 6 (the unlabelled edges represent a cost of 0); the solution we get on this "reduced" graph is optimal.

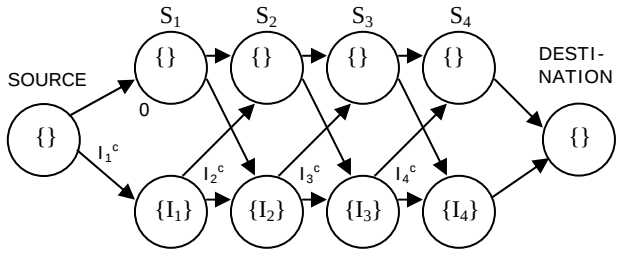

Figure 6. Graph with cost-based pruning

When the benefits of structures are not independent, in SPS(***s***) we use the lower bound cost of any configuration that contains ***s*** for statement S_k as $COST(S_k,\{\boldsymbol{s}\})$. We note that the claim above remains true when the assignment of the costs to individual structures is done as described above. The argument is similar as for the independent structure case. We reuse the notation from the proof above. We assume the optimizer is well behaved; for a query the addition of a structure to a configuration can never increase the cost of the statement. For the case where $\boldsymbol{s} \in C_k$ and $\boldsymbol{s} \in b_{k-1}$, we get the following: $p_1-p_2< q_1-q_2 \Rightarrow (p_1-q_1) + (q_2-p_2)<0$. However $p_1 \geq q_1$ due to the way we assign costs above and $q_2 \geq p_2$ for queries. This leads to a contradiction. The proof for other cases is similar and is omitted.

The effectiveness of cost-based pruning depends on how accurately and efficiently we can determine the lower bound of costs. This itself is a hard problem as we do not want to enumerate all configurations containing $\{\boldsymbol{s}\}$ to arrive at the lower bounds. A trivial lower bound is the minimal cost of a query under any physical design. This can be computed very efficiently by probing the optimizer once [3]. There are other techniques (e.g., [5] discusses how to derive costs of configurations using atomic configurations) that can be leveraged to get better lower bounds efficiently. The updates/inserts/deletes can be handled by "splitting" the update into a query part which identifies the specific rows that need to be updated and the actual update part. We omit details due to lack of space.

We expect the cost-based pruning technique to work well when $\boldsymbol{s} \in b_k$ in SPS(***s***) for *few* values of *k*. This typically happens when there are a significant number of updates/inserts/deletes in the input sequence or in the presence of the *transparency* constraint. In our experiments, see Section 8.2.2 we demonstrate the

effectiveness of this pruning technique where it results in orders of magnitude reduction in number of nodes in the graph.

5. EXPLOITING DISJOINT SEQUENCES

As we saw in Section 3, the efficiency of our solution depends on the number of input structures which in turn depends on the statements in the sequence. In general, workloads can be large. These could be trace files collected by tracing the server activity and could be over a period of days with thousands of statements that touch *different* parts of the database. In this section we present a technique that leverages the fact that groups of statements access different parts of data, to reduce the search space significantly.

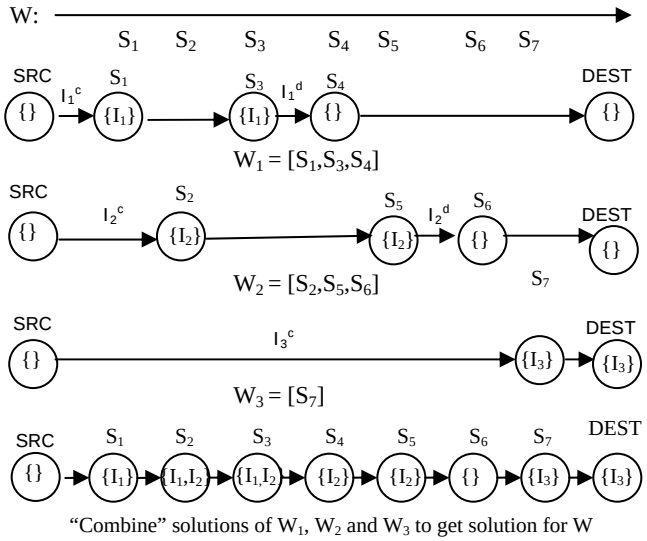

Figure 7. Disjoint sequences

We motivate the technique through an example. Figure 7 shows an input sequence workload W = [S_1, S_2... S_7]. Suppose structure I_1 is relevant for S_1, S_3 and S_4 only (these statements reference table A only), I_2 is relevant for S_2, S_5 and S_6 (these statements refer table B only) and I_3 is relevant for S_7 only (it is on table C). S_4 and S_6 are updates on the tables A and B respectively. Applying EXHAUSTIVE on S with input structure set $\{I_1,I_2,I_3\}$ would enumerate $2^3 = 8$ configurations. If we apply cost-based reduction as discussed in Section 4 we can reduce the number of configurations significantly. However at S_2 and S_3, we still need to consider the configurations corresponding to all the subsets of $\{I_1,I_2\}$ *a priori* as both I_1 and I_2 are present in stages 2 and 3. Note that I_1 and I_2 are on *different* tables in the database and are relevant for *different* statements in the sequence. The interesting question is whether we can avoid generating such configurations up front and if possible at all. It turns out that for the above sequence we can do much better as described below.

