Physical Design Refinement: The ‘Merge-Reduce’ Approach

NICOLAS BRUNO and SURAJIT CHAUDHURI
Microsoft Research

Physical database design tools rely on a DBA-provided workload to pick an “optimal” set of indexes and materialized views. Such tools allow either creating a new such *configuration* or adding new structures to existing ones. However, these tools do not provide adequate support for the incremental and flexible refinement of existing physical structures. Although such refinements are often very valuable for DBAs, a completely manual approach to refinement can lead to infeasible solutions (e.g., excessive use of space). In this article, we focus on the important problem of *physical design refinement* and propose a transformational architecture that is based upon two novel primitive operations, called *merging* and *reduction*. These operators help refine a configuration, treating indexes and materialized views in a unified way, as well as succinctly explain the refinement process to DBAs.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Access methods*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Physical database design, view merging and reduction, physical design refinment.

ACM Reference Format:

Bruno, N. and Chaudhuri, S. 2007. Physical design refinement: The ‘merge-reduce’ approach. ACM Trans. Datab. Syst. 32, 4, Article 28 (November 2007), 43 pages. DOI = 10.1145/1292609.1292618 http://doi.acm.org/10.1145/1292609.1292618

1. INTRODUCTION

In the last decade, automated physical design for relational databases was studied by several research groups (e.g., Agrawal et al. [2000, 2006]; Chaudhuri and Narasayya [1997, 1999]; Valentin et al. [2000]; Zilio et al. [2004]) and nowadays database vendors offer tools to automatically recommend and tune the physical design of a relational database management system—or DBMS (e.g., Agrawal et al. [2004]; Dageville et al. [2004]; Zilio et al. [2004]). These tools require the

Authors’ address: Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: {nicolasb, surajitc}@microsoft.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 0362-5915/2007/11-ART28 $5.00 DOI 10.1145/1292609.1292618 http://doi.acm.org/10.1145/1292609.1292618

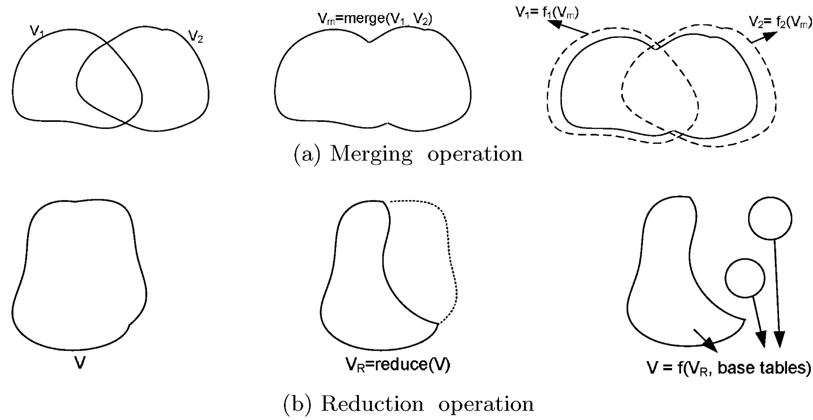
DBA to gather a representative workload, possibly using profiling tools in the DBMS, and then are able to recommend indexes and materialized views that fit in the available storage and would make the representative workload execute as fast as possible. However, the above paradigm of physical database design does not address the following key scenarios that are of great importance in enterprises:

—*Responding to incremental changes.* Gradual changes in data statistics or usage patterns may make the existing physical design inappropriate. At the same time, physical design changes are disruptive (as query plans can drastically change). For incremental changes in the data statistics or workload, DBAs desire changes in physical design that are as few as possible and yet meet the constraints on the physical design (such as storage, update cost, or limited degradation with respect to the optimal physical design). Unfortunately, an altogether new design (driven by automated tools) might be very different from the original one as these tools have very limited support for such "incremental tuning."

—*Significant manual design input.* Despite the wide availability of automated tools, the physical design process in database installations of moderate to high complexity often rely on manual input from DBAs. The need for such manual input arise due to several reasons. First, while the automated tools reduce the complexity of the physical design process, it is still nontrivial to identify a representative workload that can be used to drive the physical design in its entirety. Second, automated tools do not consider all factors that impact physical design (e.g., the impact of replication architectures). Finally, the output of a physical design tool may be fine-tuned by DBAs based on their lifelong experience. However, a configuration designed with such manual input often results in non-obvious redundancy, which increases the storage (and update) requirements. DBAs are thus interested in incrementally refining this initial configuration to remove redundancy, without significantly impacting efficiency.

These examples show the need of additional tools that go beyond statically recommending a configuration for a given workload. Specifically, we believe that it is important to automatically refine a configuration by eliminating implicit redundancy without compromising efficiency. We call this the *Physical Design Refinement* problem. Our idea is to start from the initial, possibly redundant configuration, and progressively refine it until some property is satisfied (e.g., the configuration size or its performance degradation meets a pre-specified threshold).

We can think of a refinement session as composed of a series of basic transformations, which locally change the current configuration by trading space and efficiency. In this article, we identify two atomic operations, *merging* and *reduction*, which provide this basic functionality. Merging and reduction unify different techniques that apply to indexes and materialized views proposed earlier in the literature. Intuitively (see Figure 1), merging combines two views and avoids storing their common information twice, but requires compensating actions to retrieve the original views (f_1 and f_2 in the figure). Reduction, in turn,



(a) Merging operation

(b) Reduction operation

Fig. 1. Merging and reduction as primitive operators for physical design tuning.

keeps a smaller subexpression of a view, but requires additional work (possibly accessing base tables) to recreate the original view. We can see merging and reduction as the analogous to *union* and *subset* operators for sets.[1] More generally, we can see our proposed techniques as moving from a traditional bottom-up architecture for tuning physical designs (e.g., see Chaudhuri and Narasayya [1997]; Agrawal et al. [2000]; Valentin et al. [2000]) to a transformation-based engine, which gives more flexibility and allows different optimization goals to be stated and tackled in a more unified way. This is analogous to the appearance of transformation-based query optimizers (e.g., the Cascades framework [Graefe 1995]) as an alternative to classical bottom-up query optimizers (e.g., System-R [Selinger et al. 1979]). We believe that the merge and reduce operators, along with a transformation-based engine to guide the search of configurations, have the potential of becoming the foundation of next-generation design tuning tools, by unifying seemingly disparate and ad hoc techniques into a common, flexible framework.

This article builds upon the work in Bruno and Chaudhuri [2006a] to address challenges in physical design refinement, and is structured as follows. In Sections 2 and 3 we introduce the primitive operations of merging and reduction. In Section 4 we address the *physical design refinement problem*, or PDR. In Section 5 we introduce important variations and generalizations of the original PDR problem (e.g., a variation that attempts to minimize the space used by the final configuration without exceeding a cost bound, denoted *Dual-PDR*, and a generalization that limits the number of transformations that may be applied to the original configuration, denoted *CPDR*). In Section 6 we formally define the physical design scheduling task that is used to deploy the refinement. In Section 7 we report on an experimental evaluation on the techniques of this article, and in Section 8 we review related work.

[1] We can eventually obtain every combination of elements in a family of sets by applying union and subset operations. Analogously, merging and reduction can be seen as the fundamental building blocks to manipulate designs for indexes and materialized views in a DBMS.

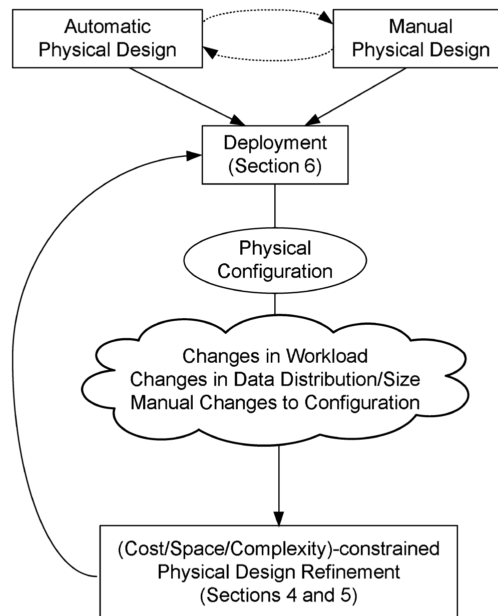


Fig. 2. Physical design refinement and scheduling cycle.

Figure 2 shows a typical interaction of a DBA and the different techniques introduced in this article. Initially, the DBA uses automatic tools in conjunction with manual tuning to obtain an initial configuration that is deployed (Section 6). After the new physical configuration is in place, the DBMS processes workloads until some triggering condition happens (e.g., changes in the data distribution, manual changes to the configuration, or other motivating example discussed earlier). At this point, the DBA intervenes and refines the physical design using some of the techniques discussed in Sections 4 and 5. Depending on the unique characteristics of the DBMS, workloads, and physical design, DBAs might select which optimization problem to address using one of PDR, Dual-PDR, or CPDR. After interacting with these refining tools, a new configuration is obtained and deployed, closing the cycle.

2. MERGING OPERATION

In this section we describe the merging operation between materialized views. Merging two materialized views V_1 and V_2 results in a new materialized view V_M that reduces the amount of redundancy between V_1 and V_2. The resulting view V_M is usually smaller than the combined sizes of V_1 and V_2 at the expense of longer execution times for queries that exploit V_M instead of the original ones. As a simple example, consider the following two materialized views:

V_1=
```
SELECT a,b
FROM R
WHERE a<10
```

V_2 =
```
SELECT b,c
FROM R
WHERE b<10
```

Suppose that the space required to materialize both V_1 and V_2 is too large. In this case, we can replace both V_1 and V_2 by the alternative V_M defined as

```
V_M =  SELECT a,b,c
       FROM R
       WHERE a<10 OR b<10
```

The main property of this alternative view is that every query that can be answered using either V_1 or V_2 can also be answered by V_M. The reason is that we can rewrite both V_1 and V_2 in terms of V_M as follows:

```
V1 ≡  SELECT a,b          V2 ≡  SELECT b,c
      FROM V_M                  FROM V_M
      WHERE a<10                WHERE b<10
```

If the tuples that satisfy both $R.a < 10$ and $R.b < 10$ are a significant fraction of R, the size of V_M might be much smaller than the sum of the sizes of the original views V_1 and V_2. In fact, V_M is the smallest view that can be used to generate both V_1 and V_2. It is also important to note that queries that are answered using V_1 or V_2 are less efficiently answered by V_M. The reason is that V_M is a generalization of both V_1 and V_2 and contains additional, nonrelevant tuples with respect to the original views. In other words, by merging V_1 and V_2 into V_M we are effectively trading space for efficiency. We now formally define the view merging operation.

2.1 Formal Model

To formalize the view merging operation, we consider three query languages. Let $\mathcal{L}_I$ be the language that defines input views, $\mathcal{L}_M$ the language that defines merged views, and $\mathcal{L}_C$ the language that defines compensating actions to recreate the original views in terms of the merged view.

Definition 2.1. Given V_1 and V_2 from $\mathcal{L}_I$, we denote $V_M = V_1 \oplus V_2$ the merging of V_1 and V_2 when the following properties hold:

(1) V_M belongs to $\mathcal{L}_M$.
(2) $C_1(V_M) \equiv V_1$ and $C_2(V_M) \equiv V_2$ for some $C_1(V_M)$ and $C_2(V_M)$ in $\mathcal{L}_C$.
(3) If the view matching algorithm matches V_1 or V_2 for a subquery q, it also matches V_M for q (a view matching algorithm matches a view V for a subquery q if q can be answered from V).
(4) V_M cannot be further restricted with additional predicates and continue to satisfy the previous properties.

