Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications*

Karl Schnaitter
UC Santa Cruz
karlsch@ucsc.edu

Neoklis Polyzotis
UC Santa Cruz
alkis@ucsc.edu

Lise Getoor
Univ. of Maryland
getoor@cs.umd.edu

ABSTRACT

One of the key tasks of a database administrator is to optimize the set of materialized indices with respect to the current workload. To aid administrators in this challenging task, commercial DBMSs provide advisors that recommend a set of indices based on a sample workload. It is left for the administrator to decide which of the recommended indices to materialize and when. This decision requires some knowledge of how the indices benefit the workload, which may be difficult to understand if there are any dependencies or interactions among indices. Unfortunately, advisors do not provide this crucial information as part of the recommendation.

Motivated by this shortcoming, we propose a framework and associated tools that can help an administrator understand the interactions within the recommended set of indices. We formalize the notion of index interactions and develop a novel algorithm to identify the interaction relationships that exist within a set of indices. We present experimental results with a prototype implementation over IBM DB2 that demonstrate the efficiency of our approach. We also describe two new database tuning tools that utilize information about index interactions. The first tool visualizes interactions based on a partitioning of the index-set into non-interacting subsets, and the second tool computes a schedule that materializes the indices over several maintenance windows with maximal overall benefit. In both cases, we provide strong analytical results showing that index interactions can enable enhanced functionality.

1. INTRODUCTION

One of the fundamental tasks that a database administrator needs to perform is tuning the indices in the physical design. This is a difficult optimization problem, since typically the goal is to maximize the benefit of the materialized indices with respect to an expected workload, under constraints on resources [2, 4].

In order to assist in this process, commercial systems offer automated index advisors. An index advisor takes as input the expected workload and the constraints, and generates a recommended set of indices; this is referred to as a *recommended configuration*. Typically, the recommended configuration is also accompanied by statistics such as the total estimated benefit on the expected workload, and a breakdown of the benefit per index or per query.

While the returned statistics provide a glimpse of the benefit of the recommended configuration, they do not give a complete picture. Basically, they ignore the key issue of index interaction, which is crucial in understanding the cost/benefit trade-off of the recommended indices. Informally, an index a interacts with an index b if the benefit of a is affected by the presence of b and vice-versa. Interactions heavily affect the usage of indices in optimized query plans and are thus crucial in understanding how the configuration will affect the performance of the database system. Moreover, knowledge of the interactions can help the administrator to make informed decisions about which indices to actually materialize.

We provide a simple example that illustrates the significance of index interactions. In our example, the recommended configuration contains three indices a, b, c that are all useful for some subset of the workload. Suppose that b and c are not used individually in optimal plans but they bring a great deal of benefit when they are employed together, e.g., using index intersection. This is an example of a positive interaction between indices. Also, suppose that b and c together can be used in lieu of a for certain queries, albeit with lower benefit compared to a. This is an example of a negative interaction, since the presence of an index (in this case, a) precludes the usage of other indices in an optimal plan (in this case, b and c). Knowledge of these dependencies helps the administrator to better understand how the three indices affect query performance. For instance, it becomes obvious that materializing b on its own will not bring much benefit, since it interacts positively with c. Or, the administrator may decide to not materialize a, if the difference in performance from its substitution with b or c is within some tolerable threshold.

The importance of index interactions has also been emphasized in the context of tools for automated index selection [1, 2, 3, 6, 7, 10, 17, 18]. Interactions affect the computation of index benefit which in turn complicates significantly the search for a good configuration [6]. Previous studies have attempted to work around this issue either by modeling interactions under specific conditions for the query optimizer, or by making heuristic assumptions about the characteristics of interactions. Clearly, index selection tools can benefit from a systematic methodology that can characterize index interactions without making limiting assumptions about the properties of the interactions or the optimizer.

Motivated by these observations, we study the problem of characterizing interactions within a given set of indices and with respect to a specific workload. This setting matches directly the aforementioned scenario where a system administrator examines the output

*This work was supported in part by an award from the UC-SC/LANL Institute for Scalable Scientific Data Management and by an IBM Faculty Development Award.

VLDB '09, August 24-28, 2009, Lyon, France

of an index advisor tool. We first formalize the notion of index interaction in this context, and then define two useful problems: computing the degree of interaction between all pairs of indices, and determining the pairs of indices whose degree of interaction exceeds a threshold. Both problems are important in helping the administrator understand the dependencies between indices in a recommended configuration. Moreover, a methodology that solves these problems can be integrated into index selection tools in order to avoid the use of heuristic assumptions about index interactions.

A precise characterization of index interactions is straightforward to derive if the workload cost is known for all possible index configurations. Following prior work on index selection [7, 10], we can evaluate query cost with what-if optimization. This approach ensures consistency with the behavior of the query optimizer, but what-if optimization is unfortunately time-consuming. Even with heuristics that make more efficient use of the optimizer [5, 14], it can be prohibitively expensive to evaluate the workload cost under all subsets of even a moderately sized index configuration.

In order to enable a more efficient analysis, we develop novel algorithms that leverage the inherent structure of the optimizer's cost model and avoid enumerating all subsets of the configuration. Our algorithms have the added feature that they can provide approximate information about index interactions at any point during their execution, and thus they can operate under a time budget for analyzing interactions. This feature is important for the integration of the algorithms in tuning tools.

We present an experimental study of the proposed algorithms using a prototype implementation of our methodology on top of IBM DB2. Our prototype uses the DB2 Design Advisor to obtain a recommended configuration for a given workload and then proceeds to analyze the interactions within the configuration. Experimental results on the TPC-H workload validate the effectiveness of our algorithms in efficiently identifying index interactions even for complex configurations.

Finally, we propose two novel database tuning tools that incorporate information about index interactions. The first tool provides a rich visualization of index interactions. The visualization format leads to a natural partitioning of the indices into subsets that do not interact. We formalize the properties of this partitioning and show how it may be computed efficiently. The second tool uses knowledge of index interactions to schedule the materialization of new indices. The goal is to maximize the observed benefit over time, but we find that generating the optimal schedule is computationally hard. Fortunately, knowledge of index interactions may be exploited to formulate an intelligent scheduling heuristic, and we demonstrate the robustness of our approach with a formal analysis. Overall, these two concrete applications demonstrate the usefulness of index interactions in enabling new tools for database tuning.

2. PRELIMINARIES

Basic Concepts. We assume we are given a workload $\mathcal{W}$ of query and update statements over some relational database. We use q to denote a statement in $\mathcal{W}$.

We are also given a set of indices $\mathcal{S}$ that represents a possible configuration for optimizing the processing of $\mathcal{W}$. We do not make specific assumptions about the origin of $\mathcal{S}$, e.g., it can be generated by an index advisor tool, or specified by the database administrator. At a high level, our goal is to characterize the interactions between indices in $\mathcal{S}$ with respect to the statements in $\mathcal{W}$.

For any index configuration $X \subseteq \mathcal{S}$ and query $q \in \mathcal{W}$, we use $optplan_q(X)$ to denote the optimal execution plan for q that only uses indices in X. In practice, $optplan_q(X)$ can be retrieved directly from the system's query optimizer by employing what-if optimization (which is supported in all commercial systems). We interpret $optplan_q(X)$ as an opaque object with two abstract properties. The first property is simply the optimal plan cost, which is denoted $cost_q(X)$. The other property is the subset of available indices that are used in the optimal plan, which is denoted $used_q(X)$. Hence, $used_q(X) \subseteq X$ by definition, and $X - used_q(X)$ may be nonempty if some available indices are not used by the optimizer.

The existence of a cost function leads naturally to the definition of index benefit, as follows.

DEFINITION 2.1. *Given index-sets $X, Y \subseteq \mathcal{S}$ and a statement $q \in \mathcal{W}$, the benefit of X with respect to Y and q is defined as* $benefit_q(X, Y) = cost_q(Y) - cost_q(X \cup Y)$.

It is also useful to consider the cost and benefit metrics expressed over the complete workload. We thus define $benefit(X, Y)$ and $cost(X)$ as the summation of the per-statement metrics over all statements in $\mathcal{W}$.

Index Interactions. As stated earlier, we are interested in characterizing the interactions between indices in $\mathcal{S}$ in terms of benefit. Loosely speaking, two indices a and b interact when their benefits are not independent. This can happen, for instance, when a overlaps with b and thus the optimizer can substitute one for the other in a physical plan. This constitutes a type of negative interaction, since the presence of one index reduces the benefit of the other. Another case is when a and b are used together in the same physical plan, e.g., in an index intersection. This constitutes a type of positive interaction, where the presence of one index increases the benefit of the other.

Formally, we introduce a metric $doi_q(a, b, X)$, termed the *degree of interaction*, that captures how strongly a and b interact in the processing of q, assuming that the hypothetical index-set $X \subseteq \mathcal{S}$ is materialized. Before proceeding with the definition of $doi_q(a, b, X)$, we make an important observation for the case of update statements. We assume that an update q can be broken into two components: a select shell q_{sel} (which is a query) that retrieves the tuples to be updated, and an update shell q_{upd} that performs the actual update and also updates any affected materialized indices. Clearly, index interactions can exist within the select shell q_{sel} and our model accounts for them directly. On the other hand, an interaction within q_{upd} or between an index in q_{upd} and an index in q_{sel} would imply that the presence of an index affects the update cost of another. It is not clear whether this type of interaction is meaningful (or even common) in practice, and in addition it introduces another layer of complexity in the characterization of index interactions. As a reasonable compromise, our model captures index interactions only in the select shell of update statements.

We are now ready to define the degree of interaction metric that we use in our work.