We define the notion of **disjoint sequences**. Two sequences X and Y are said to be disjoint if (1) X and Y do not share any statements and (b) no statement in X shares any relevant physical design structure with any statement in Y. This implies that given a physical design structure ***s***, all the relevant statements corresponding to ***s*** are present in *only* one such sequence, i.e., its impact is limited to one sequence. On applying this to our example in Figure 7, whether we create or drop index I_2 on table B cannot impact the cost of statements in W_1 (recall that S_1, S_3 and S_4 are on table A). We note that disjoint sequences may be interleaved with respect to how these occur in the original sequence (W_1 and W_2 in Figure 7 are interleaved as S_2 in W_2 starts before W_1 ends).

It may be tempting to suggest that we can "break" the input sequence into a set of disjoint sequences, solve each one independently as the choice of configurations in one disjoint sequence does not impact choices for other sequences and "combine" the results to get the globally optimal solution. In the absence of storage violations this strategy is indeed optimal and can lead to much better search performance than the alternative approach that tunes the input as a single sequence. The efficiency comes from significant reduction in number of configurations (and hence nodes) that needs to be generated for the graph. In our current example, each disjoint sequence in our example is solved for just one index (W_i for I_i) and configurations like $\{I_1,I_2\}$ are never considered. However if there are storage violations, then the above strategy does not work as the resulting solution is not valid. In our example if we do not have sufficient storage for both I_1 and I_2, by using the above strategy we get $\{I_1,I_2\}$ at S_2 and S_3 as shown in Figure 7 which is not valid. An interesting question is how can we generate a valid solution efficiently in the latter?

We present two operators that mirror the ideas above. The first operator **Split** described in 5.1 takes an input a sequence and the relevant set of structures and splits it into a set of disjoint sequences. The second operator **Merge** (in 5.2) takes as input a set of disjoint sequences and their respective solutions and combines these to generate a valid solution for the sequence derived from superimposing input set of sequences. The two operators can be combined to efficiently generate close to optimal solutions.

5.1 Split

The algorithm to achieve this is straightforward. For every structure we know the set of statements that are relevant by looking at the syntactic structure of statements. Consequently, for every statement, the set of relevant structures is known. Now, the split is achieved by doing a **transitive closure** over the statements as follows. (1) Start with each statement as a separate sequence. With every sequence associate its relevant set of structures. (2) If two sequences share any structure, combine them into one sequence (union of the statements) and union their set of structures. (3) Continue step 2 till no more sequences can be combined. At the end, our input sequence is split into a set of disjoint sequences that neither share any statements nor any structures. In the example above, we split the input sequence W into 3 disjoint sequences W_1 = [S_1,S_3,S_4], W_2 = [S_2,S_5,S_6] and W_3 = [S_7] with relevant structure sets $\{I_1\}$, $\{I_2\}$ and $\{I_3\}$ respectively.

5.2 Merge

The input to Merge is a set of disjoint sequences and their respective solutions. The output of Merge is a solution that obeys storage constraint and is defined over all the statements from input disjoint sequences. Let $\mathbf{P}=\{p_1,\ldots p_m\}$ represent the set of solutions where p_i corresponds to the solution for W_i ($1\le i\le m$) that is provided as input to Merge. Merge proceeds in two steps.

Step 1: From the input set of solutions **P**, a solution $\mathbf{P_u}$ is constructed by performing a union of configurations of $\mathbf{p_i}$ ($1\le i\le m$)

at each stage. In our current example as shown in Figure 8, we get $\mathbf{P_{1,3}}$ by performing the union of solutions of W_1 and W_3. $\mathbf{P_u}$ represents the union of solutions of disjoint sequences W_1, W_2 and W_3. Note that the union involves the superimposition of the individual statements in the input sequences as well as the union of configurations at each stage.

Step 2: If all the configurations that are part of $\mathbf{P_u}$ obey storage bound then $\mathbf{P_u}$ is optimal and is the output of Merge. However if there are configurations in $\mathbf{P_u}$ (as shown in Figure 8) that violate storage bound, then $\mathbf{P_u}$ is not valid. In that case we make local changes to $\mathbf{P_u}$ to make it a valid solution. The intuition behind local optimization is based on the following observation. The solutions in the regions where storage bounds do not get violated remain optimal as long as the *preceding* and *following* configurations remains unchanged. This is guaranteed by the shortest path algorithm. If the cost of the violating ranges is small with respect to the entire sequence execution cost, then applying local optimizations to the violating ranges only to generate a valid solution can result in *nearly* optimal solutions.