View merging and view matching are indeed related problems. The idea of view merging is to obtain, for a given pair of views, some sort of *minimal* view that can be matched to a subquery whenever the original ones do. Although both problems are different, some of the technical details that are introduced below are related to those in the view matching literature.

As an example, suppose that both $\mathcal{L}_I$ and $\mathcal{L}_M$ are the subset of SQL that only allows simple conjunctions over single tables, and $\mathcal{L}_C$ is the full SQL language.

Consider the following views:

V_1 =
```
SELECT a,b
FROM R
WHERE 10<a<30 AND 10<b<30
```

V_2 =
```
SELECT a,b
FROM R
WHERE 20<a<40 AND 20<b<40
```

Merging the views above results in

$V_1 \oplus V_2$ =
```
SELECT a,b
FROM R
WHERE 10<a<40 AND 10<b<40
```

In this case, however, the merged $V_1 \oplus V_2$ can be larger the combined sizes of the input views, as this depends on the number of tuples that satisfy ($10 < a < 20$ AND $30 < b < 40$) or ($30 < a < 40$ AND $10 < b < 20$) (and therefore would be included in $V_1 \oplus V_2$ even though they do not belong to either V_1 or V_2). In contrast, suppose that we relax $\mathcal{L}_M$ to also include disjunctions. In this case,

$V_1 \oplus V_2$ =
```
SELECT a,b
FROM R
WHERE (10<a<30 AND 10<b<30) OR (10<a<40 AND 20<b<40)
```

Now $V_1 \oplus V_2$ is no larger than V_1 and V_2 put together, because the tuples that satisfy ($20 < a < 30$ AND $20 < b < 30$) are included in both V_1 and V_2 but only once in $V_1 \oplus V_2$. In general, merged views can be larger than their combined inputs even when there is redundancy, depending on the expressive power of $\mathcal{L}_M$.

2.2 The $\mathcal{L}_{\texttt{MV}}$ Language

In this section we focus on specific query languages and address the view merging operation in detail. Specifically, we set $\mathcal{L}_I$ and $\mathcal{L}_M$ as the subset of SQL that can be used in a DBMS for materialized view matching (we denote this language as $\mathcal{L}_{\texttt{MV}}$). A view is then given by the following expression:

SELECT $S_1, S_2, \ldots$	– project columns (see below)
FROM $T_1, T_2, \ldots$	– tables in the database
WHERE J_1 AND J_2 AND ...	– equi-join predicates
R_1 AND R_2 AND ...	– range predicates (see below)
Z_1 AND Z_2 AND ...	– residual predicates (see below)
GROUP BY $G_1, G_2, \ldots$	– grouping columns

where

—S_i are either base-table columns, column expressions, or aggregates. If the group by clause is present, then every S_i that is not an aggregate must be either equal to one of the G_j columns or be an expression in terms of them.

—R_i are range predicates. The general form of a range predicate is a disjunction of open or closed intervals over the same column (point selections are special cases of intervals). An example of a range predicate is `(1<a<10 OR 20<a<30)`.

—Z_i are residual predicates, that is, the set of predicates in the query definition that cannot be classified as either equi-join or range predicates.

In other words, we can express in $\mathcal{L}_{\texttt{MV}}$ the class of SPJ queries with aggregation. The reason that predicates are split into three disjoint groups (join, range, and residual) is pragmatic. During query optimization, it is easier to perform subsumption tests for view matching if both the view and the candidate subquery are written in this structured way. Specifically, we can then perform simple subsumption tests component by component and fail whenever any of the simple tests fails. For instance, we check that the join predicates in the query are a superset of the join predicates in the view, and the range predicates (column by column) in the query are subsumed by the corresponding ones in the view. Some subsumption tests are more complex than others, notably when group-by clauses are present. We note that this procedure focuses on simplicity and efficiency and therefore can miss some valid matchings due to complex logical rewritings that are not considered by the optimizer. Specifically, consider the case of residual predicates. The problem of determining whether two predicates are equivalent can be arbitrarily complex.[2] For that reason, the matching procedure that we consider just checks that every conjunct in the residual predicate of the view appears (syntactically) in the candidate query. Otherwise, although the view can still subsume the query, no match is produced.

We simplify the notation of a view in $\mathcal{L}_{\texttt{MV}}$ as (S, T, J, R, Z, G) where S is the set of columns in the select clause, T is the set of tables, J, R, and Z are the sets of join, range, and residual predicates, respectively, and G is the set of grouping columns. In this work we restrict the merging operation so that the input views agree on the set of tables T. The reason is twofold. On one hand, many top-down optimizers restrict the view matching operation to queries and views that agree on the input tables (presumably, if a candidate view contains fewer tables than the input query q, it should have matched a subquery of q earlier during optimization). On the other hand, merging views with different input tables can be done by combining the reduce operator of Section 3 and the merging operation as defined in this section. We next define the merging operator in $\mathcal{L}_{\texttt{MV}}$.

2.2.1 *Case* 1: *No Grouping Columns.* Consider merging $V_1 = (S_1, T, J_1, R_1, Z_1, \emptyset)$ and $V_2 = (S_2, T, J_2, R_2, Z_2, \emptyset)$. If the merging language were expressive enough, we could define $V_1 \oplus V_2$ as

SELECT $S_1 \cup S_2$
FROM T
WHERE (J_1 AND R_1 AND Z_1) OR (J_2 AND R_2 AND Z_2)

which satisfies properties 2 and 4 in Definition 2.1. To satisfy property 1 (i.e., rewriting $V_1 \oplus V_2$ in $\mathcal{L}_{\texttt{MV}}$), we have no option but consider the whole predicate in the WHERE clause as a single conjunctive residual predicate Z. The problem is that now the merged view would not be matched whenever V_1 or V_2 are matched (property 3) because of the simple procedures used during view matching in

[2]Consider a table with four integer columns (x, y, z, n). Checking that predicates $x + 1 = x$ and $x^n + y^n = z^n \wedge n > 2$ are equivalent is essentially the same as proving Fermat's last theorem. It took over 300 years to prove that specific conjecture; expecting such capabilities from a view matching algorithm is unrealistic.

general and with respect to residual predicates in particular. We need to obtain the smallest view V_M that is in $\mathcal{L}_{\mathtt{MV}}$ and satisfies property 3. For that purpose, we rewrite the above "minimal" predicate as follows:

$$(\mathtt{J}_1 \wedge \mathtt{R}_1 \wedge \mathtt{Z}_1) \vee (\mathtt{J}_2 \wedge \mathtt{R}_2 \wedge \mathtt{Z}_2) \equiv (\mathtt{J}_1 \vee \mathtt{J}_2) \wedge (\mathtt{R}_1 \vee \mathtt{R}_2) \wedge (\mathtt{Z}_1 \vee \mathtt{Z}_2) \wedge \mathtt{C},$$

where $\mathtt{C}$ is the conjunction of all crossed disjuncts $((\mathtt{J}_1 \vee \mathtt{R}_2) \wedge (\mathtt{R}_1 \vee \mathtt{Z}_2) \wedge \cdots)$. Our strategy is to relax this expression until we obtain a predicate that can be written in $\mathcal{L}_{\mathtt{MV}}$ and matches any candidate query that is matched by the original views. Although this procedure seems in general to introduce a lot of redundancy and result in larger views, we experimentally determined that in real-world scenarios this is not the case.

We first relax the expression above by removing the conjunct $\mathtt{C}$. The reason is that it leaves us with three conjuncts ($\mathtt{J}_1 \vee \mathtt{J}_2$, $\mathtt{R}_1 \vee \mathtt{R}_2$, and $\mathtt{Z}_1 \vee \mathtt{Z}_2$), which we next map into the three groups of predicates in $\mathcal{L}_{\mathtt{MV}}$. First consider $\mathtt{J}_1 \vee \mathtt{J}_2$ and recall that each J_i is a conjunct of equi-join predicates. We cannot simply use $\mathtt{J}_1 \vee \mathtt{J}_2$ in the resulting view because the language specifies that this must be a conjunction of simple equi-joins (i.e., no disjunctions are allowed). We rewrite

$$\mathtt{J}_1 \vee \mathtt{J}_2 \equiv \left(\mathtt{J}_1^1 \wedge \mathtt{J}_1^2 \wedge \mathtt{J}_1^3 \wedge \cdots\right) \vee \left(\mathtt{J}_2^1 \wedge \mathtt{J}_2^2 \wedge \mathtt{J}_2^3 \wedge \cdots\right) \equiv \bigwedge_{\mathtt{i},\mathtt{j}} \left(\mathtt{J}_1^{\mathtt{i}} \vee \mathtt{J}_2^{\mathtt{j}}\right)$$

and relax this predicate as follows: we keep each (i, j) conjunct for which $J_1^i \equiv J_2^j$ and discard (i.e., relax) the remaining ones. We obtain then $\bigwedge_{J^k \in J_1 \cap J_2} J^k$ as the set of join predicates in the merged view. Note that this predicate can be much more general than the original $J_1 \vee J_2$, but the view matching procedure would match V_m with respect to the join subsumption test in this case. We use the same idea for $Z_1 \vee Z_2$ and therefore the residual predicate for V_m is $\bigwedge_{Z^k \in Z_1 \cap Z_2} Z^k$.

It turns out that we can do better for range predicates $R_1 \vee R_2$ due to their specific structure. Using the same argument, we first rewrite $R_1 \vee R_2$ as $\bigwedge_{\mathtt{i},\mathtt{j}} (\mathtt{R}_1^{\mathtt{i}} \vee \mathtt{R}_2^{\mathtt{j}})$ where each R_1^i and R_2^j are disjunctions of open or closed intervals over some column. As before, if R_1^i and R_2^j are defined over different columns, we discard that conjunct. However, if they are defined over the same column, we keep the predicate even when R_1^i and R_2^j are not the same, by taking the union of the corresponding intervals (we denote this operation with the symbol $\bigsqcup$). To avoid missing some predicates, we first add conjuncts $-\infty < x < \infty$ to one of the range predicates if column x is only present in the other range predicate (it does not change the semantics of the input predicates but restricts further the result). Also, if after taking the union the predicate over some column x becomes $-\infty < x < \infty$, we discard this conjunct from the result. As an example, consider

$$\begin{array}{rccccccc}
R_1 = & (10<a<20 \vee 30<a<40) & \wedge & 20<b<30 & \wedge & c<40 & & \\
R_2 = & 15<a<35 & \wedge & 10<b<25 & \wedge & c>30 & \wedge & 10<d<20 \\
\hline
R_1 \bigsqcup R_2 = & 10<a<40 & \wedge & 10<b<30 & & & \wedge & 10<d<20
\end{array}$$

After obtaining join, range, and residual predicates as described above, we assemble the set of columns in the merged view. At a minimum, this set must

contain the union of columns present in both input views. However, this is not enough in general, as illustrated next. Consider for instance

$V_1 =$
```
SELECT a
FROM R
WHERE 10<c<20
```

$V_2 =$
```
SELECT b
FROM R
WHERE 15<c<30
```

The candidate merged view V = `SELECT a,b FROM R WHERE 10<c<30` does not satisfy property 2 in Definition 2.1 because V_1 and V_2 cannot be obtained from V. The reason is that we need to apply additional predicates to V (`c<20` to obtain V_1 and `15<c` to obtain V_2), but V does not expose column `c`. For that reason, we need to add to the set of columns in the merged view all the columns that are used in join, range, and residual predicates that are eliminated in the merged view. Similarly, if some range predicate changed from the input to the merged view, we need to add the range column as an output column, or otherwise we would not be able to reconstruct the original views. To summarize, the merging of two views as described in this section is as follows:

$$
\begin{array}{rllllll}
V_1 = (& S_1 & , T & , J_1 & , R_1 & , Z_1 & , \emptyset\ ) \\
\oplus\ V_2 = (& S_2 & , T & , J_2 & , R_2 & , Z_2 & , \emptyset\ ) \\
\hline
V_1 \oplus V_2 = (& S_1 \cup S_2 \cup \{ \text{required columns} \} & , T & , J_1 \cap J_2 & , R_1 \bigsqcup R_2 & , Z_1 \cap Z_2 & , \emptyset\ )
\end{array}
$$

We note that all the transformations mentioned above take into account column equivalence. If both input views contain a join predicate R.x = S.y, then the range predicates R.x < 10 and S.y < 10 are considered to be the same.