DEFINITION 2.2. *Let a, b be indices in $\mathcal{S}$, and $X \subseteq \mathcal{S}$ be an index-set such that $X \cap \{a, b\} = \emptyset$. For a query statement q, the degree of interaction $doi_q(a, b, X)$ is defined as:*

$$doi_q(a, b, X) = \frac{|benefit_q(\{a\}, X) - benefit_q(\{a\}, X \cup \{b\})|}{\min\{cost_q(X \cup M) \mid M \subseteq \{a, b\}\}}.$$

For an update statement q, $doi_q(a, b, X)$ is defined as:

$$doi_q(a, b, X) = doi_{q_{sel}}(a, b, X).$$

Observe that the computation of $doi_q(a, b, X)$ involves solely query statements, and hence it is possible to substitute each update statement q with its select shell q_{sel} without modifying the computed

interactions. Thus, *for the rest of the paper we restrict $\mathcal{W}$ to contain only query statements.* We stress again, however, that this assumption is without loss of generality.

The degree of interaction is essentially expressed as a fraction of benefit difference to query cost. More concretely, the numerator captures the difference in the benefit of a when b is added to the physical design. A large difference implies a strong interaction between a and b and vice versa, and we can thus consider this expression as an absolute metric for the degree of interaction between a and b. The denominator relates the absolute metric to the cost of processing q using X and a, b. We employ the minimum cost under the four possible index sets in order to obtain a conservative metric for the amount of interaction. Thus, an absolute difference of 100 cost units is more important if the minimum query cost is 10 units compared to a minimum query cost of 1000 units.

The above definition can be used to derive a natural interaction metric that is independent from the choice of X.

DEFINITION 2.3. *Given a statement $q \in \mathcal{W}$ and indices $a, b \in \mathcal{S}$, the degree of interaction between a and b with respect to q is defined as $doi_q(a,b) = \max\{doi_q(a,b,X) \mid X \subseteq \mathcal{S}\}$.*

This metric exhibits symmetry, as $doi_q(a,b) = doi_q(b,a)$. However, $doi_q(a,b)$ is not a true "distance" metric, since the triangle inequality does not hold and $doi_q(a,a)$ is positive whenever $benefit_q(a,X) > 0$ for at least one $X \subseteq \mathcal{S}$. Another point is that $doi_q(a,b)$ does not reveal the type of interaction between a and b, i.e., negative, positive, or mixed, since we are primarily interested in the magnitude of the interaction. We note that it is possible to define alternative metrics that capture the maximum negative and maximum positive degree of interaction respectively between two indices. These metrics do not change the complexity of the overall problem, and they require a straightforward extension of the techniques that we present later.

Problem Statements. The degree of interaction between any pair of indices $a, b \in \mathcal{S}$ is crucial in understanding the effect of $\mathcal{S}$ on the processing of statements in $\mathcal{W}$. We thus arrive at the following fundamental problem that we address in our work:

Degree of Interaction Problem (DOIP) *Given an index-set $\mathcal{S}$ and a workload $\mathcal{W}$, compute $doi_q(a,b)$ for any two distinct indices $a, b \in \mathcal{S}$ and statement $q \in \mathcal{W}$.*

A complementary problem that is equally important is identifying the pairs of indices that exhibit a strong interaction with respect to the statements in q. Thus, the goal is not to obtain the precise degree of interaction but only to decide whether it is high enough. More concretely, we assume we are given a non-negative threshold τ that specifies a lower bound on what constitutes a strong degree of interaction. We define the binary relation $\sim_\tau$ as follows: $a \sim_\tau b$ if and only if $a \neq b$ and $doi_q(a,b) > \tau$ for some query $q \in \mathcal{W}$. The problem can now be defined as follows:

Index Interaction Problem (IIP) *Given an index-set $\mathcal{S}$, a workload $\mathcal{W}$, and a threshold $\tau \geq 0$, compute the binary relation $\sim_\tau$.*

An interesting technical point is that $a \sim_\tau b$ requires at least one $q \in \mathcal{W}$ and $X \subseteq \mathcal{S}$ such that $doi_q(a,b,X) > \tau$. On the other hand, $a \not\sim_\tau b$ essentially requires that $doi_q(a,b,X) \leq \tau$ for all choices of q, X. Thus, it is not possible to determine index independence based on a subset of the choices for q and X, which implies that the worst case complexity for IIP is as bad as DOIP.

3. OPTIMAL PLAN CHARACTERIZATION

In our statement of the DOIP and IIP problems, index interactions are computed in terms of the function $cost_q$ that yields the cost of optimal plans for each query $q \in \mathcal{W}$. If the $cost_q$ function is completely arbitrary, a correct algorithm for either DOIP or IIP has to examine every pair of indices a, b and every index-set $X \subseteq \mathcal{S}$ in order to compute $doi_q(a,b)$. This implies a complexity of $\Omega(2^{|\mathcal{S}|}|\mathcal{S}|^2)$ which can be prohibitively high for real-world workloads.

In practice, query optimization does not depend on the configuration in an arbitrary way. For instance, the cost of a query should not increase when indices are added to the configuration, since this can only expand the space of feasible plans. In order to avoid such anomalies, we make a natural assumption for the behavior of the optimizer. Conceptually, we assume that the optimizer chooses the cheapest execution plan from its search space, while breaking ties in a consistent way. This is formalized by the following definition.

DEFINITION 3.1. ***Well-Behaved Optimizer:*** *An optimizer is* well behaved *if for any query $q \in \mathcal{W}$ there exists a set of plans $\mathcal{P}_q$ and a total order on $\mathcal{P}_q$ such that (i) plans are ordered by non-decreasing cost and (ii) given an index-set X, the optimizer chooses the first plan in the ordering that employs indices solely from X.*

From now on we assume that the optimizer is well behaved. Even with this assumption, the optimizer has a great deal of freedom in its choices since no constraints are placed on the space of plans $\mathcal{P}_q$ or the calculation of plan cost. The main consequence of this assumption is that the optimizer must choose the first available plan in the total order, which is always the cheapest plan since plans are ordered by ascending cost. The order of plans with equal cost is not specified, but all plans chosen for an individual query must be consistent with some total ordering.

We note that our assumption applies to query statements only. The cost of an update statement also includes the cost to update indices and this is not captured properly by the total ordering of plans. However, as clarified in Section 2, our index interaction model for update statements is based on the corresponding query shell, where we may assume the optimizer is well behaved.

The remainder of this section builds further infrastructure on top of the well-behaved optimizer. First we prove useful properties that follow from the definition and then introduce a technique to expose the structure in the cost function, $cost_q$.

3.1 Properties of Optimal Plan Generation

We derive two properties of a well-behaved optimizer that we use extensively in the development of our algorithms. We state the two properties and their proofs below, and then discuss their interpretation.

PROPERTY 3.1. ***Monotonicity****: For any index-sets X, Y and query q, if $X \subseteq Y$ then $cost_q(X) \geq cost_q(Y)$.*

PROOF. Let $\mathcal{P}_q$ be the ordered set of plans considered by the well-behaved optimizer, and let $seq_q(Y)$ be the ordered sequence of plans in $\mathcal{P}_q$ using indices solely in Y. If $X \subseteq Y$, then $optplan_q(X)$ must appear in $seq_q(Y)$. Since $optplan_q(Y)$ is the first member of $seq_q(Y)$, this implies that $optplan_q(X)$ does not precede $optplan_q(Y)$ in the total order. Hence, $cost_q(Y) \leq cost_q(X)$. □

PROPERTY 3.2. ***Sanity****: For any index-sets X, Y and query q, if $used_q(Y) \subseteq X \subseteq Y$ then $optplan_q(X) = optplan_q(Y)$.*

PROOF. Define $seq_q(X)$ and $seq_q(Y)$ as in the previous proof. As before, we know that $optplan_q(X)$ appears in $seq_q(Y)$ when $X \subseteq Y$. We also know $optplan_q(Y)$ is in $seq_q(X)$ from the assumption that $used_q(Y) \subseteq X$. Using the reasoning in the previous proof, these observations imply that $optplan_q(X)$ does not precede $optplan_q(Y)$ and $optplan_q(Y)$ does not precede $optplan_q(X)$. Hence, $optplan_q(X) = optplan_q(Y)$. □

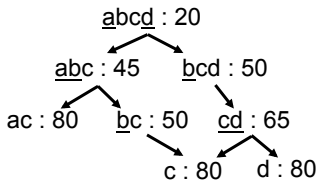

Figure 1: An example IBG for $\mathcal{S} = \{a, b, c, d\}$. The set $used_q(Y)$ for a node Y is depicted by underlining the corresponding indices. The $cost_q(Y)$ is shown beside each node.

The monotonicity property simply states that the cost of a query can only improve if more indices are available. The sanity property states that the optimal plan under a set Y does not change if we exclude from Y indices that are not used in the plan. In what follows, we use $X \lhd Y$ as a shorthand for the notation $used_q(Y) \subseteq X \subseteq Y$ and we say that X is covered by Y, or Y covers X.

We stress that we do not use the previous two properties to bypass the optimizer. Indeed, our algorithms will rely on what-if optimization in order to obtain the optimal plan under hypothetical index sets. These properties, however, imply that the $cost_q()$ function exhibits some structure, which we can exploit to improve efficiency.

As a minor observation, we note that the monotonicity property enables us to simplify the denominator in the definition of $doi_q(a, b, X)$ and obtain the following expression:

$$doi_q(a, b, X) = \frac{|benefit_q(\{a\}, X) - benefit_q(\{a\}, X \cup \{b\})|}{cost_q(X \cup \{a, b\})}.$$

We henceforth assume this definition for $doi_q(a, b, X)$.