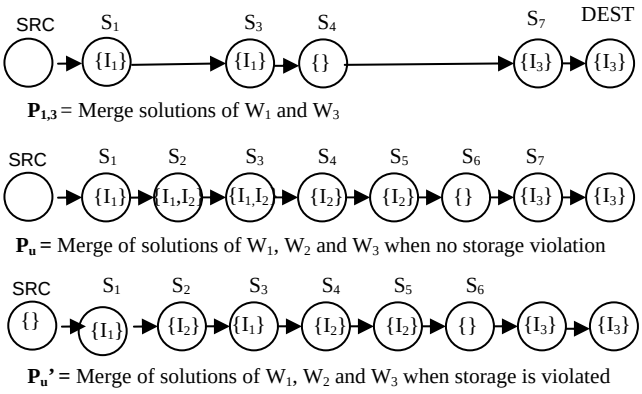

Figure 8. Merge of disjoint sequences

Let us see how we apply this to our current example. Using the notation from 2.2, $C_1=\{I_1\}$, $C_2=C_3=\{I_1,I_2\}$, $C_4=\{I_2\}$ and so on in $\mathbf{P_u}$ (obtained from step 1 above). Assume that we have sufficient storage for only one index. Then $\mathbf{P_u}$ is not a valid solution as C_2 and C_3 (= $\{I_1,I_2\}$) violate storage bound. By leveraging the observation above, we isolate the violating sequence which in this case is $[S_2,S_3]$. For the remaining parts of the sequence ($[S_1]$ and $[S_4,S_5,S_6,S_7]$), there are no storage violations in their respective solutions. If we locally optimize $[S_2,S_3]$ to get a solution that respects the storage bound with SOURCE=$\{I_1\}$ and DESTINATION=$\{I_2\}$, we preserve the optimality of solutions of $[S_1]$ and $[S_4,S_5,S_6,S_7]$. Here, if the benefit of indexes I_1 and I_2 far outweigh their create costs, then optimizing $[S_2,S_3]$ with SOURCE=$\{I_1\}$ and DESTINATION=$\{I_2\}$ results in $[\{I_2\},S_2,\{I_1\},S_3,\{I_2\}]$. Next we combine the locally optimal solutions of $[S_1]$ (from $\mathbf{P_u}$), $[S_2,S_3]$ (computed locally as above) and $[S_4,S_5,S_6,S_7]$ (from $\mathbf{P_u}$) to get $\mathbf{P_u'}$ which is a valid solution.

The sequence execution cost of $\mathbf{P_u}$ is a lower bound on the cost that we can get for any solution that obeys the storage bound and therefore for the optimal solution. If the cost of the solution we get by applying Merge (=$\mathbf{P_u'}$) is close to the sequence execution cost of $\mathbf{P_u}$ then we have a nearly optimal solution at hand.

In general one may apply Merge in different ways over the disjoint sequences to build alternative search schemes (for example, one may Merge disjoint sequences pair wise in a greedy manner). This is made possible as Merge preserves disjointedness, e.g., if W_1, W_2 and W_3 are mutually disjoint, then output sequence after Merge of W_1 and W_3 is disjoint with respect to W_2.

In our experiments, see Section 8.2.4, we used a simple strategy where we generated a valid solution for the entire sequence by first applying Split, and then Merge over the solutions of *all* disjoint sequences. We observed that even this simple strategy often led to an order of magnitude speed up compared to a strategy that blindly treated the input as a single sequence. Also the resulting solution after Merge was close to optimal (within 3%) even under very limited storage bounds. We note that in data warehouse scenarios (see 8.2.4 for TPCH experiments that simulates this) the applicability of this becomes limited as almost all queries reference the fact table and we do not get multiple disjoint sequences. However if the same server hosts multiple such data warehouses (and workloads) then this technique can still be used very effectively.

6. GREEDY HEURISTIC

As described in Section 3, the number of configurations, and hence nodes and edges in our graph, is exponential in the number of candidate physical design structures for the workload. The EXHAUSTIVE approach (Section 3) can therefore infeasible in practice where a workload can often have a large number of candidate structures. Observe that the above problem exists even in the approach where the workload is treated as a *set*, e.g., [3,5, 15]. We note that previous work in the context of set-based workloads (e.g., [3, 5]) have developed techniques to deal with the combinatorial explosion that results in the number of configurations that need to be considered. These techniques typically use a greedy heuristic to search through the space of configurations instead of looking at the entire (exponential) space. In this section, we describe how to adapt such a greedy heuristic for the case of workload as a sequence. We refer to our algorithm as GREEDY-SEQ. The GREEDY-SEQ algorithm uses a function we refer to as UnionPair. We first describe this function in Section 6.1, and present the overall algorithm in Section 6.2.

6.1 UnionPair

UnionPair takes as input two solutions denoted by $p_1=[a_1,S_1,\ldots,a_N,S_N,a_{N+1}]$ and $p_2=[b_1,S_1,\ldots b_N,S_N,b_{N+1}]$. Observe that the inputs are both solutions for the *same* sequence[1] $[S_1, \ldots, S_N]$ but the configurations in each solution are over a disjoint space of physical design structures. *UnionPair* generates a new solution for the sequence as described below. Initially, a graph is constructed that has all the nodes (and edges) from the two input solutions p_1 and p_2. At each stage k in the graph, additional configurations (as described below) are generated from configurations a_k and b_k and corresponding nodes and edges are added to the graph. The output of *UnionPair* is the shortest path solution in the graph thus generated.

Next we discuss how to generate the additional configurations at each stage. Intuitively, at each stage k in the graph, i.e., (for S_k) we are looking for the best configuration (we refer to this as d_k) we can generate from the structures in ($a_k \cup b_k$) that satisfies the

[1] Observe also that unlike UnionPair, *Merge* (Section 5.2) is defined over solutions to *disjoint* sequences.

storage constraint. In general, the number of configurations we need to consider are exponential in $|(a_k \cup b_k)|$. We observe however, that in many common cases the configuration $(a_k \cup b_k)$ itself is optimal at stage *k* (e.g., if S_k is a query and $(a_k \cup b_k)$ obeys the storage constraint) since $(a_k \cup b_k)$ preserves the benefits of both a_k and b_k. In general however, $(a_k \cup b_k)$ can have worse update characteristics compared to a_k and b_k. Note that since we preserve the original configurations a_k and b_k in the generated graph, the update characteristics of $(a_k \cup b_k)$ get accounted for automatically in our strategy (for example if $(a_k \cup b_k)$ has a high update overhead, the shortest path output may ignore it and pick a_k instead). Figure 9 represents the generated graph for the above input pair assuming $d_k = (a_k \cup b_k)$.