Example 2.2. The following example illustrates the ideas described in this section. If V_1 and V_2 are materialized views as described below:

$V_1 =$
```
SELECT x,y
FROM R,S
WHERE R.x=S.y AND
      10<R.a<20 AND
      R.b<10 AND
      R.x+S.d<8
```

$V_2 =$
```
SELECT y,z
FROM R,S
WHERE R.x=S.y AND
      15<R.a<50 AND
      R.b>5 AND R.c>5 AND
      S.y+S.d<8 AND R.d*R.d=2
```

the merge of V_1 and V_2 is

$V_1 \oplus V_2$ =
```
SELECT x,y,z,a,b,c,d
FROM R,S
WHERE R.x=S.y AND
      10<R.a<50 AND
      R.x+S.d<8
```

2.2.2 *Case* 2: *Grouping Columns.* We now consider the case of merging views that involve `group-by` clauses. Grouping operators partition the input relation into disjoint subsets and return a representative tuple and some aggregates from each group. Conceptually, we see a `group-by` operator as a postprocessing step after the evaluation of the SPJ subquery. Consider the

merged view obtained when the grouping columns are eliminated from the input views. If the group-by columns in the input views are different, each view partitions the input relation in different ways. We then need to partition the merged view in the coarsest way that still allows us to recreate each input view. For that purpose, the set of group-by columns in the merged view must be the union of the group-by columns of the input views. Additionally, each column that is added to the select clause due to predicate relaxation in the input views must also be added as a grouping column. Note that we need to handle a special case properly. If one of the input views contains no group-by clause, the merged view should not contain any group-by clause either, or else we would compromise correctness (i.e., we implicitly define the union of a set of columns and the empty set as the empty set[3]). In these situations, we additionally unfold all original aggregates into base-table columns so that the original aggregates can be computed from the resulting merged view. To summarize, we define $(S_1, T, J_1, R_1, Z_1, G_1) \oplus (S_2, T, J_2, R_2, Z_2, G_2)$ as $(S_M, T, J_1 \cap J_2, R_1 \bigsqcup R_2, Z_1 \cap Z_2, G_M)$ where

—S_M is the set of columns obtained in the no group-by case, plus the group-by columns if they are not the same as the input views; if the resulting $G_M = \emptyset$, all aggregates are unfolded into base-table columns;

—$G_M = (G_1 \cup G_2) \cup$ columns added to S_M (note that $G_1 = \emptyset \vee G_2 = \emptyset \Rightarrow G_M = \emptyset$).

Example 2.3. The following example illustrates the ideas in this section. If V_1, V_2, and V_3 are materialized views as described below:

$V_1 =$
```
SELECT R.x,SUM(S.y)
FROM R,S
WHERE R.x=S.y AND
      10<R.a<20
GROUP BY R.x
```

$V_2 =$
```
SELECT R.x,R.z
FROM R,S
WHERE R.x=S.y AND
      15<R.a<50
```

$V_3 =$
```
SELECT S.y,SUM(S.z)
FROM R,S
WHERE R.x=S.z AND
      10<R.a<25
GROUP BY S.y
```

then the following equalities hold:

$V_1 \oplus V_2 =$
```
SELECT R.x,R.a,S.y,R.z
FROM R,S
WHERE R.x=S.y AND
      10<R.a<50
```

$V_1 \oplus V_3 =$
```
SELECT R.x,S.y,R.a,SUM(S.y),SUM(S.z)
FROM R,S
WHERE R.x=S.y AND
      10<R.a<25
GROUP BY R.a,R.x,S.y
```

[3]For tables with unique constraints, we can define the set of group-by columns of a query without a group-by clause as the set of all columns in the table, and thus keep the definition of union unchanged. However, this is not correct for tables with duplicate values, because a group-by clause with all columns eliminates duplicate rows and therefore is not equivalent to the query without the group-by clause.

Note that in order to recreate the original views in the presence of general algebraic aggregates, we sometimes need to add additional columns in the merged view (e.g., `SUM(c)` and `COUNT(*)` for an original aggregate `AVG(c)`).

2.3 Indexes over Materialized Views

So far we have discussed the merging operation applied to materialized views, without paying attention to indexes over those materialized views. In reality, each materialized view is associated with a set of indexes, and those indexes are used during query processing. Previous work in the literature has considered index merging and view merging as separate operations. We know describe how we can handle both structures in a unified manner. For this purpose, we consider all indexes as defined over some view (base tables are also trivial views, so this definition includes regular indexes as well). Specifically, for a sequence of columns I and a view V that contains all I columns in its `SELECT` clause, we denote $I \mid V$ the index with columns I over the materialized view V. For the special case $I = \emptyset$, we define $\emptyset \mid V$ to be the unordered heap containing all the tuples in V (for simplicity, we use V and $\emptyset \mid V$ interchangeably).

2.3.1 *Unified Merging Operator.* We now define the merging of two arbitrary indexes over views. In this section we overload the operator $\oplus$ to operate over indexes, views, or indexes over views (we explicitly state which case we are referring to when this is not clear from the context). Consider the simplest case of merging two indexes defined over the same view. In this case:

$$(I_1 \mid V) \oplus (I_2 \mid V) = (I_1 \oplus I_2) \mid V,$$

where $I_1 \oplus I_2$ is the traditional index-merging operation as defined in Bruno and Chaudhuri [2005] and Chaudhuri and Narasayya [1999]. That is, $I_1 \oplus I_2 = I_M$ where I_M contains all columns in I_1 followed by all columns in $I_2 - I_1$ As an example, we have that $([a, b, c] \mid V) \oplus ([b, a, d] \mid V) = [a, b, c, d] \mid V$. Index merging is very effective when the input indexes share a common prefix. If this is not the case (e.g., $[a, b] \oplus [c] = [a, b, c]$), the penalty of replacing usages of the second input index with the merged index is more pronounced. However, the merged index is still better than the alternative of using the primary index or heap by enabling a narrower scan over the relevant data (e.g., if index $[c]$ were used to evaluate a nonsargable predicate on column c, the merged index $[a, b, c]$ would still be effective, especially if the underlying table contained many columns).

To address the general case, we need to first introduce the notion of *index promotion*. Consider an index $I \mid V$ and suppose that $V_M = V \oplus V'$ for some view V'. Promoting I over V to V_M (denoted $I \uparrow V_M$) results in an index over V_M that can be used (with some compensating action) whenever $I \mid V$ is used. This promoted index contains all columns in the original index followed by every column that was added to the select clause in V_M[4]. For instance,

[4]Other column orderings are possible, but we omit these details for simplicity.

consider

$V_1 =$
```
SELECT x,y
FROM R
WHERE 10<a<20
```
$V_2 =$
```
SELECT y,z
FROM R
WHERE 15<a<30
```

and the merged view

$V_1 \oplus V_2 =$
```
SELECT a,x,y,z
FROM R
WHERE 10<a<30
```

We then have that $[x] \uparrow (V_1 \oplus V_2)$=$[x, a]$. Using index promotion, we now define the merging of two indexes over views as follows:

$$(I_1 \mid V_1) \oplus (I_2 \mid V_2) = ((I_1 \oplus I_2) \uparrow (V_1 \oplus V_2)) \mid (V_1 \oplus V_2).$$

That is, we first obtain the merged index $I_1 \oplus I_2$, then the view $V_1 \oplus V_2$, and finally we promote the merged index to the merged view.

3. REDUCTION OPERATION

In the previous section we described a mechanism to decrease the amount of redundancy between a pair of indexes over views. The idea was to merge them into a new index over a view that might be smaller than the combined inputs, but at the same time less efficient to answer queries. In this section we present a second operator that works over a single input.

Specifically, we exploit the fact that when the query optimizer attempts to match a query expression q, it will consider not only views that subsume q completely, but also views that subsume some of the subexpressions of q. As a simple example suppose that the optimizer is matching the following query expression:

$$q = \Pi_{R.a,R.b,S.c}(\sigma_{R.a=15}(R \bowtie_{R.x=S.y} S)).$$

In this case, the view matching engine would consider all available views V that subsume query expression q. If some view V matches q, the expression is rewritten using V and compensating actions (e.g., $q = \sigma_{R.a=15}(V)$ for $V = R \bowtie_{R.x=S.y} S$). However, query optimizers would also consider views that match subexpressions of q, such as, for example, views that subsume the following subexpression of q (which omits table S but additionally projects column $R.x$ so that a compensating join can be applied):

$$q' = \Pi_{R.a,R.b,R.x}(\sigma_{R.a=15}(R)).$$

Since $q = \Pi_{q'.a,q'.b,S.c}(q' \bowtie_{q'.x=S.y} S)$, we can recreate q from any view V' that matches q' by additionally performing a join with the primary index of S. In general, we can restrict an index over a view with some of its subexpressions, and then apply compensating actions to recreate the original structure. We call this operation *reduction* and denote it with the symbol ρ.

3.1 Formal Model

To formalize the view reduction operation, we again consider three query languages. Let $\mathcal{L}_I$ be the language that defines input views, $\mathcal{L}_R$ the language that defines reduced views, and $\mathcal{L}_C$ the language that defines compensating actions to recreate the original views in terms of the reduced view.

Definition 3.1. Given V from $\mathcal{L}_I$, we denote $V_R = \rho(V)$ a reduction of V when the following properties hold:

(1) V_R belongs to $\mathcal{L}_R$.
(2) $C(V_R) \equiv V$ for some $C(V_R)$ in $\mathcal{L}_C$.
(3) If the view matching algorithm matches V for a query expression q, it will attempt (and succeed) matching V_R for a subquery of q.

We next address the reduction operation in detail when both $\mathcal{L}_I$ and $\mathcal{L}_R$ are the $\mathcal{L}_{\texttt{MV}}$ language (see Section 2.2) and for commonly used view matching engines.

3.2 Reduction in the $\mathcal{L}_{\texttt{MV}}$ Language

For efficiency purposes, query optimizers restrict the subqueries that are considered for view matching for a given query expression q. Most often, these optimizers only consider subexpressions q' with fewer joins than q, but containing all applicable predicates in q that affect the tables in the subexpression q'. In these common scenarios, the reduction operation takes an index on a view $IV \in \mathcal{L}_{\texttt{MV}}$, a set of tables T', and a set of columns K' as inputs, and returns a new index $\rho(IV, T', K')$. For an index $I \mid V$, where $V = (S, T, J, R, Z, G)$, the operational semantics of $\rho((I \mid V), T', K')$ are given in three steps as follows:

(1) If $T' \not\subset T$, the reduction is ill defined and we stop. Otherwise, we obtain the reduced version of V that only references tables T', defined as $V' = (S', T', J', R', Z', G')$, where

 —$J' \subseteq J$, $R' \subseteq R$, and $Z' \subseteq Z$, where each base-table column referenced in J', R' and Z', refers exclusively to tables in T';
 —S' contains the subset of columns in S that belong to tables in T' plus all columns in T' referenced in $J - J'$, $R - R'$, and $Z - Z'$;
 —if $G \neq \emptyset$, G' contains all the columns in G that belong to tables in T' plus all columns in S'-S; otherwise, $G' = G = \emptyset$.