3.2 Index Benefit Graph

The concept of an *Index Benefit Graph* (IBG) was introduced by Frank et al. [11]. An IBG enables a space-efficient encoding of the properties of optimal query plans when the optimizer is well behaved. Our algorithms build upon this functionality, however we note that our treatment of the IBG has important differences compared to the original study, as explained in Section 7.

The IBG for a specific query q is a DAG over subsets of $\mathcal{S}$. Each node represents an index-set $Y \subseteq \mathcal{S}$ and records $used_q(Y)$ and $cost_q(Y)$. In a slight abuse of notation, we also use Y to denote the node in the IBG. The nodes and edges of the IBG are defined inductively as follows: The IBG contains the node $\mathcal{S}$; For each node Y and each used index $a \in used_q(Y)$, the IBG contains the node $Y' = Y - \{a\}$ and the directed edge (Y, Y').

Figure 1 shows an example IBG for $\mathcal{S} = \{a, b, c, d\}$. One interesting observation is that bcd and bc differ by index d, yet no edge exists between them because $d \notin used_q(bcd)$. Also, notice that $bcd \rhd bc$ and hence the two nodes are somewhat redundant with respect to information on optimal plans (but they are needed to complete the graph).

Because the IBG nodes only have one child per used index, the size of an IBG for a particular index-set can vary drastically. Some interesting ways to measure the size of an IBG are the number of nodes, the maximum children per node (i.e. *fan-out*), and the maximum path length (i.e. *height*). In the worst case, $used(Y) = Y$ for each node Y and this results in a node for each subset of $\mathcal{S}$, a fan-out of $|\mathcal{S}|$, and a height of $|\mathcal{S}|$. However, in practice the optimizer may not use every index in Y (especially if Y is large), in which case the IBG can be much smaller. Indeed, Figure 1 contains only 8 of the 16 possible subsets, a fan-out of 2, and a height of 3.

An IBG is naturally constructed by a top-down process, starting from $\mathcal{S}$ as the topmost node. For each node Y in the IBG, the process performs a what-if optimization and, for each $a \in used_q(Y)$, adds $Y - \{a\}$ to the children of Y. Each child is built recursively unless it already exists, which may be checked by storing nodes in a hash table. Overall, constructing an IBG with N nodes and fan-out f requires N what-if optimizations, $O(fN)$ operations on the hash table of index-sets, and $O(fN)$ other basic operations.

The key property of the IBG is that it is sufficient to derive $cost_q(X)$ and $used_q(X)$ for *any* index-set $X \subseteq \mathcal{S}$, even if X is not represented directly in the IBG. This is made possible by the following theorem, which is an immediate consequence of Theorem 3 in [11] when the optimizer is well behaved.

THEOREM 3.1. *For any index-set $X \subseteq \mathcal{S}$, there exists a node Y in the IBG such that $X \lhd Y$, i.e., $used(Y) \subseteq X \subseteq Y$.*

Thus, any X is covered by an IBG node Y, and the sanity property implies that $cost_q(X) = cost_q(Y)$ and $used_q(X) = used_q(Y)$. Given that the IBG might not contain a node for each subset of $\mathcal{S}$, we may view the IBG as a space-efficient representation of the optimizer's behavior for a single query.

A covering node for X can be identified efficiently using the following simple algorithm: Start from the root node and iteratively move to a child that corresponds to the removal of an index not in X, until no such child exists. There may be many choices for the followed path, but the final node is always a covering node for X. This algorithm requires enumerating the outgoing edges of each node in the path to find an appropriate child. If the IBG has height h and fan-out f, then the time to find a covering node is $O(fh)$.

4. COMPUTING INDEX INTERACTIONS

In this section, we present novel algorithms that efficiently solve DOIP and IIP (Section 2). The presented algorithms rely heavily on the machinery introduced in the previous section: information about index interactions is derived by analyzing the IBG for each query $q \in \mathcal{W}$, taking advantage of the sanity and monotonicity properties to ensure a correct analysis.

The following subsections present the details of our algorithms. We first describe an algorithm that operates on a single query q and determines the degree of interaction for all pairs of indices by analyzing the IBG of q. This algorithm forms the basis for the algorithms that solve DOIP and IIP for a complete workload, which are presented in the subsequent subsections.

4.1 Computing the Degree of Index Interaction on a Single Query

We present an algorithm that computes $doi_q(a, b)$ for every pair of indices a, b in $\mathcal{S}$. The main idea is to construct the IBG of q first and then derive the degrees of interaction by a careful analysis of the graph's structure. In what follows, we first present the basic intuition and then give a complete algorithm description.

4.1.1 Algorithm Intuition

We start by presenting a naive algorithm that correctly computes $doi_q(a, b)$ for all $a, b \in \mathcal{S}$ but has a prohibitively high complexity for practical applications. We then use it as the vehicle for developing and explaining our proposed techniques.

Figure 2 shows the pseudocode for the NAIVE algorithm. The algorithm first constructs the IBG, which provides a concise representation of the optimal plans for any set $X \subseteq \mathcal{S}$. Subsequently, the algorithm iterates over every possible set X and pair of indices $a, b \in \mathcal{S} - X$ and computes $doi_q(a, b, X)$. The iteration skips in-

Function NAIVE
Input: Index-set $\mathcal{S}$. Query q.
Output: $doi_q(a,b)$ for each distinct $a,b \in \mathcal{S}$.
Data: Hash table $t_q : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$.

1 Initialize $t_q[a,b] \leftarrow 0$ for each distinct $a,b \subseteq \mathcal{S}$;
2 Construct the IBG for q;
3 **foreach** $X : X \subseteq \mathcal{S}$ **do**
4 **foreach** *distinct* $a,b \in \mathcal{S} - X$ **do**
5 $X_a \leftarrow X \cup \{a\}$; $X_b \leftarrow X \cup \{b\}$; $X_{ab} \leftarrow X \cup \{a,b\}$;
6 Find graph nodes Y, Y_a, Y_b, Y_{ab} such that
7 $Y \rhd X, Y_a \rhd X_a, Y_b \rhd X_b, Y_{ab} \rhd X_{ab}$;
8 $d \leftarrow \frac{|cost_q(Y) - cost_q(Y_a) - cost_q(Y_b) + cost_q(Y_{ab})|}{cost_q(Y_{ab})}$;
9 $t_q[a,b] \leftarrow \max\{t_q[a,b], d\}$;
10 **return** $doi_q(a,b) = t_q[a,b]$ for each $\{a,b\} \subseteq \mathcal{S}$;

Figure 2: Naive algorithm for computing $doi_q(a,b)$ for all indices $a,b \in \mathcal{S}$.

Function QINTERACT
Input: Index-set $\mathcal{S}$. Query q.
Output: $doi_q(a,b)$ for each distinct $a,b \in \mathcal{S}$.
Data: Hash table $t_q : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$.

1 Initialize $t_q[a,b] \leftarrow 0$ for each distinct $a,b \in \mathcal{S}$;
2 Construct the IBG for q;
3 **foreach** Y *in the IBG* **do**
4 **foreach** *distinct* $a,b \in \mathcal{S} - used_q(Y)$ **do**
5 $\mathcal{Y} \leftarrow$ generate quadruples (Y, Y_a, Y_b, Y_{ab}) based on Y, a, b ;
6 **foreach** $(Y, Y_a, Y_b, Y_{ab}) \in \mathcal{Y}$ **do**
7 $d \leftarrow \frac{|cost_q(Y) - cost_q(Y_a) - cost_q(Y_b) + cost_q(Y_{ab})|}{cost_q(Y_{ab})}$;
8 $t_q[a,b] \leftarrow \max\{t_q[a,b], d\}$;
9 **return** $doi_q(a,b) = t_q[a,b]$ for each $\{a,b\} : a,b \in \mathcal{S}$;

Figure 3: Algorithm QINTERACT for computing $doi_q(a,b)$ for all $a,b \in \mathcal{S}$. The specifics of the quadruple generation on line 5 are discussed in Section 4.1.2.

dices in X since these indices cannot interact within X. The maximum over all sets X is stored in the hash table entry $t_q[a,b]$, so that at the end of the computation $t_q[a,b]$ yields the desired metric $doi_q(a,b)$.

Each time the algorithm computes $doi_q(a,b,X)$, it requires the cost of q under the quadruple of configurations (X, X_a, X_b, X_{ab}) where $X_a \equiv X \cup \{a\}$, $X_b \equiv X \cup \{b\}$ and $X_{ab} \equiv X \cup \{a,b\}$. The algorithm evaluates these costs by covering each set with a node from the IBG. These covering nodes are named Y, Y_a, Y_b, Y_{ab} but note that these choices are not unique. The sanity property ensures that the costs of corresponding nodes are equal, implying that $doi_q(a,b,X)$ can be computed using the quadruple (Y, Y_a, Y_b, Y_{ab}) in place of (X, X_a, X_b, X_{ab}).

Recall that the number of nodes in the IBG may be much smaller than the number of subsets of $\mathcal{S}$. This suggests that there may be other subsets $X' \neq X$ such that Y, Y_a, Y_b, and Y_{ab} are the covering nodes of X', $X' \cup \{a\}$, $X' \cup \{b\}$, and $X' \cup \{a,b\}$. In other words, the same Y-quadruple may be used to compute $doi_q(a,b,X)$ for multiple choices of X.