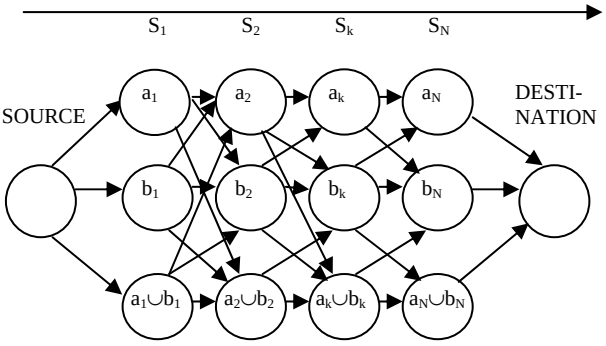

Figure 9. Generated graph for UnionPair

In general, it may be important to introduce other configurations in addition to $(a_k \cup b_k)$ as well. In such cases, we can take advantage of previous techniques developed (e.g., in [3,5]) to generate *nearly* optimal configurations very efficiently. We omit further details due to lack of space.

6.2 The GREEDY-SEQ Algorithm

1. For every structure in the set **S**=$\{s_1,..s_M\}$, find the optimal solution using the graph formulation as described in Section 3. At this point, we have a set of solutions **P** for individual structures. Let **P**= $\{p_1,\ldots p_M\}$ and $\mathbf{p_i}=[a_{i1},S_1\ldots S_N,a_{iN+1}]$.
2. Let C be the set of all configurations over all p_i's.
3. Run a greedy search over **P** as follows.
   a. Let **r** = $[c_1, S_1, \ldots c_N, S_N, c_{N+1}]$ represent the least cost solution in **P**. **P=P-{r}**. Let C = C ∪ $\{c_1, \ldots c_{N+1}\}$
   b. Pick an element **s** from **P** such that **t**=**UnionPair**(**r**,**s**) has the minimal sequence execution cost for among all elements of **P** and sequence execution cost of **t** is less than that of **r**. If no such element exists go to step 4. **P=P-{s}**. **P=P** ∪{**t**}. Go to a.
4. Generate the graph with all the configurations in **C** at each stage. Run the shortest path over this graph and return the solution.

Figure 10. The GREEDY-SEQ algorithm

We use *UnionPair* to build our greedy solution. The individual steps of GREEDY-SEQ are described in Figure 10.

Let's take a look at how GREEDY-SEQ works in the following example. Assume that the input sequence has 8 statements $[S_1,S_2,S_3,S_4,S_5,S_6,S_7,S_8]$, input set of structures is $\{I_1,I_2,I_3,I_4\}$ and storage is infinite. Also for statements S_i and S_{4+i} index I_i ($1 \le i \le 4$) and no other index is relevant and that the benefit of every index for the relevant statement is greater than its creation (and drop) cost. Also assume that the cost using index $I_i < I_{i+1}$ ($1 \le i \le 3$). The exhaustive approach in conjunction with cost-based pruning would still lead to 2^4 configurations at stage 4. Using GREEDY-SEQ in Step 4 above we have 8 configurations only and we still get the optimal solution as it *lazily* generates the configurations ($\{I_1,I_2\},\{I_1,I_2,I_3\}$ and $\{I_1,I_2,I_3,I_4\}$) that significantly impact the over all quality of recommendation. In Step 3, we look at a few extra configurations (because of *UnionPair*) but the overall number of generated configurations is typically much smaller than EXHAUSTIVE. In Section 8.2.3, we present results that demonstrate the effectiveness of GREEDY-SEQ where it results in close to optimal solutions and with significantly better performance compared to EXHAUSTIVE.

While GREEDY-SEQ appears to work well in practice, it is important to note that it can in general be sub-optimal. The following example demonstrates such a case. Table 1 shows the cost of 4 statements in the sequence for various indexes. Assume that the storage available is 100 MB and that I_1 requires 80 MB, I_2 and I_3 require 40 MB each. GREEDY-SEQ returns $[\{I_1\},S_1,\{I_1\},S_2,\{I_1\},S_3,\{I_1\},S_4,\{I_1\}]$ while optimal solution is $[\{I_2,I_3\},S_1,\{I_2,I_3\},S_2,\{I_2,I_3\},S_3,\{I_2,I_3\},S_4,\{I_2,I_3\}]$.

Table 1. Sub-optimality of GREEDY-SEQ

Sequence / Index	S_1	S_2	S_3	S_4	Total Cost
Initial Cost	100	100	100	100	400
I_1	50	100	100	50	300
I_2	70	70	100	70	310
I_3	70	100	70	70	310

The two main reasons for sub-optimality of GREEDY-SEQ are storage constraints and interactions across various physical design structures. We note there are already effective greedy search techniques that result in very good and efficient solutions in the context of set-based workload to overcome the limitations mentioned above. For example, one such technique discussed in [5] advocates generating configurations with up to a certain size exhaustively and proceeding greedily there after. The other [15] uses measures like benefit per unit store instead of pure benefit to mirror knapsack like approaches. We note that these techniques can be integrated easily with GREEDY-SEQ.

Let's analyze the complexity of GREEDY-SEQ discussed above for a **N**-statement sequence and **M** structures. The graph for each structure is same as 3.1 and has O (N) edges and nodes. Step 1 requires O(N*M) time. Step 3 can be repeated at most **M** times (In each invocation an element, i.e., two paths in **P** get merged and subsequently removed from the set and the merged path gets added). Since we only retain shortest path solutions in **P**, an element in **P** always has O (N) edges and nodes. This allows us to solve step 3 in O ($N*M^2$) time. In step 4, C has O(M*N) configurations as we generate at most M solutions in step 3 and

each solution has O(N) configurations; hence number of nodes in step 4 is $O(M*N^2)$ and edges is $O(M^2*N^3)$. Therefore GREEDY-SEQ runs in $O(M^2*N^3)$. However in practice, we found number of nodes in step 4 to be O(M*N) as step 3 led to O(M) configurations resulting in $O(N*M^2)$ running time. We validate our analysis above through experiments in Section 8.2.3.