 If V' contains Cartesian products, we consider the reduction invalid and we stop (a Cartesian product does not provide any efficiency advantage and it is always much larger than the input relations).

(2) We obtain I' from I by first removing all columns that do not belong to tables in T', and then adding all columns in S' (this step is similar to $I \uparrow V'$).
(3) If $K' \not\subseteq I'$, the reduction is ill defined and we stop. Otherwise, we define $\rho((I \mid V), T', K') = K' \mid V'$.

Example 3.2. The following example illustrates the ideas described in this section. If V is the view defined below:

```
V =  SELECT R.c, S.c
     FROM R, S
     WHERE R.x=S.y AND
           10<R.a<50 AND
           20<S.a<30 AND
           R.b+S.b<10
     GROUP BY R.c, S.c
```

then $\rho\big([R.c, S.c] \mid V, \{R\}, \{R.c, R.x\}\big) = ([R.c, R.x] \mid V')$, where

```
V' =  SELECT R.c, R.b, R.x
      FROM R
      WHERE 10<R.a<50
      GROUP BY R.c, R.b, R.x )
```

4. PHYSICAL DESIGN REFINEMENT

We now formally define the physical design refinement problem motivated in the introduction, using merging and reduction as the basic building blocks. Consider a physical database configuration $C = \{I_1 \mid V_1, \ldots, I_n \mid V_n\}$ composed of indexes over views (recall that all base-table indexes are defined over trivial views). We assume that C was obtained by tuning the DBMS for a typical workload by either a skilled DBA or some automated tool (e.g., Agrawal et al. [2004]; Dageville et al. [2004]; Zilio et al. [2004]). The size of a configuration C is the combined size of all indexes in C plus the size of heaps for indexes on views that do not have a primary index in C (we need a primary index or heap for each view):

$$size(C) = \sum_j size(I_j \mid V_j) + \sum_{V_k \text{ without primary index in } C} size(\emptyset \mid V_k).$$

Now suppose that after some time the database grows, or the DBA manually adds additional indexes (see Section 1), and $size(C)$ becomes larger than the allocated space. We would like to obtain a configuration that fits in the storage constraint without compromising the *quality* of the original configuration C (we measure the quality of C as the impact C has on a "representative" workload, as explained in the next section). Instead of considering every possible index for the new configuration, we restrict our search to those that are either in the initial configuration or can be derived from it via a series of merging and reduction operations. The rationale is that in this way we can succinctly explain the refinement process and analyze the impact of the physical changes more easily. Moreover, this alternative allows to *locally* adapt each original execution plan with local compensating actions so that it uses the views in the new configuration. To understand this, consider the query execution plan at the left of Figure 3. The highlighted subplan in the figure seeks an index $I = (a, b, c \mid V)$ using predicate $a < 10$ and outputs columns b and c upward in the tree. Now suppose that we reduce index I into $I' = (a, b \mid V)$. Clearly, we cannot simply replace I with I' in plan P, because I' does not contain the required column c.

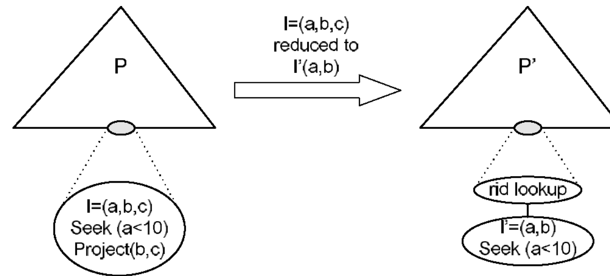


Fig. 3. A local transformation to reuse a query plan with a reduced index.

However, we can construct an alternative subplan using a compensating action (in this case, a record-id lookup into the primary index of the view V) and replace the original subplan that uses index I in P with the new subplan that uses I' but gives equivalent results. The resulting plan P' (shown at the right in Figure 3) is valid and equivalent to P, but uses the reduced index I' instead of the original I. Therefore, if a DBA were used to the original execution plan P, a change in the physical design from I to I' would present only limited changes to the query execution plan. (We note, however, that DBAs can always reoptimize the query and obtain the optimal plan under the new configuration).

4.1 Problem Statement

Before introducing the physical design refinement problem, we define the search space by introducing the *closure* of a configuration under the merging and reduction operations:

Definition 4.1. Let C be a configuration and let C_i ($i \geq 0$) be defined as follows (see Sections 2 and 3 for the formal definition of operators $\oplus$ and ρ):

—$C_0 = C$.

—$C_{i+1} = C_i \cup \{IV_1 \oplus IV_2 \text{ for each compatible } IV_1, IV_2 \in C_i\}$
$\cup \{\rho(IV, T, K) \text{ for each } IV \in C_i \text{ and valid choices of } T \text{ and } K\}$.

We define $closure(C) = C_k$, where k is the smallest integer that satisfies $C_k = C_{k+1}$.

In words, the closure of a configuration C is the set of all indexes over views that are either in C or can be derived from elements of C through a series of merging and reduction operations. Our goal is to obtain a subset of this closure that fits in the available storage and is as efficient as possible for a given representative workload.

Definition 4.2 (*Physical Design Refinement (PDR) Problem*). Given a configuration $C = \{I_1 \mid V_1, \ldots, I_n \mid V_n\}$, a representative workload W,[5] and a

[5]A representative workload can be produced by a skilled DBA based on a broad knowledge of the DBMS usage. If this is not possible, Section 4.2 explores different alternatives to automatically generate representative workloads.

storage constraint B, we define PDR(C, W, B) as the refined configuration C' such that

(1) $C' \subseteq closure(C)$;
(2) $size(C') \leq B$;
(3) $\sum_{q_i \in W}(\alpha_i \cdot\ cost(q_i, C')$ is minimized, where $cost(q, C)$ is the optimizer estimated cost of query q under configuration C, and α_i is an optional weight associated to query q_i.

Unfortunately, the *PDR* problem is NP-hard, as we prove in Appendix A.

4.2 Obtaining a Representative Workload

The optimizing function in the PDR problem described above is a measure of *quality* of each candidate configuration C, which in turn is defined as the expected cost of a *representative workload* under C. In this section we present three approaches to generate such representative workload when not explicitly provided, and discuss their relative benefits and costs.

4.2.1 *Inferred Workload.* This alternative is the cheapest to obtain and can be used in any situation. The insight is to note that the current configuration was obtained as the result of a tuning session either by a DBA or an automated tool. It is then expected that all indexes over views in the current configuration are somehow useful in answering queries in the actual workload. We then propose to *infer* a hypothetical workload with queries that mimic the functionality of each index that is present in the current configuration. We assume that if a new configuration can efficiently process such hypothetical workload, the benefits of the original indexes would be preserved. An index can be used in one of three ways: (i) to scan a vertical fragment of a view, (ii) to scan a *sorted* vertical fragment of a view, and (iii) to seek tuples from a view based on a sargable condition on its key columns. We then associate each index $IV = I \mid (S, T, R, J, Z, G)$ in C with a set of queries, called *queries(IV)*, and define the inferred workload $inferredW(C) = \cup_{IV \in C} queries(IV)$. Each query in *queries(IV)* stresses a different kind of index usage for *IV*, as shown below:

Scan	Ordered Scan	Seek[6]
`SELECT I`	`SELECT I`	`SELECT I`
`FROM T`	`FROM T`	`FROM T`
`WHERE R AND J AND Z`	`WHERE R AND J AND Z`	`WHERE R AND J AND Z`
`GROUP BY G`	`GROUP BY G`	`[ AND "σ(prefix I)" ]`
	`ORDER BY I`	`GROUP BY G`
		`[ HAVING "σ(prefix I)" ]`

The inferred workload described above is very convenient. First, it is very simple to implement and does not require server changes. Also, it only depends on the current configuration and therefore can be used in absence of

[6]We use a `HAVING` clause if $G \neq \emptyset$, and a `WHERE` clause otherwise. In both cases, the expression "σ(`prefix I`)" refers to a sargable predicate over a prefix of the index columns.

any additional knowledge on the DBMS usage. Finally, as we show experimentally, it is competitive in a variety of scenarios. We should point out that, although inferred workloads are an attractive approach, in some situations the resulting workloads are not quite representative, as illustrated in the following example.

Example 4.3. Consider a workload consisting of n_1 occurrences of query q_1 and n_2 occurrences of query q_2, where

$q_1 =$ `SELECT a FROM R WHERE a<10`
$q_2 =$ `SELECT b FROM S WHERE b<10`

In this situation, the optimal configuration that might be obtained by a DBA or automated tool is $C = \{I_1, I_2\}$ where $I_1 = (R.a)$ and $I_2 = (S.b)$. Now suppose that the space taken by C is too large and we decide to refine C. The inferred workload, *inferredW*(C), consists of the following queries:

$$inferredW(C) = \{ \texttt{SELECT a FROM R,} \\ \texttt{SELECT a FROM R ORDER BY a,} \\ \texttt{SELECT a FROM R where } \sigma(a)\texttt{,} \\ \texttt{SELECT b FROM S,} \\ \texttt{SELECT b FROM S ORDER BY b,} \\ \texttt{SELECT b FROM S where } \sigma(b) \quad \}$$

Further assume that tables R and S are of the same size, and columns a and b are of the same width. In this case, the inferred workload *inferredW*(C) is symmetric, and *PDR (C, inferredW(C), B)* would arbitrarily choose to keep either I_1 or I_2 (there is no possibility of index merging or reduction, so $C = closure(C)$ in this situation). Suppose, without loss of generality, that PDR results in a new configuration $C' = \{I_1\}$ (independently of the relative frequencies n_1 and n_2). Then, by making $n_2 >> n_1$, we can get an arbitrarily suboptimal C' compared to the alternative configuration $\{I_2\}$. The reason is that the inferred workload does not account for the relative *importance* of each index in the original configuration.

4.2.2 *Profiled Workload.* In the previous section we argued that unless we track the frequency of queries in the actual workload and the different types of index usages, we might not be able to infer representative workloads. To address this drawback, we propose to track, with very small overhead, the optimization and execution of queries in the database. In this way, we are able to obtain accurate weights to refine the queries in the inferred workload *inferredW*. Specifically, each time a query is optimized, we traverse the resulting execution plan and identify each index usage (i.e., scan, sorted scan, or seek). Additionally, for "seek usages" we obtain the (expected) number of tuples that are sought by the index. Then, during normal query execution, we maintain four counters attached to each index in the database using the information gathered during optimization: *total_scan_usages*, *total_sorted_scan_usages*, *total_seek_usages*, and *total_sought_tuples*. At any time that we need a *profiled workload*, we first obtain the inferred workload as before and use the maintained index counters

to assign weights α_i to each inferred query. For the "σ`(prefix I)`" predicates, we generate a conjunctive predicate whose cardinality equals the average number of sought tuples *total_sought_tuples/total_seek_usages*.

Profiled workloads require slightly more overhead at runtime than inferred ones, since we need to analyze each query during optimization and maintain some counters as queries are executed. At the same time, the resulting workloads are more representative, as we show in the experimental evaluation. In fact, profiled workloads can better handle scenarios on which the workload changes compared to the workload that was used to obtain the initial configuration.