EXAMPLE 4.1. *Consider again the IBG shown in Figure 1 and suppose we are interested in the value of $doi_q(a,b,\{c\})$. This degree of interaction is determined by the query costs for the quadruple $\{c\}$, $\{c,a\}$, $\{c,b\}$, and $\{c,a,b\}$. These costs are straightforward to derive from the IBG, since these sets correspond exactly to graph nodes. Carrying out the arithmetic yields $doi_q(a,b,\{c\}) = |(80-80)-(50-45)|/45 \approx 0.1$. On the other hand, consider $doi_q(a,b,\emptyset)$. This relies on the query costs for the quadruple $\emptyset$, $\{a\}$, $\{b\}$, and $\{a,b\}$. None of these sets exist in the IBG, but these nodes are covered one-to-one by the previous quadruple, i.e., $\{c\} \rhd \emptyset$, $\{c,a\} \rhd \{a\}$, $\{c,b\} \rhd \{b\}$, and $\{c,a,b\} \rhd \{a,b\}$. Therefore, the same quadruple used to compute $doi_q(a,b,\{c\})$ may be used to compute $doi_q(a,b,\emptyset)$.* ∎

This example indicates that the NAIVE algorithm may be doing redundant work by considering all subsets $X \subseteq \mathcal{S}$, since different X-quadruples may be covered by the same Y-quadruple of graph nodes. Hence, our goal is to calculate the degree of interaction for a sufficient collection of Y-quadruples in order to account for all possible choices of X. This approach can lead to substantial savings if the IBG is small in size.

4.1.2 *The* QINTERACT *Algorithm*

Our new algorithm, termed QINTERACT, is based on the idea presented above. The pseudocode is given in Figure 3. As shown, the algorithm follows a very similar structure to the NAIVE algorithm of Figure 2, since the IBG is constructed first (line 2) and then there is an analysis phase which follows to compute interactions (lines 3–8). The key difference from NAIVE is the enumeration of IBG nodes instead of all subsets of $\mathcal{S}$ in the analysis phase.

For each node Y in the IBG, the QINTERACT algorithm performs a number of tests for interaction between indices. At a high level, the goal is to capture the interactions that are witnessed by some of the sets X covered by Y, but not necessarily all such X. When processing Y, the algorithm tests interaction for all pairs of indices $a,b \in \mathcal{S} - used_q(Y)$. The used indices of Y are excluded because any index used in Y must also exist in the node X covered by Y, and we know that $doi_q(a,b,X) = 0$ whenever a or b is in X.

The first step in processing a pair of indices is the formation of quadruples (Y, Y_a, Y_b, Y_{ab}) (line 5), each of which are used to compute a degree of interaction and update the current maximum in $t_q[a,b]$, much in the same way as NAIVE. As mentioned before, the Y-quadruples should be generated to account for each X with at least one choice of Y. Our investigation shows that this goal is non-trivial and requires very careful choices for the components Y_a, Y_b, and Y_{ab}.

A natural, but incorrect, approach is to generate nodes that simply test whether an interaction is witnessed by each graph node Y. To strengthen this approach, a and b should be excluded from Y since $doi_q(a,b,Y) = 0$ when a or b is in Y. Hence, we can try generating quadruples that yield the degree of interaction witnessed by $Y - \{a,b\}$. This implies the following strategy for line 5:

Given Y, a, b where $a,b \notin used_q(Y)$, choose IBG nodes
$Y_a \rhd (Y - \{a,b\}) \cup \{a\}$
$Y_b \rhd (Y - \{a,b\}) \cup \{b\}$
$Y_{ab} \rhd (Y - \{a,b\}) \cup \{a,b\}$
Consider the following choice of $\mathcal{Y}$:
$\mathcal{Y}^{\text{wrong}} \leftarrow \{(Y, Y_a, Y_b, Y_{ab})\}$

We can verify that $Y \rhd Y - \{a,b\}$ because $a,b \notin used_q(Y)$. Therefore, the value of d that is computed using the single tuple in $\mathcal{Y}^{\text{wrong}}$ will capture $doi_q(a,b,Y - \{a,b\})$. Following the reasoning used earlier, d will also capture the value of $doi_q(a,b,X)$ for any other X such that $X \lhd Y \wedge X_a \lhd Y_a \wedge X_b \lhd Y_b \wedge X_{ab} \lhd Y_{ab}$. Unfortunately, this may not be sufficient to capture all index interactions, which may result in an underestimate of $doi_q(a,b)$. This fact is proven in the following example.

EXAMPLE 4.2. *Consider the problem of computing $doi_q(a,b)$ on the IBG shown in Figure 4(a). As a first step, we can compute*

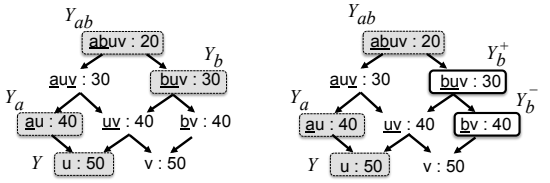

Figure 4: Quadruples of nodes in the computation of $doi_q(a,b)$. Part (a) depicts the case where a single quadruple (Y, Y_a, Y_b, Y_{ab}) is used. Part (b) depicts the case where the algorithm uses Y_b^- and Y_b^+ instead of Y_b.

the value of $doi_q(a,b,\emptyset)$ by observing that $\{u\} \rhd \emptyset$, $\{a,u\} \rhd \{a\}$, $\{b,v\} \rhd \{b\}$, and $\{a,b,u,v\} \rhd \{a,b\}$. This yields $doi_q(a,b,\emptyset) = |(50-40)-(40-20)|/20 = 0.5$. However, if we use $\mathcal{Y}^{wrong}$ to implement line 5 of QINTERACT*, the algorithm will not discover an interaction between a and b when visiting any node. For example, the shaded quadruple (Y, Y_a, Y_b, Y_{ab}) would be examined when visiting node $\{u\}$, but these nodes do not witness an interaction between a and b.* ∎

Conceptually, the flaw with $\mathcal{Y}^{\text{wrong}}$ is that that there may be some $X \subseteq \mathcal{S}$ such that the sets $X, X \cup \{a\}, X \cup \{b\}, X \cup \{a,b\}$ do not have costs that match any quadruple (Y, Y_a, Y_b, Y_{ab}). Below, we describe a correction to this flaw, along with some high level reasoning behind the strategy. We formally prove that the final strategy is correct in the accompanying technical report [16].

We start with an important observation that is drawn from the technical report: When a, b are fixed and we consider an arbitrary $X \subseteq \mathcal{S}$, there is some IBG node Y such that

$$\begin{aligned} & cost_q(Y) = cost_q(X) \\ \wedge\ & cost_q(Y_a) = cost_q(X \cup \{a\}) \\ \wedge\ & cost_q(Y_{ab}) = cost_q(X \cup \{a,b\}). \end{aligned}$$

Hence, the Y-quadruples considered by $\mathcal{Y}^{\text{wrong}}$ will see three out of four costs that are needed to compute $doi_q(a,b,X)$. Based on this observation, a reasonable strategy is to use the three exact costs provided above, and find tight bounds for the fourth required value $cost_q(X \cup \{b\})$. Since the algorithm is only interested in the maximum $doi_q(a,b,X)$ over X, it is sufficient to consider the value of $cost_q(X \cup \{b\})$ within the bounds that yields the largest degree of interaction. To derive bounds on $cost_q(X \cup \{b\})$, we find IBG nodes Y_b^-, Y_b^+ conforming to

$$\begin{aligned} U_{ab} &= used_q(Y) \cup used_q(Y_a) \cup used_q(Y_{ab}) \\ Y_b^- &\rhd (U_{ab} - \{a,b\}) \cup \{b\} \\ Y_b^+ &\rhd (Y - \{a,b\}) \cup \{b\} \end{aligned}$$

and assume $cost_q(X \cup \{b\}) \in [cost_q(Y_b^+), cost_q(Y_b^-)]$. The inspiration behind this approach is the following fact, which follows easily from our definitions and the constraint $a, b \notin used_q(Y)$:

$$\begin{aligned} & (X \lhd Y) \ \wedge\ (X \cup \{a\} \lhd Y_a) \ \wedge\ (X \cup \{a,b\} \lhd Y_{ab}) \\ & \text{if and only if } U_{ab} - \{a,b\} \subseteq X \subseteq Y - \{a,b\} \end{aligned}$$

The first line is related to the earlier observation that three of the four required cost values in the computation of $doi(X,a,b)$ are captured by some Y, Y_a, Y_{ab}. When these equivalent conditions hold, the monotonicity and sanity properties imply that the costs of Y_b^+, Y_b^- provide tight bounds for the fourth value $cost_q(X \cup \{b\})$. To make the final leap in reasoning, we notice that the expression for $doi_q(a,b,X)$ is a convex function of $cost_q(X \cup \{b\})$, which implies the maximum value is taken when $cost_q(X \cup \{b\})$ has an extreme value. This means that it is sufficient to consider only the cases where $cost_q(X \cup \{b\})$ is equal to $cost_q(Y_b^+)$ or $cost_q(Y_b^-)$. Finally, our implementation of line 5 chooses two quadruples.

$$\mathcal{Y} \leftarrow \{(Y, Y_a, Y_b^-, Y_{ab}), (Y, Y_a, Y_b^+, Y_{ab})\}$$

Using this strategy, we have the following correctness result.

THEOREM 4.1. *The values $doi_q(a,b)$ returned by* QINTERACT *are correct for every distinct $a, b \in \mathcal{S}$.*

We conclude our presentation of QINTERACT with a running time analysis. The running time of IBG construction was detailed in Section 3, so we focus on the analysis phase. As in Section 3, we assume the IBG has N nodes, height h, and fan-out f. For each IBG node Y and relevant indices a, b, the QINTERACT algorithm generates two quadruples and does a constant amount of computation for each. Hence, the processing for Y, a, b is dominated by the time to generate a quadruple. A quadruple is generated by searching for a constant number of of covering nodes, and each search requires $O(fh)$ time (Section 3). Hence, $O(fh)$ time is spent processing Y, a, b, which results in $O(fhN \cdot |\mathcal{S}|^2)$ time to compute the $\Theta(|\mathcal{S}|^2)$ return values. In cases where the size of the IBG is polynomial in $|\mathcal{S}|$, this implies polynomial time complexity overall.