7. DISCUSSION

7.1 End to End Solution

We briefly outline a possible end to end solution based on the ideas discussed in the paper. Figure 11 shows the architecture of our solution. We *Split* the input sequences into disjoint sequences (Section 5.1) based on the candidates (Section 2.2 describes how to get candidates). We apply the optimality preserving cost-based pruning (Section 4) on each sequence. Subsequently we tune each sequence separately using EXHAUSTIVE (Section 3) or GREEDY-SEQ (Section 6). Finally we *Merge* (Section 5.2) the results of disjoint sequences to get the over all solution. We note that the various techniques discussed in the paper can be put together in different ways based on quality and performance requirements. For example, one may want to use the EXHAUSTIVE strategy and not apply the *Split* and *Merge* operations if quality is the driving factor and enumerating all the configurations is not prohibiting.

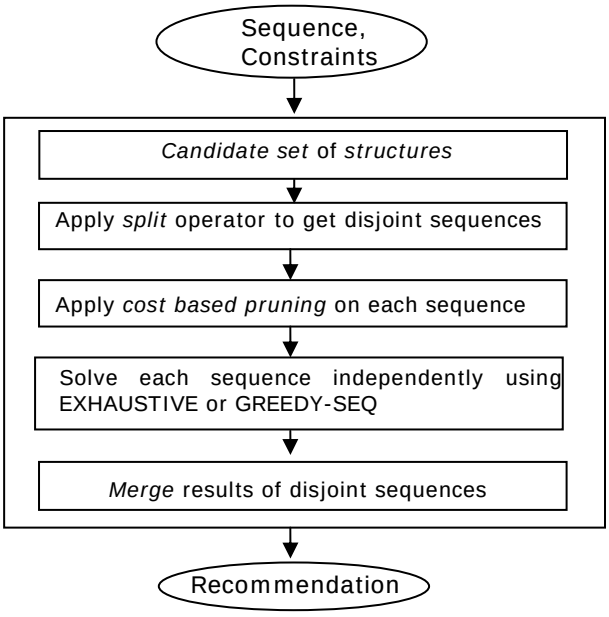

Figure 11. Architecture of our solution

7.2 Extending to Sequence of Sets

Extending our techniques when the workload is treated as a sequence of sets of statements is straightforward. In the graph formulation discussed in Section 3 each stage represents a set of statements rather than individual statements and each node represents a (set, configuration) pair. The configurations selected optimize the performance of a set. There are known techniques [5] that we can leverage to find such configurations efficiently. Here the physical design changes occur at the set boundaries instead of possibly at each statement.

8. EXPERIMENTS

We have implemented the algorithms presented in this paper via a prototype that extend the Database Tuning Advisor [1] in Microsoft SQL Server 2005. In our experiments we demonstrate the following:

- In the presence of updates and/or at low storage bounds, sequence-based approach leads to much better quality recommendations than a set-based approach.
- Cost-based pruning technique discussed in Section 4 results in significant pruning in number of nodes that need to be generated in the graph.
- The quality of recommendations from our greedy approach discussed in Section 6 is close to that achieved by an exhaustive solution that enumerates all configurations. The performance of greedy scales linearly with the workload size in practice and quadratically with the number of structures.
- Our split and merge approach for exploiting disjoint sequences (Section 5) results in close to optimal quality solutions and is much more efficient than the alternative that treats the input as a single sequence on "real" workloads.

8.1 Experimental Setup

The experiments were performed on a HP workstation with 2.8 GHz CPU with hyper-threading and 1 GB RAM on a commercially available database server. All the databases used in the experiment were stored locally on 80 GB hard disk. We used the available what-if APIs on the server to simulate the "create" and "drop" costs of various structures. This was further used to simulate the cost of transitions between various configurations.

We use the following workloads in our experiments to demonstrate the effectiveness of our techniques.

- TPCH-1-*n* consists of the first *n* queries from TPC-H 22 query benchmark [14]. TPCH-1-*n*-M-*m* is same as TPCH-1-*n* except the queries of TPCH-1-*n* are repeated *m* times.
- TPCH-1-*n*-U-*k*-MID consists of TPCH-1-*n* and updates to LINEITEM table that increases the #rows by k%. Updates are in the middle of the workload. TPCH-1-*n*-U-*k*-END same as TPCH-1-*n*-U-*k*-MID except updates are present at the end.
- Two "real" workloads and databases, WKLD1 and WKLD2 on databases DB1 (~250 MB with about 1000 tables) and DB2 (~500 MB with about 250 tables) respectively. WKLD1 has a mix of 1000 insert/update/delete/select statements. WKLD2 contains complex stored procedures as statements.

8.2 Experimental Results

8.2.1 Effectiveness of sequence-based approach

Here we compared the quality of sequence and set-based tuning approaches. We used TPCH1G database and workloads TPCH-1-22, TPCH-1-22-U-10-MID and TPCH-1-22-U-10-END at two storage bounds (i) a low storage bound of 1.2 GB which allows for maximum 20% of data size for redundant structures and (ii) a high storage bound of 3 GB which allows for an extra 2 GB space for redundant structures. Table 2 below compares the quality of the two approaches. The % improvement is relative to the optimal output of set-based approach and thus getting any further

improvement was not an easy task. When there were no updates and high available storage (TPCH-1-22 at 3GB), both approaches led to similar results as expected. However when updates were added to the workload (or at low storage), sequence-based approach could further improve the solutions of set-based approach by another 16-28%. In the low storage cases with updates, set-based approach led to almost no improvement over the initial physical design while sequence-based approach improved it by about 25%. The running time of the two techniques was very similar as (a) the overhead of computing shortest path itself was negligible and (b) both techniques looked at almost the same set of structures and configurations. This shows that sequence-based tuning leads to much superior recommendations compared to set-based tuning approach in the presence of updates and/or low limited storage; it performs the required trade off between storage requirements, update costs and benefits of structures dynamically to decide not only the specific structures to create and drop but also where in the sequence to do the required creates and drops.