PROPERTY 4.4. *Let W be a workload, and W_p be the profiled workload generated after W was processed in the DBMS. If we are only able to optimize queries based on local transformations, PDR(C, W, B) = PDR(C, W_p, B) (i.e., W and W_p are indistinguishable for the purposes of refinement).*

The crucial assumption in the property above is that the optimizer, faced with different configurations, can only *locally* transform the plan that was optimized under the original configuration (see Figure 3). Specifically, we do not allow join reordering, group-by pushing, or other complex transformations. Instead, all the optimizer can do is finding the best access plan for each base-table (or view) predicate in the execution plan. If that is the case, the difference in cost of a query plan under two configurations is only affected by the choice of a fixed set of access path requests (see Bruno and Chaudhuri [2006b, 2005] for more details). But W_p encodes such requests, and therefore captures the variable portion of the execution plans in W. The difference in cost between two configurations would therefore be the same whether we used W or W_p, and thus PDR would result in the same answer. Of course, in reality query optimization goes beyond local transformations. Nevertheless, the property illustrates that profiled workloads have stronger guarantees than inferred ones for the purposes of physical design refinement.

4.2.3 *Fully Logged Workload.* Naturally, the most accurate way of generating a representative workload is to fully log the queries that are executed by the DBMS, as it is done in the context of traditional physical design tools. This technique results in more overhead than the previous two because we need to log all the queries that are executed in the DBMS,[7] but obviously is the most accurate way to generate a representative workload.

For simplicity, in the rest of this section we use the inferred workload, *inferredW*, as the default input to PDR (the results are analogous for other input workloads as well). Since the workload *inferredW*(C) is automatically generated from the initial configuration C, we use PDR(C, B) as a shorthand for PDR$(C,$ *inferredW*$(C), B)$.

[7]We can use sampling and its stratified variants [Chaudhuri et al. 2002; König and Nabar 2006] to reduce the logging overhead at the expense of a loss in representability.

4.3 Pruning the Search Space

We now present some properties that are useful in defining heuristics for traversing the search space and approximating PDR (see proofs in Appendix A). For a configuration C and an index $IV \in closure(C)$, we define $base(IV)$ to be the set of original indexes in C which are part of a derivation that uses merging and reduction to produce IV.

PROPERTY 4.5. *Let C be a configuration,* IV_1 *and* IV_2 *be indexes in* closure(C)*, and* $IV_M = IV_1 \oplus IV_2$*. If* $IV_M \notin$ closure($C-$ base($IV1$))*, then* $PDR(C, B)$ *cannot include both* IV_1 *and* IV_M*.*

Property 4.5 shows that, if we merge two indexes IV_1 and IV_2, in some cases the optimal solution cannot contain both the merged index and any of its inputs. We next show that sometimes certain indexes cannot be part of the optimal solution.

PROPERTY 4.6. *Let C be a configuration,* IV_1 *and* IV_2 *be indexes in* closure(C)*, and* $IV_M = IV_1 \oplus IV_2$*. If* (i) size(IV_M) $>$ size(IV_1) + size(IV_2)*, and* (ii) *for each* $IV_k \in$ closure(C) *such that* $IV_M = IV_M \oplus IV_k$*, it still holds that* size(IV_M) $>$ size(IV_1) + size(IV_2) + *size*(IV_k)*, then* $IV_M \notin PDR(C, B)$*.*

Analogous properties for the reduction operator do exist and are shown below (we omit the proofs, however, since these are very similar to those of the properties above).

PROPERTY 4.7. *Let C be a configuration, IV be an index in* closure(C)*, and* $IV_R = \rho(IV, T, K)$ *for some tables T and columns K. If* $IV_R \notin$ closure ($C -$ *base*(IV))*, then* $PDR(C, B)$ *cannot include both IV and* IV_R*.*

PROPERTY 4.8. *Let C be a configuration, IV be an index in* closure(C)*, and* $IV_R = \rho(IV, T, K)$ *for some tables T and columns K. If* (i) size(IV_R) $>$ size(IV)*, and* (ii) *for each* $IV_k \in$ closure(C) *such that* $IV_R = IV_R \oplus IV_k$ *it still holds that* *size*(IV_R) $>$ *size*(IV_1) + *size*(IV_2) + *size*(IV_k)*, then* $IV_R \notin PDR(C, B)$*.*

In the next section we exploit the above properties to heuristically speed up our solution to the PDR problem.

4.4 A Heuristic Approach to Approximate PDR

In this section we introduce a heuristic approach to solve PDR that is inspired on the greedy solution to the fractional knapsack problem [Brassard and Bratley 1996]. In the fractional knapsack problem, we are given an integer capacity B and a set of objects o_i, each one with value a_i and volume b_i. The output is a set of fractions $0 \leq f_i \leq 1$ (one per object) such that the combined volume $\sum_i f_i b_i$ is no larger than B and the value $\sum_i f_i a_i$ is maximized (we can see the traditional "0/1" knapsack formulation as restricting the fractional knapsack so that each f_i is either zero or one). To solve the fractional knapsack problem, we first sort the input objects o_i in ascending order of the value-volume ratio a_i/b_i and then remove objects from this sequence until either the remaining objects fill completely the capacity B, or the last removed object o_k exceeds B.

In the latter case, we add back a fraction of o_k so that the total volume is exactly the input B capacity.[8] This assignment is optimal for the fractional knapsack problem. In the 0/1 case (i.e., no fractional objects are allowed), this heuristic performs very well in practice and a very simple refinement guarantees a factor-2 approximation to the optimal solution [Brassard and Bratley 1996].

Note that we can adapt the knapsack problem to our scenario. Our initial set consists of all the indexes in the closure of the original configuration. We define the volume of an index as the size it uses in the DBMS, and the value of an index as cost of the workload when the index is present minus the cost of the workload when the index is not present. In this case, a straightforward adaptation of the greedy solution described above would first generate the closure of the input configuration C, and then progressively remove from this configuration the index with the smallest "value-volume" ratio until the remaining ones satisfy the storage constraint. This approach has the following problems:

—The size of *closure*(C) can be in the worst case exponential in the number of original indexes. At the same time, intuitively the *best* views are either the original ones, or obtained via a short sequence of operations (recall that each operation degrades the performance of the workload). Most of the indexes in *closure*(C) are not present in the optimal configuration.

—The size (volume) of an index is not constant but depends on the configuration it belongs to. The reason is that we need to account for a primary index or heap associated with each different view definition. If many indexes share their view definition, we need a single primary index or heap for them.

—The impact (value) that each index has on the expected workload cost also depends on the configuration. We cannot assign a constant "value" to each index because of complex interactions inside the optimizer. An index that is not used to answer some query might become useful in conjunction with another index (e.g., for merge-join plans). Also, an index that is not very useful can become so if some other index is eliminated from the configuration.

—The greedy solution to the fractional knapsack problem does not exploit the domain-specific properties of Section 4.3 for pruning the search space.

To address these issues, we propose a *progressive* variation of the solution to the fractional knapsack problem. A simplified pseudocode is shown in Figure 4. Essentially, we start with the original configuration (line 1) and progressively refine it into new configurations that are smaller and slightly more expensive. While the current configuration *CF* is too large to fit in the available space (line 2), we identify a set of transformations to refine *CF* (lines 3–5) pick the most promising one (line 6), and apply it to *CF* (line 7) to obtain the next configuration. When we obtain a configuration that is within the storage constraint, we return it in line 8. One class of transformations (line 3) is the

[8] In reality, we sort objects in reverse order and keep a prefix of the sequence. This is equivalent to the solution described above, which leads more easily to our adaptation.

```
GreedyPDR (C:configuration, W:workload, B:storage bound)
01 CF = C
02 while (size(CF) > B)
03    TR = { delete IV for each IV ∈ CF }
04    TR = TR ∪ { IV1 ⊕ IV2 for each valid IV1, IV2 ∈ CF }
05    TR = TR ∪ { ρ(IV,T,K) for each valid T, K, and IV ∈ CF }
06    select transformation T ∈ TR with smallest penalty
07    CF = CF - "T's antecedents" ∪ "T's consequent"
          // Merge: antecedent are input views, consequent is merged view
          // Reduction: antecedent is input view, consequent is reduced view
          // Deletion: antecedent is view, consequent is empty
08 return CF
```

Fig. 4. Progressive knapsack for the physical design refinement problem.

same as in the greedy solution to the fractional knapsack problem (i.e., we remove indexes). However, the other two transformations (lines 4–5) explore the augmented search space (i.e., the closure of the original configuration) *on demand* by replacing one or two indexes with either a merged or reduced index.

In the remainder of this section we discuss some details of algorithm *GreedyPDR*:

—We consider the following transformations in lines 3–5: (i) deletion of each index in the current configuration *CF*, (ii) merging of each pair of compatible indexes in *CF*, and (iii) reductions of each index in *CF*. Specifically, for (iii) we consider reductions $\rho(IV, T, K)$ so that K are prefixes of the columns in the resulting index, and T are subsets of tables that match another view in *CF*.

—We use a heuristic derived from Properties 4.5 and 4.7 in line 4 and remove the input indexes whenever we introduce a transformed (merged or reduced) index in *CF*. Note that we do not check whether the transformed index can be generated by other derivations (see Properties 4.5 and 4.7) so there might be false negatives.

—We use a heuristic derived from Properties 4.6 and 4.8 in line 3 and not consider transformations (merges and reductions) whose result is $(1 + \alpha)$ times larger than the combined sizes of their inputs, for a small value of α. This heuristic avoids considering merged views with Cartesian products that originate from disjoint sets of join predicates. However, in general this heuristic might also result in false negatives.

—Rather than assigning a constant "value" and "volume" to each index, we use a dynamic approach that considers the interactions with the optimizer. For a given configuration C, we define the *penalty* of a transformation (i.e., deletion, merging, reduction) as $\Delta_{cost}/\Delta_{space}$, where Δ_{cost} is an estimate of the degradation in cost that we would expect if we applied the transformation, and Δ_{space} is the amount of space that we would save by applying the transformation. Penalty values are then a measure of units of time that we lose per unit of space that we gain for a given transformation. We obtain Δ_{space} and Δ_{time}

values as in Bruno and Chaudhuri [2005], and use penalties as the dynamic version of the value-volume ratio in the original knapsack formulation.

—To avoid incremental errors in estimation, we reoptimize the inferred workload under the new configuration *CF* after each transformation. We minimize optimization calls by only reoptimizing the queries in the workload that used an index that got removed from *CF*.[9] The rationale is that we keep replacing indexes with coarser alternatives, so any query that did not use, say, IV_1 in a given configuration, should not use $IV_1 \oplus IV_2$ or $\rho(IV_1, T, K)$ if they became additionally available. This heuristic saves significant time and almost never degrades the quality of the final configurations.

4.4.1 *A Note on Update Queries.* So far we implicitly focused on workloads composed entirely of SELECT queries. In reality, most workloads consist of a mixture of "select" and "update" queries. The main impact of an update query is that indexes defined over the updated table (or dependent views) might also be updated as a side effect. Similarly to Bruno and Chaudhuri [2005], we *conceptually* separate each update query into two components: a pure select query (which we process as before), and a small update shell. For instance, the query

```
UPDATE T
SET a=b+1, c=2*c
WHERE a<10 AND d<20
```

is seen as a pure select query (S) and an update shell (U):

```
S =  SELECT b+1, 2*c             U =  UPDATE TOP(k) T
     FROM T                           SET a=a, c=c
     WHERE a<10 and d<20
```

where k is the estimated cardinality of the select query (S). In the presence of updates, penalty values $\Delta_{cost}/\Delta_{space}$ for certain transformations might be negative (e.g., a transformation that removes an index I can actually decrease the total execution time if I has a large update overhead and relatively smaller benefits for query processing). For that reason, when updates are present we can transform a configuration into another that is both smaller and more efficient. Thus, we should not exit the loop in lines 2–7 after a configuration fits in the available storage since a later configuration might be even more efficient while still satisfying the storage constraint. To handle these scenarios, we relax the condition in line 2 of Figure 4 as follows:

```
02  while (size(CF) > B) or (last_transformation_penalty < 0)
```

To simplify the presentation of the algorithms in the next section, we keep assuming select-only workloads, understanding that the concepts in this section are still applicable in the resulting algorithms.