4.2 Algorithms for DOIP

We now turn our attention to the Degree of Interaction Problem (DOIP) where the goal is to compute $doi_q(a,b)$ for every $q \in \mathcal{W}$ and $a, b \in \mathcal{S}$. We present two algorithms, termed SERIAL and INTERLEAVED, that use the logic of QINTERACT as a sub-routine in order to solve DOIP.

The two algorithms that we present share the following important feature: They maintain a per-query hash table $l_q : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ such that, at any point in time, $l_q[a,b]$ provides a lower bound on $doi_q(a,b)$. Each bound $l_q[a,b]$ is initialized to 0, meaning that the algorithm does not have any information on the interaction between a and b, and it converges to the final value $doi_q(a,b)$ as the algorithm is running. The bounds thus provide an any-time approximate solution to DOIP, and can even serve in lieu of the actual solution if the database administrator finds the approximation sufficient. The following paragraphs describe the details of the two algorithms.

The SERIAL algorithm is a direct adaptation of QINTERACT to DOIP. Essentially, the algorithm iterates over every query q in $\mathcal{W}$ and performs lines 1–8 of QINTERACT, except that the hash table t_q is replaced with l_q. Once the processing of q is complete, each entry $l_q[a,b]$ is equivalent to $doi_q(a,b)$.

The INTERLEAVED algorithm interleaves IBG construction and analysis among all queries in the workload in an attempt to compute more interactions sooner. It thus avoids the "blocking" nature of SERIAL which computes interactions for one query at a time.

More concretely, INTERLEAVED performs a round-robin iteration over the queries in $\mathcal{W}$, and for each q it expands its IBG by invoking the what-if optimizer on exactly one node. To illustrate this process, assume that a query q has the IBG shown in Figure 1. Initially, the graph will contain $\mathcal{S} = \{a,b,c,d\}$ but $used_q(\mathcal{S})$ and $cost_q(\mathcal{S})$ will not be known. The first time q is examined, a what-if call will be made to compute $used_q(\mathcal{S})$ and $cost_q(\mathcal{S})$ and to add the nodes $Y_1 = \{a,b,c\}$ and $Y_2 = \{b,c,d\}$ which will be marked as unexpanded. After examining other queries, INTERLEAVED will revisit q and invoke the what-if optimizer on one of the unexpanded nodes, say Y_j, to discover $cost_q(Y_j)$ and $used_q(Y_j)$, which in turn will cause more unexpanded nodes to be introduced, and so on. We assume that unexpanded nodes are processed in order of their size, which guarantees that the IBG is built in a breadth-first fashion.

After a node expansion, INTERLEAVED processes the new node by testing all relevant pairs of indices for interaction, using the approach from lines 3–8 of qINTERACT, except that the hash table t_q is replaced with l_q. In this processing, there is a possibility that one or more covering nodes that are required for the analysis may not be expanded. INTERLEAVED keeps track of these occurrences and retries the computation after more of the graph is built. In our experience observing the behavior of INTERLEAVED, the breadth-first expansion of the IBG is conducive to minimizing the likelihood of needing to revisit a node.

Overall, we expect INTERLEAVED to update its lower bounds more steadily compared to SERIAL and it is thus a more appealing choice for generating an approximate solution to DOIP. We note that we also explored methods for deriving upper bounds on $doi_q(a, b)$ based on partially constructed IBGs. However, the resulting bounds were very loose and thus did not provide much information. Our investigation indicated that obtaining a tight upper bound is technically challenging, but more generally its usefulness is unclear due to the difficulty of determining index independence.

4.3 Algorithms for IIP

Recall that the Index Interaction Problem (IIP) entails the computation of the binary relation $\sim_\tau$, where $a \sim_\tau b$ iff $doi_q(a, b)$ exceeds a threshold $\tau \geq 0$. Our solution to this problem is essentially an adaptation of the aforementioned SERIAL and INTERLEAVED algorithms. The idea is to output the pair (a, b) in the solution as soon as $l_q[a, b] > \tau$ for some query q, since the lower bounding property also implies that $doi_q(a, b) > \tau$. Thus, at any point in time, SERIAL and INTERLEAVED provide a subset of $\sim_\tau$ which can serve as an approximate solution to IIP, and they converge to the true solution upon completion.

Both algorithms can be optimized by modifying qINTERACT to skip pairs of indices a, b that already appear in the solution. However, the worst case complexity for either algorithm is exactly the same as its counterpart for DOIP, since pairs of independent indices $a \not\sim_\tau b$ force the algorithms to compute the precise value of $doi_q(a, b)$ for every query $q \in \mathcal{W}$.

5. EXPERIMENTAL STUDY

In this section, we conduct an empirical study to evaluate the efficiency of algorithms SERIAL and INTERLEAVED in different scenarios. We first describe the aspects of our experimental methodology, then provide a summary of our main results.

5.1 Methodology

Testing platform. Our experiments use a prototype implementation of our interaction modeling methodology written in the Java language. The implementation includes algorithms SERIAL and INTERLEAVED for DOIP and IIP. The code is executed on a machine running Mac OS X with a 2.4GHz dual-core processor and 2GB RAM. The database system in our experiments is the freely available IBM DB2 Express-C which includes an advisor tool that can generate index recommendations for a representative workload. We use the EVALUATE INDEXES mode of DB2 to perform what-if optimization.

Data and queries. We use the TPC-H data set with the default scale of 1 GB to evaluate our techniques for discovering index interactions. We chose TPC-H since it is commonly used in studies on index selection [2, 15, 12]. Our representative workload $\mathcal{W}$ comprises the 22 benchmark queries, which provide an interesting variety of query plans where indices may interact.

Candidate index generation. Both SERIAL and INTERLEAVED assume that a set of candidate indices $\mathcal{S}$ is provided as input. Of course, there are many methods for generating $\mathcal{S}$ based on the database and representative workload. In our setting, it is reasonable to use the default advisor provided by DB2 to select the candidate indices. Since larger index-sets have more potential for index interactions, we do not specify any constraints on the size of the recommendation. This essentially results in an index configuration that is optimal for the workload. We refer to this set of indices as $\mathcal{S}^{\mathsf{ALL}}$. As an alternative approach to candidate generation, we also consider the set of indices $\mathcal{S}^{\mathsf{1C}}$ that contains a single-column index on $R.A$ for each attribute $R.A$ that occurs in some member of $\mathcal{S}^{\mathsf{ALL}}$. In other words, $\mathcal{S}^{\mathsf{1C}}$ contains all single-column indices that might be relevant for the workload. We note that $|\mathcal{S}^{\mathsf{ALL}}| = 51$ and $|\mathcal{S}^{\mathsf{1C}}| = 48$.

Evaluation metrics. As described in Section 4, SERIAL and INTERLEAVED begin with no knowledge of interactions and progressively obtain information throughout their execution. It is thus interesting to examine the rate at which each algorithm obtains information on index interaction before it completes.

We measure running time with two metrics. One of our metrics, which is the most natural, is the real "wall clock" time consumed by the algorithm. This metric provides an idea of the practicality of our approach, but the real time is highly dependent on the database system and computer architecture. A more stable metric for running time is the number of query optimizations performed for IBG construction. In our experiments, we found that this was the most expensive operation, and the true running time of the algorithm was highly correlated with the number of query optimizations.

The metric for measuring algorithm progress depends on whether the algorithm is applied to IIP or to DOIP. For IIP, the metric is the number of pairwise interactions given a specific threshold τ. We vary τ over the set $\{0.01, 0.1, 1\}$ to observe the trends for different thresholds. For DOIP, we measure progress as the distance between the lower bound $l_q[a, b]$ and the actual interaction $doi_q(a, b)$ over all queries in $\mathcal{W}$. The metric we use for this is *mean relative error* for all interacting pairs of indices in $\mathcal{S}$, which is expressed as

$$MRE = \frac{1}{\sum_{q \in \mathcal{W}} |P_q|} \cdot \sum_{q \in \mathcal{W}} \sum_{(a,b) \in P_q} 1 - \frac{l_q[a, b]}{doi_q(a, b)}$$

$$\text{where } P_q = \{(a, b) \in \mathcal{S} \times \mathcal{S} \mid a \neq b \wedge doi_q(a, b) > 0\}$$

Note that each term in the summation is nonnegative since each $l_q[a, b]$ is initially zero and increases monotonically to $doi_q(a, b)$.

5.2 Results

We now present our performance results for the two algorithms, SERIAL and INTERLEAVED (abbreviated "intlvd" in the figures). We first show the performance of these algorithms when solving IIP, and then discuss their performance for DOIP. Some of the different scenarios in our study showed very similar trends, so we restrict our presentation to a representative subset of the results.

Convergence to IIP Solution. We measure the convergence of SERIAL and INTERLEAVED for the IIP problem in terms of the number of known interacting pairs over time. The first results that we present use optimizer invocations to measure time, and the following results use wall clock time.

Figure 5 shows our measurements using the $\mathcal{S}^{\mathsf{ALL}}$ candidate set and threshold of $\tau = 0.01$. This low setting of τ causes the number of interacting pairs $a \sim_\tau b$ to be 180. The curves that we observed for $\tau = 0.1$ and 1.0 had very similar shape, although the total number of interactions were reduced to 122 and 92 respec-

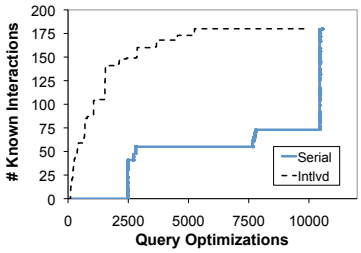

Figure 5: Convergence with $\mathcal{S}^{\text{ALL}}$ candidate set and $\tau = 0.01$. Results with $\tau = 0.1, 1.0$ showed similar trends.