Table 2. Sequence vs. Set-based tuning comparison

Workload	% improvement compared to set - based approach at 1.2 GB	% improvement compared to set-based approach at 3 GB
TPCH-1-22	19%	0%
TPCH-1-22-U-10-MID	22%	16%
TPCH-1-22-U-10-END	25%	28%

8.2.2 Effectiveness of cost-based pruning technique.

In this experiment, we demonstrate how the cost-based pruning technique results in significant reduction in number of nodes that are added to the graph. First, we compared this to the exhaustive strategy EXHAUSTIVE discussed in Section 3 that considers all possible configurations that obeys the provided storage bound. EXHAUSTIVE-BEN refers to our strategy where cost-based pruning is applied at every stage on the graph. We used the database TPCH1G and smaller workloads TPCH-1-5 and TPCH-1-5-M-10 for this experiment as it was infeasible to enumerate all possible configurations for the entire TPCH-1-22 workload for EXHAUSTIVE. We evaluated the effectiveness of EXHAUSTIVE-BEN at two storage bounds (i) a low storage bound of 1.2GB and (ii) a high storage bound of 3 GB as described in 8.2.1 above.

Table 3. Effectiveness of cost-based pruning

Workload	**% Reduction in # nodes compared to EXHAUSTIVE at 1.2GB**	**% Reduction in # nodes compared to EXHAUSTIVE at 3GB**
TPCH-1-5	99.4%	99.8%
TPCH-1-5-M-10	94.3%	97.7%

Table 3 shows two orders of magnitude reduction in #nodes for EXHAUSTIVE-BEN compared to EXHAUSTIVE. It is interesting to note that when storage bound was increased from 1.2GB to 3GB, we got relatively more reduction. That is because the number of configurations in EXHAUSTIVE increased much more rapidly than EXHAUSTIVE-BEN. We also note that as the benefit of individual structures increases (we achieved this by making 10 copies of the queries and there are no updates in the workload), the pruning achieved by EXHAUSTIVE-BEN decreases. However the reduction was still significant compared to EXHAUSTIVE (two orders of magnitude).

In the second part, we eliminated EXHAUSTIVE from consideration. We evaluated the effectiveness of EXHAUSTIVE-BEN for workloads with different update characteristics. We used TPCH1G database and TPCH-1-22, TPCH-1-22-U-*k*-MID (k=1, 10 and 20), TPCH-1-22-U-1-END as workloads for this part.

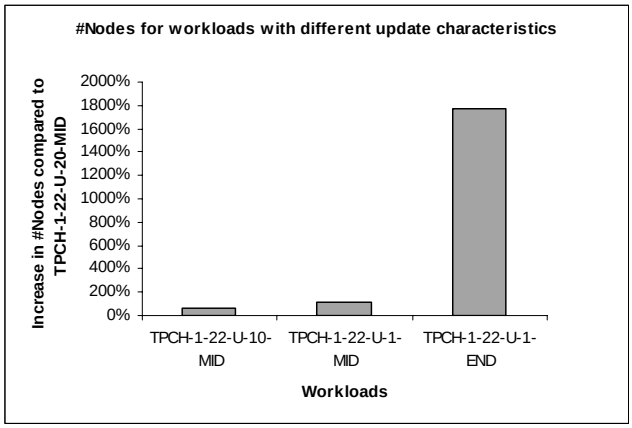

Figure 12. Variation of #Nodes with updates

Figure 12 shows the increase in #nodes for update workloads compared to the workload TPCH-1-22-U-20-MID that has the maximum update overhead. We observed that as the updates became more expensive the reduction in #nodes increased significantly. The #nodes for TPCH-1-22-U-1-MID was more than twice the #nodes for TPCH-1-22-U-20-MID. We also observed that the reduction depended very strongly as to where the updates occurred in the workload. When the updates were at the end (TPCH-1-22-U-1-END) the #nodes was more than 18 times the #nodes in TPCH-1-22-U-20-MID, i.e., EXHAUSTIVE-BEN did not result in significant pruning. This is expected as the shortest path algorithm can choose to drop the structures that incur heavy update overheads *just* before the updates and not before that. Consequently, the results for TPCH-1-22-U-1-END and TPCH-1-22 (no updates) were similar. This shows that EXHAUSTIVE-BEN is most effective in the presence of updates especially when these are interspersed with the queries in the workload (workloads with MID).

8.2.3 Effectiveness of GREEDY-SEQ.

In this experiment, we evaluated the effectiveness of our greedy technique GREEDY-SEQ discussed in Section 6. In the first part of the experiment, we compared our approach to EXHAUSTIVE-BEN. We used the database TPCH1GB and the following workloads: TPCH-1-3, TPCH-1-5-M-5 and TPCH-1-22.