[9]We handle triggers by considering the cascading queries along with the triggering query in a single atomic block, and therefore consider all the relevant indexes altogether.

4.5 Discussion: Why Merge and Reduce?

The closure of an input configuration under the merge and reduction operators induces a restricted search space of configurations. We now explain why the merge and reduction operators in fact cover the set of relevant indexes over views for the physical design problem described in this paper, in the context of typical query optimizers.

Consider a subquery q_1 that exactly matches a view V_1 (i.e., q_1 and V_1 are semantically equivalent). Then the whole q_1 can be matched and answered by either V_1 or some generalization V' of V_1 (e.g., V' is obtained by adding to V_1 additional group-by columns or relaxing its selection predicates). By definition, V' is larger than V_1 and therefore less effective in answering subquery q_1. Why should we then consider V' in our search space, since it is both larger and less efficient for q_1 than the original V_1? The only reasonable answer is that q_1 is not the only query in the workload. Instead, there might be some other subquery q_2 (which is matched perfectly by V_2), for which V' can also be used. In this scenario, having V' instead of the original V_1 and V_2 might be beneficial, since the space for V' might be smaller than the combined sizes of V_1 and V_2 (albeit being less efficient for answering q_1 and q_2). We should then consider V' in our search space, noting that V' must be a generalization of both V_1 and V_2. Now, the merging of $V_1 \oplus V_2$ seems the most appropriate choice for V', since it results in the most specific view that generalizes both V_1 and V_2 (other generalizations are both larger and less efficient than $V_1 \oplus V_2$). We conclude that the merge operation covers all the “interesting” views that can be used to answer query expressions originally matched by the input set of views.

Let us now consider subexpressions. In fact, a view V_R which is not a generalization of a query q can still be used to rewrite and answer q (see Section 3). It would also make sense, then, to consider in our search space such subexpressions of q that can be used to speed up its processing. In general, the “reductions” that we look for are somewhat dependent on the view matching engine itself. View matching engines typically restrict the space of transformations and matching rules for efficiency purposes. Specifically, usually the only subqueries that can be matched and compensated with a restricted view contain fewer joins. But this is how the reduction operator is defined (i.e., eliminating joins from the original view). However, in principle, if the view matching engine is capable of “unioning” several horizontal fragments of a single template expression to answer a given subquery, we should certainly consider range partitioning over a column as a potential primitive operator in contrast to the definition $\rho(IV, T', K')$ as defined in Section 3.2. Thus, the reduction operator in its generality (as in Definition 3.1) covers all the “interesting” views that can be used to rewrite queries originally matched by the input set of views (we note that any generalization of a reduced view can also be used but this is covered, again, by the merge operator).

Putting all together, we believe that the merge and reduction operators are the primitive building blocks capable of generating a wide class interesting views to consider (as the expressive power of typical view matching engines

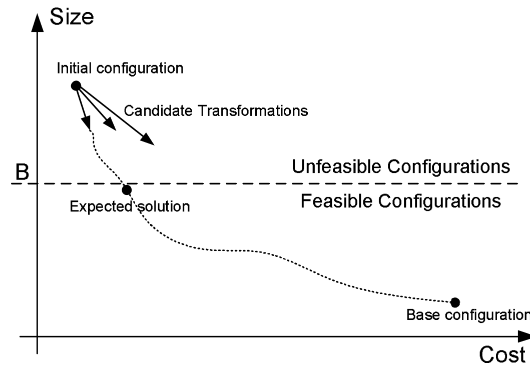


Fig. 5. Geometric interpretation of the PDR problem.

expands, it would translate into more richer versions of the merge and reduction operations).

5. VARIATIONS OF THE ORIGINAL PDR PROBLEM

In this section we introduce important variations and extensions to the original PDR problem discussed in the previous section. We start by discussing a geometric interpretation of the PDR problem and the greedy solution of Section 4.4. Figure 5 shows a two-dimensional scatter plot to illustrate the main ideas. Each point in the plot corresponds to a configuration. The y-axis represents the size of the configuration and the x-axis represents the expected cost for evaluating the input workload under the given configuration. We can see that the initial configuration has a large size but small cost. Also, the base configuration (that is, the configuration that contains only mandatory indexes) has the smallest possible size, but a high cost due to lack of additional indexes. The dotted horizontal line that crosses B delimits the feasibility region for PDR (any configuration above that line is too large to be valid). We can see that the initial configuration is unfeasible (otherwise the problem is trivial) and the base configuration is feasible (otherwise the problem has no solution). The goal of PDR is to find a configuration in the feasible region that is as efficient as possible (i.e., toward the left of the plot). Each transformation (i.e., deletion, merging, or reduction operations) transforms a given configuration into another one that is smaller but less efficient (i.e., moves the configuration in the down-right direction). The figure graphically shows three candidate transformations and their expected effect on the initial configuration. The heuristic used in Section 4.4 chooses the transformation that minimizes the value $\Delta_{cost}/\Delta_{space}$. Graphically, this transformation corresponds to the one with the angle closest to the vertical line (i.e., the one that descends the steepest towards the feasible region).

This geometric interpretation is useful in understanding some of the extensions discussed next. Specifically, in Section 5.1 we study a more comprehensive way to explore the search space. in Section 5.2 we introduce the dual of the basic PDR problem. Finally, in Section 5.3 we present a constrained version of

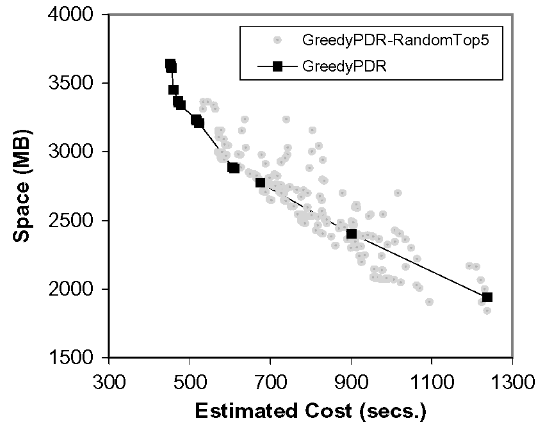


Fig. 6. GreedyPDR does not discover all the configuration in the cost/space skyline.

the PDR problem that disallows configurations that are too different from the original one.

5.1 Adding Backtracking to GreedyPDR

Looking at Figure 5 we observe that our objective is to proceed along the steepest descent until we get into the feasible region. The main difficulty is that, at each iteration, we do not know the true value of $\Delta_{cost}/\Delta_{space}$, but we have just an estimate. For that reason, we are bound to making some mistakes and choosing the wrong transformation (due to the greedy nature of our algorithm, we even risk getting "cornered" in a bad subspace of solutions). Moreover, it is not guaranteed that a single sequence of transformations results in all the configurations in the "cost/space skyline," because these might only be obtained from taking different paths. To illustrate this point, we first ran *GreedyPDR* using a TPC-H database and workload (see Section 7 for more details on the experimental setting) and obtained the configurations that are connected by lines in Figure 6. Then we ran *GreedyPDR* multiple times, but at each iteration we chose, instead of lowest-penalty transformation, a random transformation among the top five. The figure shows all the configurations that we discovered in this way, and illustrates that the configurations in the skyline are not necessarily the ones obtained by *GreedyPDR*.

To mitigate this problem, we propose a variation of the *GreedyPDR* algorithm that more comprehensively searches the space of configurations. Specifically, we allow multiple invocations of the original *GreedyPDR* solution starting from different initial points and using different transformations. Figure 8 shows a pseudocode of *GreedyPDR-BT* that implements this idea (we note that other backtracking choices are also possible). Lines 4–9 are almost the same as in the original *GreedyPDR*. A key difference is that in line 8 we select the best *unused* transformation for the current configuration (because the same configuration might be considered multiple times). After each iteration of lines 4–9 in *GreedyPDR-BT*, we obtain a new feasible configuration, and in line 10 we maintain the best overall configuration across iterations. Line 3 selects the

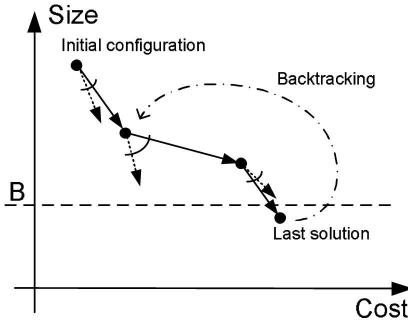


Fig. 7. Backtracking to the configuration with the largest estimated penalty error.

```
GreedyPDR-BT (C:configuration, W:workload, B:storage bound)
01 CBest = CBase   // CBase is the base configuration
02 while (not timeOut())
03    CF = (CBest=CBase) ?  C : pickConfiguration(CBest)
04    while (size(CF) > B)
05       TR = { delete IV for each IV ∈ CF }
06       TR = TR ∪ { IV1 ⊕ IV2 for each valid IV1, IV2 ∈ CF }
07       TR = TR ∪ { ρ(IV,T,K) for each valid T, K, and IV ∈ CF }
08       select unused transformation T ∈ TR with smallest penalty
09       CF = CF - "T's antecedents" ∪ "T's consequent"
10    if (cost(CF,W)<cost(CBest,W)) CBest=CF
11 return CBest

pickConfiguration (C:configuration)
01 CS = { CP : CP is an ancestor of C }
      // CP is ancestor of C if C was obtained from CP using transformations
03 return C ∈ CS with the maximum value of score(C), where
      score(C) = "expected-penalty - actual-penalty" for C
```

Fig. 8. Backtracking in the configuration space to obtain better solutions.

initial configuration for each iteration (i.e., implements the backtracking mechanism). The first time, we start with the input initial configuration as in the original *GreedyPDR*. In subsequent iterations, we consider as candidate starting points the set of ancestors of the best configuration found so far (i.e., the configurations that were iteratively transformed from the initial to the current best configuration). Among these, we pick the one that resulted in the largest error when estimating the penalty value (see Figure 7, where dotted arrows represent the estimated penalty values and plain arrows the actual ones). The rationale is that by restarting from such configuration and choosing some other transformation, we might be able to "correct" previous mistakes. The function *pickConfiguration* in Figure 8 selects the initial configuration for each new iteration of the main algorithm. When calling *pickConfiguration*, we already evaluated all the ancestors of the current best configuration, and therefore we have the actual values of Δ_{cost} and Δ_{space}, which we use to obtain the "actual" penalty value and thus the error in its estimation. Finally, line 2 in *GreedyPDR-BT* controls the overall time we spend in the algorithm. The *timeOut* function

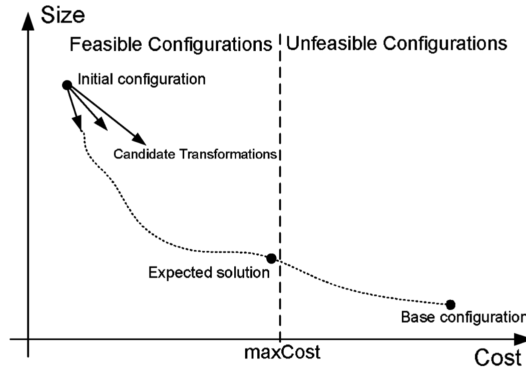


Fig. 9. Geometric interpretation of the Dual-PDR problem.

can be implemented in different ways. For instance, we can give a wall-clock limit to *GreedyPDR-BT*, or bound the number of iterations of the inner loop of lines 4–9.