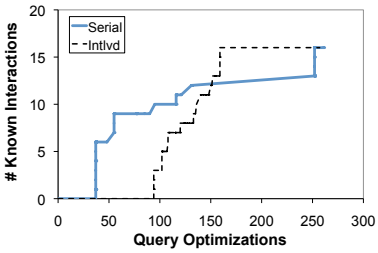

Figure 6: Convergence with $\mathcal{S}^{\text{1C}}$ candidate set and $\tau = 0.01$. Results with $\tau = 0.1$ showed similar trends.

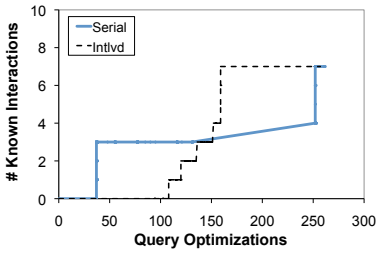

Figure 7: Convergence with $\mathcal{S}^{\text{1C}}$ candidate set and $\tau = 1.0$.

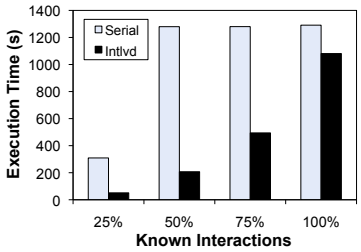

Figure 8: Time required to discover percentages of interactions in $\mathcal{S}^{\text{ALL}}$ candidate set, where $\tau = 0.1$.

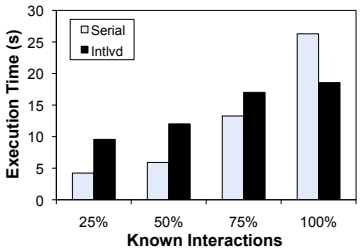

Figure 9: Time required to discover percentages of interactions in $\mathcal{S}^{\text{1C}}$ candidate set, where $\tau = 0.1$.

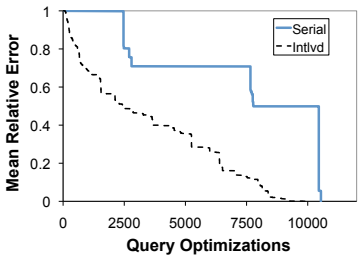

Figure 10: Convergence of MRE with $\mathcal{S}^{\text{ALL}}$ candidate set.

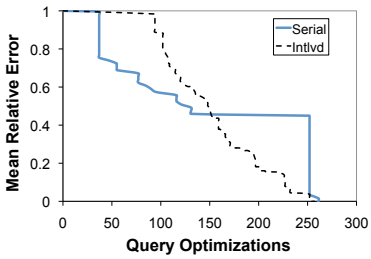

Figure 11: Convergence of MRE with $\mathcal{S}^{\text{1C}}$ candidate set.

tively. The graph clearly shows that INTERLEAVED starts discovering interactions very quickly compared to SERIAL. Moreover, INTERLEAVED finds all 180 interacting pairs after about 5,000 optimizer calls, which is about half of the number of calls required to build all of the index benefit graphs.

The trends in our observations for the $\mathcal{S}^{\text{1C}}$ candidate set are noticeably different. The difference may be due to the smaller IBGs, which required less than 300 query optimizations to construct, compared to about 10,000 for $\mathcal{S}^{\text{ALL}}$. The results for $\tau = 0.01$ are shown in Figure 6 (results for $\tau = 0.1$ were very similar). The graph indicates that serial analysis finds several of the interacting pairs by examining the first few queries, whereas interleaved analysis takes more time to find the first interactions. On the other hand, the interleaved approach still converges to complete knowledge of interactions significantly faster than serial. We see somewhat similar behavior when the value of τ is increased to 1.0, as shown in Figure 7. However, the convergence with the serial approach is much less smooth, since most of the interactions are discovered near the beginning or the end of the analysis.

We next turn our attention to the wall clock time used by the algorithms, for a moderate setting of the threshold $\tau = 0.1$. Figure 8 shows the time taken to discover a given percentage of interacting pairs within the $\mathcal{S}^{\text{ALL}}$ index-set. The interleaved approach finds the first 25% of interacting pairs in under one minute, whereas serial analysis takes several minutes to make this progress. The number of interacting pairs known in the serial approach stays below 50% of the total until the end, when the IBGs are finished constructing. The two approaches are more comparable within the $\mathcal{S}^{\text{1C}}$ index set, as shown in Figure 9. The overall time required to converge is under 30 seconds, which is much smaller than the 15 minutes needed to analyze the $\mathcal{S}^{\text{ALL}}$ index-set. This is interesting since $\mathcal{S}^{\text{ALL}}$ and $\mathcal{S}^{\text{1C}}$ have comparable size—essentially, our observations imply that the structure of the IBG has more of an effect on running time than the number of candidate indices. In this case, the $\mathcal{S}^{\text{ALL}}$ index set has many relevant indices for some queries, which leads to large IBGs.

Convergence to DOIP Solution. We now examine the progress of SERIAL and INTERLEAVED for DOIP in terms of the mean relative error (MRE) metric defined previously. In Figure 10, we observe that INTERLEAVED converges to zero error much more smoothly than SERIAL. This is a desirable property for an algorithm, since

it means that it is generally beneficial to spend more time with the analysis. The jumps in error exhibited with serial analysis are less attractive, since they show that long periods of time may pass where the algorithm does not make progress. The picture is somewhat different for the $\mathcal{S}^{1C}$ index-set shown in Figure 11. The interleaved approach takes some time to make any significant progress, but converges smoothly after the first 100 query optimizations. We can interpret this behavior based on the implementation described in Section 4.2. Since the interleaved approach alternates between queries when building IBGs, this suggests that the upper layers of the IBGs of each query do not have significant information about index interactions. It appears that the first few queries analyzed in the serial approach lead to more initial improvement in MRE.

Discussion. The results of our study show the practicality of analyzing index interactions assuming that the optimizer is well behaved. Recall that without this assumption, our only recourse is to evaluate the workload cost under all $2^{|\mathcal{S}|}$ index-sets. By extrapolating the measurements on our testing platform, this would require several million years of computation to analyze the $\mathcal{S}^{ALL}$ configuration. Our results also show the benefits of the INTERLEAVED algorithm, which generally converges faster than the SERIAL variant. The advantage is more clear for the $\mathcal{S}^{ALL}$ index-set, as illustrated by the dominance shown in Figure 5, 8, 10. We argue that this case deserves the most attention, since the total number of interactions imply a more challenging analysis.

Finally, we note that there may be significant room for improvement with respect to the absolute running time of the algorithms. As mentioned earlier, the main bottleneck in the analysis is what-if optimization. Our prototype uses the optimizer in a naive fashion, which starts from scratch each time the set of available indices changes. A more intelligent implementation could take advantage of recent work on efficient query optimization under varying physical designs [5, 14] which can improve the performance of query optimization by up to two orders of magnitude [5].

6. APPLICATION IN TUNING TOOLS

The algorithms described in Section 4 provide a solution to DOIP, which includes complete information about all index interactions that exist among indices in $\mathcal{S}$. However, a list of interacting pairs is not straightforward for a database administrator to interpret, so this information is not easy to utilize in its raw form. In this section, we investigate the development of database tuning tools that are based on the index interactions discovered by the algorithms for DOIP. These tools are meant to augment the current tool set provided by commercial DBMSs.

6.1 Visualizing Index Interactions

As mentioned before, knowledge of index interactions allows for a better understanding of the cost/benefit trade-offs involved when materializing indices. Consider a scenario where an index a is being considered as an addition to the current materialized set. If a has a negative interaction with a materialized index b, a standard what-if analysis may return low benefit for materializing a. On the other hand, knowledge of the interaction would suggest replacing b with a as another interesting design choice.

In these scenarios, it would be useful to develop visualization tools that present information about index interactions in a concise yet informative fashion. Since a solution to DOIP gives the degree of interaction for every pair of indices, it is natural to interpret these values as the weights of edges in an undirected graph. The threshold τ may be used to filter edges of low weight, meaning that the graph only includes an edge between a and b if $a \sim_\tau b$.

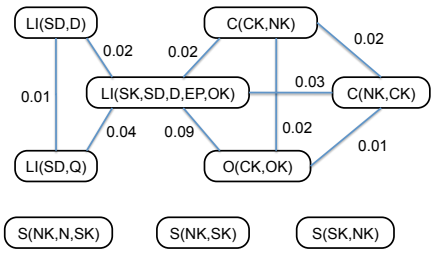

Figure 12: Graph of index interactions for TPC-H Query 7.

EXAMPLE 6.1. *A sample interaction graph is presented in Figure 12, which is derived from Q7 in the TPC-H benchmark and the $\mathcal{S}^{ALL}$ set of indices described in Section 5. The nodes of the graph comprise the members of $\mathcal{S}^{ALL}$ that are relevant to the query, and the edges are denoted with the value of doi_q for the pair of indices (edges below the threshold $\tau = 0.01$ are omitted). One immediate observation that can be made from the graph is that, although one would expect the overlapping indices on relation* S *to interact, the lack of edges between them indicates that their interaction is not strong compared to the query cost. In fact, the strongest interaction occurs between indices on different tables* LI *and* O. ∎

The graph described in the previous example was intentionally simplistic for the purpose of illustration. The basic graph could be extended in many ways through a GUI. For instance, the edges could be separated into positive and negative interactions. It would also be straightforward to allow an administrator to select an edge and examine the particular conditions in which the indices interact.