Table 4 compares the quality and performance characteristics of GREEDY-SEQ normalized with respect to EXHAUSTIVE-BEN for various workloads. For the smaller workload TPCH-1-3 both techniques resulted in similar results as both look at good configurations yet GREEDY-SEQ ran almost twice as fast as EXHAUSTIVE-BEN. For the larger workload TPCH-1-5-M-5 the performance of EXHAUSTIVE-BEN degraded much faster. It looked at about a 100 times the number of configurations as compared to what GREEDY-SEQ generated. Consequently

GREEDY-SEQ ran two orders of magnitude faster than EXHAUSTIVE-BEN while the quality degradation was about 2% compared to latter. For TPCH-1-22, we had to terminate EXHAUSTIVE-BEN after 24 hours as the #structures was more than 50 and despite all the optimizations the #nodes generated was order of hundreds of thousands rendering EXHAUSTIVE-BEN impractical. On the other hand, GREEDY-SEQ returned with a solution with ~56% improvement compared to pre-optimization workload cost in less than an hour.

Table 4. GREEDY-SEQ and EXHAUSTIVE-BEN comparison

Workload	% reduction in running time of GREEDY-SEQ compared to EXHAUSTIVE-BEN	% reduction in quality of GREEDY-SEQ compared to EXHAUSTIVE-BEN
TPCH-1-3	50%	<1%
TPCH-1-5-M-5	98.4%	2.3%
TPCH-1-22	EXHAUSTIVE-BEN was terminated after 24 hours	Not available

Here we measured the scalability of GREEDY-SEQ with varying number of queries in workload and candidate structures.

Performance of GREEDY-SEQ with number of queries: Here the number of queries was increased while number of structures was kept constant. We achieved this by making multiple copies of TPCH-1-5 workload (TPCH-1-5-M-*m* for m=10 through 70 in steps of 10). Figure 13 shows the running time normalized to that for TPCH-1-5-M-10. We observe that the running time increased almost linearly with the number of queries in the input.

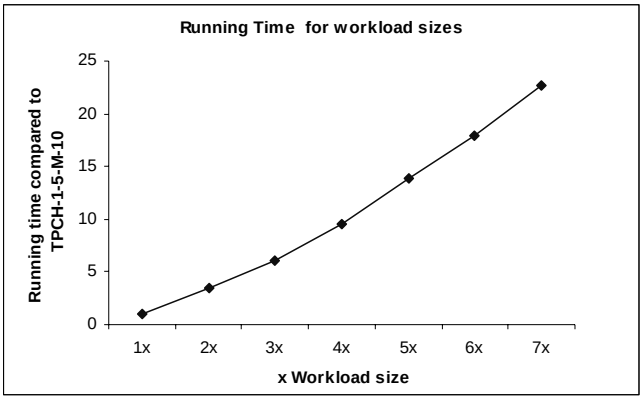

Figure 13. GREEDY-SEQ running time for workload sizes

Performance of GREEDY-SEQ with number of structures: Here we used TPCH-1-3-M-8, TPCH-1-5-M-5, TPCH-1-8-M-3 and TPCH-1-22. These workloads have similar number of queries (close to 25 each) but very different number of structures. Figure 14 shows the increase in running time for different workloads compared to TPCH-1-3-M-8. We observed that the running time increased almost quadratically with the number of structures. The number of structures applicable for TPCH-1-22 was about 5 times that for the TPCH-1-3-M-8 and resulted in about 16x increase of running time. Similarly for TPCH-1-8-M-3, the number of structures was about 3.5 times that for TPCH-1-3-M-8 and increase in running time was about 11 times. For TPCH-1-5-M-5, the number of structures was about 1.5x that for TPCH-1-3-M-8 and increase was close to 3x.

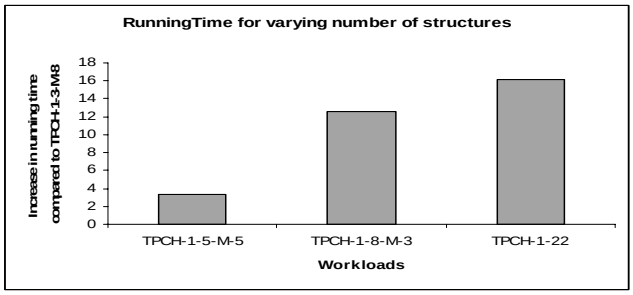

Figure 14. GREEDY-SEQ running time with # structures

This shows that the variation of running time of GREEDY-SEQ is consistent with our analysis presented in Section 6.

8.2.4 Effectiveness of split and merge.

Here we demonstrate the effectiveness of split and merge technique discussed in Section 5. We compared the two alternatives (1) SPMR which was the split and merge and (2) WO-SPMR where the entire input sequence was considered as single sequence during optimization. We used GREEDY-SEQ search technique for each disjoint sequence.

Table 5. Split and merge quality and performance

Workload	% reduction in running time compared to WO-SPMR	% cost difference compared to unconstrained optimal cost	% cost difference compared to WO-SPMR
TPCH-1-22	<0.1%	0%	0%
WKLD1	89.9%	0%	0%
WKLD1-LOW	71.4%	3.4%	3.0%
WKLD2	83%	0%	0%

We used the following in this experiment: TPCH-1-22 on TPCH1G, WKLD1 on DB1, and WKLD2 on DB2. The motivation for using "real" workloads and databases was to demonstrate the effectiveness of SPMR in practice. We allow storage to be at most 3 times the data size that is more typical in practice. We also ran WKLD1 where extra storage is only 20% data size that we refer to as WKLD1-LOW; this allowed us to measure the quality degradation of SPMR (how far it was from WO-SPMR) when storage was extremely limited leading to more storage violations during merge of individual solutions.