5.2 Dual-PDR Problem

We now introduce the Dual-PDR problem, which is an interesting variation on physical design refinement. Rather than putting a constraint on the size of the resulting configuration, we require a configuration whose cost is no worse than a certain percentage of that of the current configuration, and has the minimum possible size. When DBAs are interested in removing redundancy from manually tuned configurations, the Dual-PDR problem is an attractive approach. We next define the Dual-PDR problem formally.

Definition 5.1 (*Dual Physical Design Refinement* (*Dual-PDR*) *Problem*). Given a configuration $C = \{I_1 \mid V_1, \ldots, I_n \mid V_n\}$, a representative workload W, and a cost constraint $maxCost$, we define Dual-PDR($C, W, maxCost$) as the refined configuration C' such that

(1) $C' \subseteq closure(C)$;
(2) $\sum_{q_i \in W}(\alpha_i \cdot \ cost(q_i, C')) \leq maxCost$;
(3) size(C′) is minimized.

Figure 9 shows the graphical interpretation of the Dual-PDR problem. In the Dual-PDR problem, the feasibility region is a vertical line that crosses *maxCost*. In contrast to PDR, the initial configuration is feasible (otherwise the problem has no solution) and generally the base configuration is unfeasible (otherwise the problem is trivial). To address this new scenario, we *stratify* the penalty function so that all transformations that are expected to result in configurations in the unfeasible region (cost-wise) are *ranked lower* than the transformations that result in “valid” configurations. Nonetheless, the refinement heuristic used for PDR is still very much applicable for the Dual-PDR problem and we can reuse it. The remaining difference in the Dual-PDR problem is the stopping criterion and the fact that after stopping we return the previous configuration

```
GreedyDualPDR (C:configuration, W:workload, MC:cost bound)
01 CF = C; CP = NULL
02 while (cost(CF) ≤ MC)
03    CP = CF
04    TR = { delete IV for each IV ∈ CF }
05    TR = TR ∪ { IV1 ⊕ IV2 for each valid IV1, IV2 ∈ CF }
06    TR = TR ∪ { ρ(IV,T,K) for each valid T, K, and IV ∈ CF }
07    select transformation T ∈ TR with smallest penalty
08    CF = CF - "T's antecedents" ∪ "T's consequent"
09 return CP
```

Fig. 10. A solution for the dual physical design refinement problem.

(since the current one is by definition unfeasible). Figure 10 shows a pseudocode for the Dual-PDR problem slightly adapted from the greedy solution of Figure 4.

We can add backtracking capabilities to *GreedyDualPDR* analogously to what we do in Figure 8 for *GreedyPDR*. To keep the presentation short, however, we omit such details.

5.3 CPDR: Constraining the Number of Transformations

Physical design refinement restricts the exploration of physical structures in the closure of the original configuration, and thus ensures only incremental changes to the original design. Therefore, we can easily explain how we obtained the resulting refined configuration by detailing the sequence of transformations required to arrive to the final configuration. So far, this “small number of changes” objective was implicit in the formulation of our problem, but we had no way of bounding how different the final configuration could be from the original one. In this section, we generalize the PDR problem so that it can also take constraints on the number of allowed transformations[10] (deletion, merging, and reduction) to obtain the final configuration from the initial one. We next formally define the constrained PDR problem (or CPDR).

Definition 5.2 (*Constrained-PDR* (*CPDR*) *Problem*). Given a configuration $C = \{I_1 \mid V_1, \ldots, I_n \mid V_n\}$, a representative workload W, a storage constraint B, and an integer bound MT, we define CPDR(C, W, B, MT) as the refined configuration C' such that

(1) $C' \subseteq closure(C)$;
(2) $size(C') \leq B$;
(3) $C' = t_1(t_2(\ldots(t_k(C))\ldots))$, where t_i are valid transformations and $k \leq MT$;
(4) $\sum_{q_i \in W}(\alpha_i \cdot cost(q_i, C'))$ is minimized.

In other words, CPDR extends the classic PDR problem by imposing an additional constraint on the number of changes from the original configuration that are allowed in the result. By introducing this additional constraint, we significantly change the space of solutions. In particular, some CPDR problems are overconstrained and thus unfeasible, which did not happen for the original

[10]Note that we constrain the number of *applications* of transformations, and not the *types* of transformations themselves.

```
GreedyCPDR (C:configuration, W:workload, B:storage bound,
            MT:transformation bound)
01 CF = C
02 while (size(CF) > B and ancestorCount(CF) < MT)
03    TR = { delete IV for each IV ∈ CF }
04    TR = TR ∪ { IV1 ⊕ IV2 for each valid IV1, IV2 ∈ CF }
05    TR = TR ∪ { ρ(IV,T,K) for each valid T, K, and IV ∈ CF }
06    select transformation T ∈ TR with smallest penalty
07    CF = CF - "T's antecedents" ∪ "T's consequent"
08 if (size(CF) > B) CF=NULL   // No solution found
09 return CF
```

Fig. 11. Constrained physical design problem.

PDR or Dual-PDR problems (recall that either the base configuration or the initial one were always solutions to the corresponding problems except for the trivial exceptions). Additionally, the new constraint on the number of allowed transformations makes the exploration of the search space more difficult. The reason is that, the smaller the number of allowed transformations, the more difficult it is to reach a configuration in the feasible region.

Figure 11 shows a straightforward mechanism to extend the *GreedyPDR* algorithm of Section 5.1. The changes are (i) the additional condition in line 2 to process only feasible configurations, `ancestorCount(CF)<MT` (where `ancestorCount(C)` is the number of transformations from the initial one to C), and (ii) the final checking in line 8 to avoid returning an unfeasible configuration.

These changes ensure that we only consider feasible configurations, and therefore the resulting algorithm is sound. Unfortunately, in the presence of both space constraints and a small number of transformations, in many situations the algorithm might fail to produce any feasible solution (specially for small values of MT). Adding backtracking as we did for the previous algorithms helps ameliorating this issue, but the search is still ineffective. To address this problem, we need to modify our ranking of candidate transformations, which focus the search strategy toward feasible solutions in the constrained search space. Specifically, we *stratify* the ranking function as discussed bellow.

5.3.1 *Stratification.* The original penalty function only uses the ratio $\Delta_{cost}/\Delta_{space}$ without considering the absolute length of the "step." When the number of allowed transformations is small, a transformation with a great penalty value that minimally decreases the configuration size is not very useful (in the extreme, if there is only one remaining transformation, the only transformations that are useful are those that reduce the configuration size below the storage bound B). To handle this issue, we stratify the ranking function. Specifically, we identify a subset of transformations as *useful*, and we rank all useful transformations ahead of the remaining ones (we still consider the remaining transformations because, after all, penalty values are just approximations and we do not want to erroneously prune the search space). Consider a configuration C with cost c and space b that was obtained after applying t transformations to the original configuration. In this case, we still need to diminish the size of C by at least $(B - b)$ using at most $(MT - t)$ transformations. In that case, we proceed as follows:

(1) Sort the candidate transformations for C in descending order of Δ_{space}.

(2) Calculate $maxRest = \sum_{i=1}^{MT-t-1} \Delta_{space}$ (i.e., we sum the top $MT - t - 1$ transformations sorted by Δ_{space}). This is an indication on the maximum amount of space that we would be able to reduce after applying the current transformation. Note that this is only an indication because in future configurations the transformations might change due to interactions among transformations.

(3) Label each transformation $t \in TR$ as *useful* if its value $\Delta_{space} \geq B - maxRest$.

In other words, we consider first all transformations that, at least in an estimated sense, have the opportunity of transitioning the configuration in the feasible region within the bounded number of transformations.

5.3.2 *The Dual-CPDR Problem.* Similarly to Section 5.2, we can define the dual of the constrained PDR problem, or Dual-CPDR. In the Dual-CPDR problem, we try to minimize the size of the resulting configuration while not exceeding a certain cost and a given number of transformations. When adapting the penalty function to this new scenario, a complication—analogous to that of the CPDR problem—arises. In this situation, however, we risk obtaining suboptimal configurations when we bound the number of allowed transformations. In fact, since we already start in the feasible region (see Figure 9), we might not be aggressive enough in choosing transformations when we get closer to the last alternatives. As an example, suppose that we have a single remaining transformation to apply. In this case we should not choose the one that minimizes $\Delta_{cost}/\Delta_{space}$, but the one that simply maximizes Δ_{space} (i.e., minimizes $1/\Delta_{space}$) among the feasible ones (recall that in the Dual-CPDR problem we try to minimize the configuration size). In other words, when $(MT - t)$ is still large, we would like to proceed as in the original solution. However, as we get fewer and fewer remaining transformations to apply, we would like to give less relative weight to Δ_{cost} values. For this purpose, we *bias* the definition of penalty values for transformations as follows. Consider a configuration C with cost c and space b that was obtained after applying t transformations to the original one. In that case, we still need to minimize the size of the C without incurring $(MC - c)$ additional cost and using at most $(MT - t)$ additional transformations. We define the penalty of a transformation as

$$\frac{(\Delta_{cost})^D}{\Delta_{space}}, \qquad \text{where } D = \frac{MT - t - 1}{MT}.$$

Thus, initially when $t << MT$, D is close to one and we behave as in the original Dual-PDR problem. As the number of remaining transformations decreases (i.e., t tends to MT), the value of D tends to zero and we smoothly transition to the alternative $1/\Delta_{space}$.

6. PHYSICAL DESIGN SCHEDULING

At the end of a physical design refinement session, DBAs are required to implement and deploy the recommended configuration. Of course, this problem is

no different from the analogous case after a regular tuning session takes place. Surprisingly, this problem has not been addressed before in the literature. In this section we formalize this task and show that it can be cast as a search problem (our goal is not to provide a full treatment of the problem, since that would probably require a separate article).

Let C_0 be the current configuration and C_f be the desired configuration to deploy (C_f might have been obtained after a regular tuning session, or perhaps as the answer to a PDR problem). A physical schedule consists of a sequence of index-creation and index-drop statements that starts in configuration C_0 and ends in configuration C_f. The physical design scheduling problem consists of finding the minimum cost schedule using no more than B storage, where $B \geq \max\left(size(C_f), size(C_0)\right)$ (we might need more than $size(C_f)$ space to accommodate intermediate results while creating indexes in C_f). As an example, consider the following configurations over a database with a single table R:

$$C_0 = \{(c, d), (e)\},$$
$$C_f = \{(a, b), (a, c), (c), (e)\}.$$

A naive schedule would first remove all indexes in $(C_0 - C_f)$ and then create the indexes in $(C_f - C_0)$, that is:

$$[\texttt{drop}(c, d),\ \texttt{create}(a, b),\ \texttt{create}(a, c),\ \texttt{create}(c)]$$

To create an index, say, (c), we require to sort a vertical fragment of the table (or view) on column c. It is important to note that we can speed up the creation of (c) by using existing indexes. Specifically, if an index with leading column c already exists in the database, we can build (c) without sorting, by just scanning such an index. Therefore, the following schedule could be better than the original one (assuming that the indexes at each intermediate step fit in the storage constraint):

$$[\texttt{create}(c),\ \texttt{drop}(c, d),\ \texttt{create}(a, b),\ \texttt{create}(a, c)]$$

In this case, we use (c, d) to avoid sorting table R in c order to create (c), thus saving time. However, to realize the above saving we need to be able to store (c) and (c, d) simultaneously. In fact, there might be more efficient schedules which create *additional* intermediate structures outside $(C_f - C_0)$. Consider the following schedule:

$$[\texttt{create}(c),\ \texttt{drop}(c, d),\ \texttt{create}(a, b, c),\ \texttt{create}(a, b),\ \texttt{create}(a, c),\ \texttt{drop}(a, b, c)]$$

In this situation, we create a temporary index $(a, b, c) = (a, b) \oplus (a, c)$ before creating (a, b) and (a, c). Therefore, we need to sort (a, b, c) only once, and then (a, b) and (a, c) can be built with only a minor-sort on the secondary columns, which is much more efficient than the full alternative (e.g., if a is a key, no sorting is required for (a, b) and (a, c)). This schedule might be more efficient than the previous one, but at the same time requires additional storage for intermediate results. The general problem can be defined as follows.