Our example graph was reasonably small, but in some cases the graph may have too many edges to examine one at a time. In these situations, some insight may still be gained by examining the connected components of the graph. In Figure 12, there is one component of seven indices and three singleton components. Intuitively, the connected components are a sound visualization method because they represent a partitioning of $\mathcal{S}$ into subsets of independent benefit. In what follows, we formalize this intuition.

DEFINITION 6.1. *A* stable subset *of $\mathcal{S}$ is defined as any subset $C \subseteq \mathcal{S}$ such that $benefit(C', X) = benefit(C', X \cap C)$ for all $C' \subseteq C$, $X \subseteq \mathcal{S}$.*

The definition of a stable subset C essentially states that the benefits of indices within C are not affected by indices outside of C. This is a useful property for a system administrator to know, as it ensures that decisions may be made independently within different stable subsets.

In principle, smaller stable subsets provide more information about which indices are independent (indeed, $\mathcal{S}$ is a stable subset of itself, but this does not provide any information). Our main result states that if we consider the graph with edges defined by the $\sim_0$ relation, each connected component is stable. Moreover, $\mathcal{S}$ cannot be decomposed further into smaller stable subsets. The following theorem gives the formal result.

THEOREM 6.1. *Define $G(V, E)$ as the undirected graph where $V = \mathcal{S}$ and $(a, b) \in E$ if and only if $a \sim_0 b$. Let $C_1, \ldots, C_m$ denote a partitioning of $\mathcal{S}$ corresponding to the connected components of G. Then the following hold:*

(1) *Each C_i is a stable subset of $\mathcal{S}$.*

(2) *If $C_i' \subseteq C_i$ and $1 \leq |C_i'| < |C_i|$, then C_i' is not stable.*

(3) *$C_1, \ldots, C_m$ is the only partitioning of $\mathcal{S}$ satisfying (1), (2).*

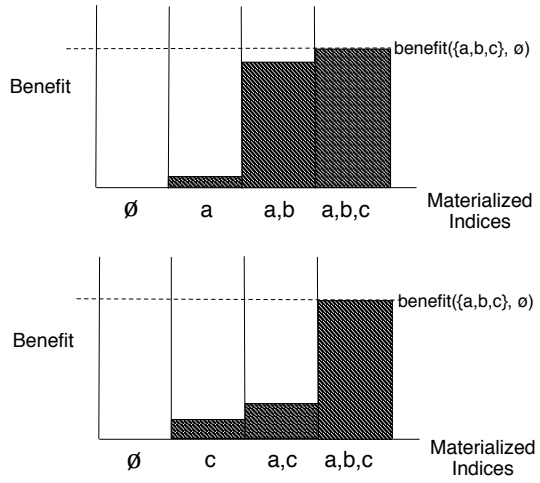

Figure 13: Benefit for the schedules in Example 6.2.

The main step of the proof [16] is to show that $C \subseteq \mathcal{S}$ is stable if and only if C is a union of connected components.

The theorem shows that the connected components have a useful interpretation on their own, even if the edges within the components are not known. It is interesting to note that we examined the convergence to the correct connected components in the experiments of Section 5. In general, we observed that discovering the connected components was significantly less expensive, as the INTERLEAVED algorithm found the correct connected components 2–5 times faster than discovering all interacting pairs of indices. We also observed that most stable subsets were small, containing fewer than ten indices in nearly all cases.

6.2 Scheduling Index Materializations

6.2.1 Motivation and Problem Formalization

Our discussion of interaction visualization showed how a solution to IIP can help an administrator understand the relationships between indices when selecting a configuration. After the configuration is decided, it is still important to choose a good schedule for materializing the new indices. The ideal schedule would materialize all indices at once during a "maintenance window" of low database activity. Unfortunately, this might not be feasible due to the size of the new indices or other constraints, which forces the indices to be gradually materialized over time. Obviously, this has the consequence that queries may only take advantage of the subset of indices that have been materialized in the past.

When an index configuration is materialized over time, it is preferable to schedule indices in a way that achieves high benefit as early as possible. The following example shows that index interactions play a significant role when determining the best schedule under this criterion.

EXAMPLE 6.2. *Let $\mathcal{S} = \{a, b, c\}$ and assume that only a and b interact. We specify the benefit function as follows:*

$$\begin{aligned} benefit(\{a\}, \emptyset) &= 5 \qquad benefit(\{c\}, \emptyset) = 10 \\ benefit(\{b\}, \emptyset) &= 5 \\ benefit(\{a, b\}, \emptyset) &= 100 \end{aligned}$$

Consider two materialization schedules which materialize indices in the order a, b, c and c, a, b. Assume that each index is materialized in a separate maintenance window. Figure 13 shows the benefit that is realized by the two schedules after each window. The first schedule is the natural choice since the majority of the benefit of $\mathcal{S}$ is realized by $\{a, b\}$, and the first schedule achieves this benefit as early as possible. On the other hand, if the positive interaction between a and b were unknown, we might be led to choose the second schedule that materializes c first, since this achieves a slightly higher benefit after the first window. ∎

In this example, it is not immediately obvious which schedule is preferred, since the first schedule has the least benefit after the first window, but the most benefit after the second window. In order to enable a principled study of materialization scheduling, we introduce some terminology and a metric for the quality of a schedule. Any sequence of distinct indices is referred to as a *materialization schedule*. If T is a materialization schedule, we use $T(i)$ to denote the i-th element of T and define $T[i, j] = \{T(k) \mid i \leq k \leq j\}$. If $\mathcal{S}$ is a set of indices and the set of elements in T is equal to $\mathcal{S}$, then we say T is a schedule of $\mathcal{S}$. In this case, we define our quality metric $accbenefit$ of T as

$$accbenefit(T) = \sum_{i=1}^{|\mathcal{S}|} benefit(T[1, i], \emptyset).$$

This metric indicates the accumulated benefit of T over time, which is equivalent to the shaded areas in Figure 13. The first schedule has a higher value of $accbenefit$, which confirms our instinct to materialize the highly beneficial pair a, b as early as possible. We use this metric to formalize the scheduling problem.

Index Materialization Scheduling Problem (IMSP) *Given an index-set $\mathcal{S}$, compute a schedule T of $\mathcal{S}$ maximizing $accbenefit(T)$.*

Before exploring solutions to IMSP, we analyze the computational complexity of the problem. We first make the problem input more precise. The statement of IMSP explicitly requires $\mathcal{S}$ as an input, but we must also consider the workload $\mathcal{W}$ and the well-behaved optimizer to be inputs to the problem, since these are needed to evaluate $benefit(X, Y)$. For the sake of this analysis, it is reasonable to model the optimizer using polynomial-time algorithms that compute $cost_q(X)$ and $used_q(X)$ for any $q \in \mathcal{W}$ and $X \subseteq \mathcal{S}$. To formulate a simple example, suppose all indices are used independently for each query in $\mathcal{W}$. An algorithm for query cost may fix constants $c_q \equiv cost_q(\emptyset)$ and $\beta_q^a \equiv benefit_q(\{a\}, \emptyset)$ for all $q \in \mathcal{W}$, $a \in \mathcal{S}$, and define $cost_q(X) = c_q - \sum_{a \in X} \beta_q^a$. We also have $used_q(X) = X$ in this example.

Consider the decision problem related to IMSP, where we wish to know whether there exists a schedule T such that $accbenefit(T)$ is above a particular value. This decision problem is clearly in NP since $accbenefit$ requires polynomial time to evaluate. The hardness of IMSP is expressed by the following theorem.

THEOREM 6.2. IMSP *cannot be solved in polynomial time unless P = NP.*

PROOF. The proof uses a reduction from the Pipelined Set Cover problem as presented by Munagala et al. [13]. Essentially, we assume that the benefit of a configuration is described in terms of a union of sets, which correspond to the sets in the Pipelined Set Cover problem. Although this representation of benefit cannot describe *all* instances of IMSP, it is general enough to reduce all instances of Pipelined Set Cover, which is sufficient to show the hardness of the problem. The details of the reduction may be found in the technical report [16]. □

Since our proof of hardness shows that IMSP has a close relationship with the Pipelined Set Cover problem, it makes sense to apply the greedy heuristics associated with Set Cover. In our

context, this amounts to choosing the indices from first to last, according to the following inductive rule:

$$T(i) = \operatorname*{argmax}_{a \in \mathcal{S} - T[1,i-1]} \mathit{benefit}(\{a\}, T[1, i-1]).$$

In the context of Pipelined Set Cover, this heuristic has been shown to approximate the cost of the optimal ordering by a factor of 4 [9]. However, the proof of this approximation ratio does not extend to IMSP. As we show next, the greedy heuristic may result in a schedule with an $\mathit{accbenefit}$ approximately $1/(|S| - 1)$ times optimal.

EXAMPLE 6.3. *Let $\mathcal{S} = \{a_1, a_2, \ldots, a_{n-2}, b_1, b_2\}$. Suppose that each a_i does not interact with any other index, and has a benefit of 1. Also assume that b_1 and b_2 have zero benefit individually, but the pair b_1, b_2 provides a benefit of β. The greedy heuristic will produce a schedule T^{G} that materializes b_1 and b_2 last, resulting in an accumulated benefit of*

$$\mathit{accbenefit}(T^{\mathsf{G}}) = \beta + \frac{(n-2)(n+3)}{2}.$$

When $\beta > 2$, it is straightforward to show that the optimal schedule T^* materializes b_1, b_2 first, so

$$\mathit{accbenefit}(T^*) = (n-1)\beta + \frac{(n-2)(n-1)}{2}.$$

As $\beta \to \infty$, the ratio $\mathit{accbenefit}(T^{\mathsf{G}})/\mathit{accbenefit}(T^*)$ approaches $1/(n-1)$. ∎

To put this example in context, it is interesting to note that *every* schedule achieves a benefit of at least $1/|\mathcal{S}|$ times optimal. This follows from the observation that each term in $\mathit{accbenefit}$ has an upper bound of $\mathit{benefit}(\mathcal{S}, \emptyset)$ and the last term is equal to $\mathit{benefit}(\mathcal{S}, \emptyset)$ for all schedules. Hence, greedy scheduling is nearly as bad as an arbitrary schedule in the worst case!