Table 5 summarizes our results. We observed that when storage was not a limiting factor (cases except WKLD1-LOW) SPMR resulted in the same quality as WO-SPMR. For TPCH-1-22, the quality and performance of the two techniques was the same; split resulted in a *single* sequence and its overhead was negligible (<0.1%). The input sequence was split into multiple disjoint sequences for WKLD1 and WKLD2 (6 and 10 respectively) which resulted in much better performance compared to WO-SPMR (5 times speed up). The case when storage was extremely low (WKLD1-LOW), we still got answers close to WO-SPMR; the parts of sequence where storage violation occurred after merging individual solutions were relatively small and sub-optimality in that region did not impact the over all solution by much (about 3% degradation even for extremely limited storage). This shows that SPMR is very effective in practice for getting very good results.

9. RELATED WORK

Automated physical design tuning solutions are offered by major database vendors such as IBM, Microsoft and Oracle. IBM offers DB2 Design Advisor [16] that recommends indexes, materialized query tables (i.e., materialized views), shared nothing partitions and multidimensional clustering of tables. Microsoft SQL Server has Database Tuning Advisor [1] that allows for integrated selection of indexes, indexed views and horizontal partitions. Oracle 10G includes an Oracle Tuning Pack that has a SQL Access Advisor [7, 10] that deals with selection of indexes and materialized views. Rao et al. [11] uses a workload to recommend data partitions. [9] talks about index selection in an adaptive fashion. There is also work in the area of monitoring databases workload and to use the information for physical design tuning. Chaudhuri et al. [4] describes a framework that can be used to gather such a workload very efficiently. Sattler et al. [13] details a system QUIET that provides continuous query driven index selection. While the workload gathered using such approaches can be provided as input to our approach directly, these techniques are all set-based and hence are unable to exploit sequence information in the workload (e.g., the scenarios described in the introduction).

There is a significant amount of literature that deals with how to determine interesting physical design structures, i.e., candidates for a given workload. Bruno et al. [3] and Valentin et al. [15] advocate the use of query optimizer to generate candidates. The work in [2,5] adopts an approach where it uses query optimizer in conjunction with various strategies like candidate selection, index and view merging on the entire workload to arrive at candidates. Rozen [12] presents a framework to choose physical design automatically where the space of materialized views is restricted to single table aggregation views with group by. There have been several papers, e.g., [8] on selection of materialized views in the context of OLAP/Data Cube. Typically these assume a space of aggregation views over dimensions. We view these as complementary to our work. The main focus of our work is how to perform sequence-based tuning using the candidates efficiently once the set is identified. In fact instead of reinventing the wheel we leverage such strategies in our approach. Also some of these techniques [3, 5] advocate the use of approximations during optimization to make the physical design selection very efficient. Chaudhuri et al. [5] mentions atomic configurations and details how to exploit these to derive costs of other configurations in order to reduce optimizer calls; [3] uses bounding techniques to estimate a cost for a configuration at times. Again these techniques complement our work and can be incorporated into our approach to enable more efficient solutions.

10. CONCLUSION

In this paper we motivate the need for exploiting sequence information in the workload for the purpose of physical design tuning. We define the problem formally and present an optimal approach to tune sequences by mapping it to shortest path problem. We present two pruning techniques as well as an efficient greedy heuristic that is effective in practice.

11. ACKNOWLEDGMENTS

We thank Christian König for his insightful comments on the paper.

12. REFERENCES

[1] Agrawal, S., Chaudhuri S., Kollar L., Marathe A., Narasayya V., and Syamala M. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of VLDB* (2004). 1110-1121.

[2] Agrawal, S., Narasayya, V., and Yang, B. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *Proceedings of ACM SIGMOD* (2004). 359-370.

[3] Bruno N., and Chaudhuri S. Automatic Physical Design Tuning: A Relaxation Based Approach. In *Proceedings of ACM SIGMOD* (2005). 227-238.

[4] Chaudhuri, S., König, A., and Narasayya, V. SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proceedings of ICDE* (2004). 473-485.

[5] Chaudhuri, S., and Narasayya, V. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. *In Proceedings of VLDB* (1997). 146-155..

[6] Cormen, T., Leiserson C., and Rivest R. Introduction to Algorithms. MIT Press Edition. Twenty-third printing, 1999. 536-537.

[7] Dageville B., Das D., Dias K., Yagoub K., Zait M., and Ziauddin M. Automatic SQL Tuning in Oracle 10g. In *Proceedings of VLDB* (2004).1098-1109.

[8] Gupta H., Harinarayan V., Rajaramana A., and Ullman J.D. Index Selection for OLAP. In *Proceedings of ICDE* (1997). 208-219.

[9] Hammer M, and Chan A. Index selection in a self-adaptive data base management system. In *Proceedings of ACM SIGMOD* (1976).1-8.

[10] Oracle Tuning Pack. *www.oracle.com/technology/products/manageability/database/pdf/ds/tuning_pack_ds_10gr1.pdf.*

[11] Rao J., Zhang C., Meggido N., and Lohman G. Automating Physical Database Design in a Parallel Database. In *Proceedings of ACM SIGMOD* (2002). 558-569.

[12] Rozen S. Automating Physical Database Design: An extensible approach, PhD Thesis, New York Univ. (1993).

[13] Sattler K.,Geist I., and Schallen E. QUIET: Continuous Query-driven Index Tuning. In *Proceedings of VLDB* (2003). 1129-1132.

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

[15] Valentin, G., Zuliani, M., Zilio, D., and Lohman, G. DB2 Advisor: An Optimizer That is Smart Enough to Recommend Its Own Indexes. In *Proceedings of ICDE* (2000).101-110.

[16] Zilio, D., Rao J., Lightstone S., Lohman G., Storm A., Garcia-Arellano C., and Fadden S. DB2 Design Advisor. Integrated Automatic Physical Database Design. In *Proceedings of VLDB* (2004). 1087-1097.