6.2.2 Scheduling with Stable Subsets

Since the greedy heuristic has major shortcomings, it is desirable to find heuristics with more robust properties. We now propose a scheduling algorithm that exploits the stable subsets defined in Section 6.1 in order to avoid the mistakes that can be made by greedy scheduling. The main idea behind our approach is that we may be able to partition a large set $\mathcal{S}$ into stable subsets $C_1, \ldots, C_m$ that are small enough to find the optimal schedule T_i for each C_i in isolation. (An optimal schedule may be found in $O(|C_i| \cdot 2^{|C_i|})$ time with dynamic programming.) Since the indices in disjoint stable subsets do not interact, it is natural to choose a schedule for $\mathcal{S}$ that preserves the ordering of each T_i. In other words, we would like to arrange $\mathcal{S}$ into a sequence that contains each T_i as a subsequence. We refer to such a schedule as an *interleaving* of $T_1, \ldots, T_m$. Although there may not be any interleaving that is optimal, this strategy avoids many shortcomings of the greedy heuristic by directly accounting for index interactions.

We now present the details of our scheduling heuristic. We focus our presentation on a simplified problem of merging the schedules of two disjoint, stable subsets $C_1, C_2 \subseteq \mathcal{S}$. Specifically, we have schedules T_1, T_2 for C_1, C_2 and we would like to choose an interleaving of T_1, T_2. A solution to this problem may easily be applied to the problem of merging m schedules by iteratively merging pairs of schedules.

We can determine the *optimal* interleaving of T_1, T_2 using a dynamic programming algorithm. Let $M(k_1, k_2)$ denote the accumulated benefit of the optimal interleaving of $T_1[1, k_1]$ and $T_2[1, k_2]$. The value of $M(k_1, k_2)$ may be expressed by the following recurrence:

For $1 \le k_1 \le |C_1|$ and $1 \le k_2 \le |C_2|$,

$$\begin{aligned} M(k_1, k_2) &= \mathit{benefit}(T_1[1, k_1] \cup T_2[1, k_2], \emptyset) \\ &\quad + \max(M(k_1 - 1, k_2), M(k_1, k_2 - 1)) \\ M(k_1, 0) &= \mathit{benefit}(T_1[1, k_1], \emptyset) + M(k_1 - 1, 0) \\ M(0, k_2) &= \mathit{benefit}(T_2[1, k_2], \emptyset) + M(0, k_2 - 1) \\ M(0, 0) &= 0 \end{aligned}$$

This recurrence leads to a simple dynamic programming algorithm for choosing the optimal interleaving of two schedules. When the schedules are on stable subsets and we assume that the two schedules are optimal, we can show that the result of this merging achieves an interesting approximation ratio that we prove in the technical report [16].

THEOREM 6.3. *Let C_1, C_2 be disjoint, stable subsets of $\mathcal{S}$ and assume that T_1, T_2 are optimal schedules for C_1, C_2 respectively. Let T be the optimal interleaving of T_1, T_2 and let T^* be the optimal schedule of $C_1 \cup C_2$. Then*

$$\mathit{accbenefit}(T) \ge \frac{\mathit{accbenefit}(T^*)}{\min(|C_1|, |C_2|) + 1}.$$

The $\min()$ operation in the approximation ratio is intuitive since we expect the merged schedule to be closer to optimal when one of the optimal schedules is very short, meaning that the optimality of the longer schedule should not be "disturbed" significantly.

The approximation ratio for our merging approach is a significant improvement over the greedy heuristic whose performance can be arbitrarily close to $1/(|C_1| + |C_2| - 1)$ in the worst case, as indicated by Example 6.3. Going back to this example, we can verify by inspection that our merging heuristic will choose a schedule with the same optimal cost of T^*. Also, this schedule can be found very efficiently given that the connected components of $\mathcal{S}$ have size 1 or 2.

In practice, we expect a tuning tool to try both our merging algorithm and the greedy heuristic and return the materialization schedule that maximizes the accumulated benefit. Although it is not possible to make guarantees on the winning algorithm, the theoretical results suggest that our approach is more robust for solving difficult scheduling problems and can thus provide meaningful alternatives to greedy scheduling.

7. RELATED WORK

Several previous studies have introduced methodologies for modeling index interactions. The first reference known to us is the work of Whang et al. [17], where it is shown that interactions cannot exist between indices on different tables if the join operators satisfy specific properties. Finkelstein et al. [10] observe that these properties may not hold for common join operators such as nested-loops, and introduce a more general rule: interactions do not exist between indices that are relevant for disjoint subsets of the workload. These works, however, do not provide methods for identifying interactions or measuring their magnitude. Bruno and Chaudhuri [3] describe a heuristic approach for identifying negative index interactions and measuring the respective degree of interaction. A more systematic approach is taken by Choenni et al. [8] who show that the existence of positive (respectively, negative) interactions is directly linked to sub-modular (respectively, super-modular) query cost functions. This result, however, is restricted to a workload of single-table queries and single-column indices; moreover, it is unclear whether the cost functions of actual optimizers exhibit sub-/super-modularity with respect to the materialized index-set. Our

work provides a systematic methodology for identifying and quantifying (positive and negative) interactions without making restrictive assumptions about the optimizer. Indeed, the techniques that we develop rely solely on the optimizer being well behaved, which is an intuitive property that is easy to verify on actual systems.

The importance of index interactions has been emphasized consistently in previous works on workload-driven index selection [7, 3, 1, 2, 18, 6, 10, 17]. The developed algorithms attempt to take into account index interactions either by modeling them directly, e.g., using the aforementioned methodologies, or by making specific assumptions about the existence of interactions. For instance, Choenni et al. [8] develop an efficient solution to a variant of the index selection problem assuming that interactions can be captured through a sub-modular (or super-modular) cost function. As another example, Chaudhuri and Narasayya [7] employ the heuristic that significant interactions exist only within index-sets up to a fixed size. Our index interaction framework can serve two purposes within this context. First, it can provide specific information about index interactions that can substitute heuristic assumptions, thus leading to more robust index selection techniques. Second, it can enable new tools that help the database administrator in understanding and using the output of an index-selection algorithm. The visualization method and scheduling algorithm described in Section 6 provide two concrete examples of the latter.

As mentioned before, our work borrows the concept of the Index Benefit Graph which was introduced by Frank et al. [11]. That study also described a property of optimal plan generation that resembles the sanity property that we introduce in Section 3.1. However, there are also significant differences to our work. We prove the properties of the IBG and of the supporting machinery based on the intuitive assumption that the optimizer is well behaved. Frank et al. base their proofs on a rather complicated axiom (Property 1 in their paper) that does not have an easy natural interpretation and is thus difficult to check for real systems. Another difference is the definition of the sanity property, which in [11] implies solely that $used_q(X) = used_q(Y)$ and thus does not make any statement about the corresponding optimal plans or their cost. Our sanity property implies the equality of optimal plans and thus of execution costs, which is more informative for the detection of interactions.

8. CONCLUSIONS

This paper introduced a systematic and general methodology for computing the interactions within an index-set and with respect to a given workload. Interactions heavily affect index benefits and are thus crucial in characterizing the cost/benefit characteristics of an index-set. We developed novel algorithms for computing index interactions by leveraging the inherent structure in the cost model of a query optimizer, and presented experimental results that validate the efficiency of our methods. We also introduced two novel database tuning tools that take advantage of index interactions, thus demonstrating further the usefulness of our methods.

As part of our future work, we intend to generalize our methodology to the problem of identifying interactions within a set of indices and materialized views. Our initial investigation indicates that this extension is feasible and may require minimal extensions to the presented framework. We also plan to examine the incremental characterization of interactions for the variants of the problem where the workload or the index-set are allowed to vary. A solution for these variants is especially interesting in the context of on-line tuning tools, where the workload is observed one-statement-at-a-time and the set of relevant indices is continuously evolving.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *VLDB*, pages 496–505, 2000.

[2] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[3] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.

[4] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *Proc. VLDB Endow.*, 1(1):4–15, 2008.

[5] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD*, pages 941–952, 2008.

[6] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Trans. Knowl. and Data Eng.*, 16(11):1313–1323, 2004.

[7] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[8] S. Choenni, H. M. Blanken, and T. Chang. On the selection of secondary indices in relational databases. *Data Knowl. Eng.*, 11(3):207–233, 1993.

[9] U. Feige and P. Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[10] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[11] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *EDBT*, pages 277–292, 1992.

[12] M. Lühring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *International Workshop on Self-Managing Database Systems*, pages 450–458, 2007.

[13] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *ICDT*, pages 83–98, 2005.

[14] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated database design. In *VLDB*, pages 1093–1104, 2007.

[15] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *International Workshop on Self-Managing Database Systems*, pages 459–468, 2007.

[16] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. Technical report UCSC-SOE-09-23, UC Santa Cruz, 2009.

[17] K.-Y. Whang, G. Wiederhold, and D. Sagalowicz. Separability - an approach to physical data base design. In *VLDB*, pages 320–332, 1981.

[18] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.