Definition 6.1 (*Physical Design Scheduling* (*PDS*) *Problem*). Given configurations C_0 and C_f and a space constraint B, obtain a physical schedule $PDS(C_0, C_f, B) = (s_1, s_2, \ldots, s_n)$ that transforms C_0 into C_f such that

(1) each s_i drops an existing index or creates a new index in $closure(C_f)$;
(2) the size of each intermediate configuration plus the required temporary space by the corresponding s_i is bounded by B;
(3) the cost of implementing $(s_1, s_2, \ldots, s_n)$ is minimized.

The two main challenges of the PDS problem are (i) an explosion in the search space due to the ability to add elements in the closure of C_f, and (ii) the space constraint, which invalidates obvious approaches based on topological orders. Below we introduce a property that connects the PDS problem with a shortest-path algorithm in an induced graph.

PROPERTY 6.2. *Consider an instance of the physical design scheduling problem $PDS(C_0, C_f, B)$, and let $G = (V, E)$ be an induced graph defined as follows:*

—$V = \{v | v \in (C_0 \cup \text{ closure}(C_f)) \wedge size(v) \leq B\}$.
—*There is a directed edge $e = (v_1, v_2)$ in E if the symmetric difference between v_1 and v_2 has a single element (i.e., $|(v_1 - v_2) \cup (v_2 - v_1)| = 1$). The weight of e is equal to the cost of creating the index in $v_2 - v_1$ starting in configuration v_1 (if $v_1 \subset v_2$), or the cost of dropping the index in $v_1 - v_2$ (if $v_2 \subset v_1$). The label of edge e is the corresponding* create *or* drop *action.*

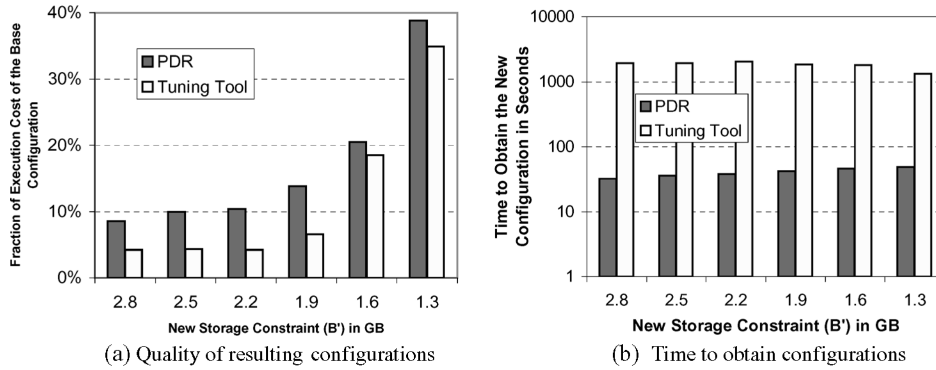
In that case, the solution of $PDS(C_0, C_f, B)$ is the sequence of labels of the shortest path between C_0 and C_f in the induced graph as defined above.

While this property does not directly lead to an efficient algorithm (i.e., the induced graph has an exponential number of nodes in the worst case), it can be used as a starting point to define search strategies. For instance, we could use an $A^\star$[Nilsson 1971] algorithm that progressively explores the search space by generating the induced graph on demand. Details of such strategies, however, are outside the scope of this work.

7. EXPERIMENTAL EVALUATION

In this section we report experimental results on an evaluation of the techniques introduced in this work.

In regard to the *Experimental setting*, we implemented the various PDR algorithms of Sections 4 and 5 as a client application in C++ and used Microsoft SQL Server 2005 as the DBMS. In our experiments we used a TPC-H database and workloads generated using the `dbgen` utility (`http://www.tpc.org/tpch/default.asp`). In Section 7.1 we evaluate the original PDR algorithm of Section 4. Then, in Section 7.2 we analyze the impact of the backtracking extensions of Section 5.1. Finally, in Section 7.3 we report an evaluation of the constrained PDR problem as defined in Section 5.3. To keep the presentation short, we only report results on the original PDR problem and omit those for the Dual-PDR problem, which are similar.



(a) Quality of resulting configurations (b) Time to obtain configurations

Fig. 12. Refining configurations versus producing new configurations from scratch.

7.1 Original PDR Problem

The goal in this section is to compare the PDR algorithm of Section 4 against state-of-the-art physical design tools [Agrawal et al. 2004] regarding the quality of refined configurations and the time it takes to obtain them. For this purpose, we proceeded as follows. First, we took a workload W and tuned it with a physical database design tool for B maximum storage, obtaining a configuration C_B^{Tool}. Second, we refined C_B^{Tool} using our PDR implementation with a stricter storage constraint of $B' < B$ and the inferred *inferredW* workload, obtaining configuration $C_{B'}^{PDR}$. Third, we retuned W from scratch using B' as the new storage constraint, obtaining configuration $C_{B'}^{Tool}$. Finally, we evaluated the cost of the original workload W under both $C_{B'}^{PDR}$ and $C_{B'}^{Tool}$, and also the time it took to produce each alternative configuration.

Figure 12 shows the results on a 1GB TPC-H database with an initial storage constraint of $B = 3$ GB and a 20-query TPC-H workload W (we obtained the 20-query workload by running the dbgen tool and removing the last two queries from its output). We then used several values of B' ranging from 2.8 GB (very little refinement) down to 1.3 GB (very aggressive refinement). In the figure we measure the cost of W under a given configuration as a fraction of its cost under the base configuration that only contains primary indexes. We see in Figure 12(a) that in all cases, the refined configuration obtained by PDR is only of slightly less quality than the alternative obtained from scratch with the tuning tool. In fact, the cost difference for the original workload between both configurations is below 10% in all cases. Additionally, Figure 12(b) shows that the time it takes to refine a configuration can be orders of magnitude smaller than that to produce a new configuration from scratch (note the logarithmic scale in Figure 12(b)).

7.1.1 *Analyzing Configurations.* We next take a closer look at the resulting configurations from both PDR and the tuning tool. For that purpose, we took the 20-query workload defined before and tuned the TPC-H database with the tuning tool so that it recommends indexes over base tables fitting in 3.1 GB (we denote such configuration $C_{3.1GB}^{Tool}$). We then ran PDR with a space bound of 2.8 GB, obtaining $C_{2.8GB}^{PDR}$, and rean the tuning tool also with

Table I. Fraction of Cost After Refining a Configuration Versus Creating a New One from Scratch

Configuration	Fraction of cost of base conf	Time to obtain configuration
$C^{Tool}_{3.1GB}$	23.74%	843 s
$C^{Tool}_{2.8GB}$	26.04%	809 s
$C^{PDR}_{2.8GB}$	27.25%	22 s

a space bound of 2.8 GB, obtaining $C^{Tool}_{2.8GB}$. Table I summarizes the cost of the original workload in the three configurations as well as the time it took to obtain such configurations. While both configurations at 2.8 GB are less effective than the original one at 3.1 GB, running an automatic tool from scratch results in a slight improvement compared to using PDR (confirming the results in the previous section for the case of configurations with index over base tables only). However, the difference is not that large, and the refined configuration has additional benefits. First, it took only 22 s to obtain the refined configuration against over 800 s for running the tuning tool again from scratch. Also, it is rather difficult to "understand" what changed from $C^{Tool}_{3.1GB}$ to $C^{Tool}_{2.8GB}$ short of doing a manual analysis of both sets of indexes (and even in this case we find several indexes in $C^{Tool}_{2.8GB}$ that have no obvious relationship with those in $C^{Tool}_{3.1GB}$). In contrast, we can easily explain what changed while doing the refinement, and which queries are being affected by which changes. Specifically, we steps in PDR were: (1) Delete `PART(size, partkey, mfgr, type)`, (2) Delete `ORDERS(custkey)`, (3) Merge `LINEITEM(orderkey, partkey, suppkey, quantity, extendedprice, discount, returnflag)` and `LINEITEM(partkey, orderkey, linenumber, quantity, extendedprice, suppkey, discount)`, (4) Delete `PART(brand, container, size, partkey)`, and (5) Merge `LINEITEM(shipdate, suppkey, extendedprice, orderkey, discount, partkey, quantity, commitdate, receiptdate)` and `LINEITEM(shipdate, discount, quantity, extendedprice, suppkey, partkey)`. This is better appreciated while comparing the execution plans of the queries in the workload for the different configurations. As an example, consider the first query in the workload:

```
SELECT returnflag, linestatus, SUM(quantity), "other aggregates"
FROM LINEITEM
WHERE shipdate <= '1998/6/03'
GROUP BY returnflag, linestatus
ORDER BY returnflag, linestatus
```

Figure 13 shows the execution plans under the three configurations under consideration. We can see that for $C^{Tool}_{3.1GB}$ (see Figure 13(a)), we first seek on a covering index for the tuples satisfying the condition on `shipdate`, and then do a group-by plus aggregation using a hash-based algorithm. Finally, we sort the intermediate results (since the hash-based alternative does not necessarily output tuples in the right order). When optimizing under $C^{PDR}_{2.8GB}$ (see Figure 13(b)), we see that the plan is almost the same, with the covering index replaced by the merged alternative described above. The resulting plan, while less efficient, is very similar to the original plan. In contrast, when optimizing under $C^{Tool}_{2.8GB}$

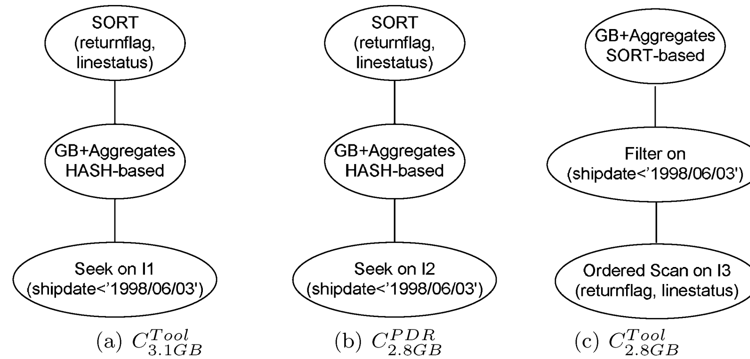


Fig. 13. Plans produced for the first query in the workload for different configurations.

(see Figure 13(c)), the resulting plan scans a covering index with `(returnflag, linestatus)` key and filters on the fly the tuples according to the `shipdate` predicate. Finally, the resulting tuples are grouped and aggregated using a sort-based alternative, since they are already in the right order.

In conclusion, while $C^{Tool}_{2.8GB}$ results in slightly better performance than $C^{PDR}_{2.8GB}$ for the input workload, it usually takes much longer to produce and results in execution plans that are very different from the original ones, which might not be desirable to DBAs.

7.1.2 *Varying Workloads.* In the previous experiments we assumed that the workload W used to initially tune the database did not change, and therefore we used the same workload W to evaluate the resulting refined configurations. In real scenarios, however, workloads tend to drift, if not in the actual queries themselves, at least in their frequency distribution. To evaluate such scenarios, we conducted the following experiment. Initially, as before, we took the same workload W and tuned it with a design tool for $B = 3$ GB storage, obtaining the initial configuration. Second, we evaluated a slightly different workload W' in the DBMS. The new workload W' was generated as follows: (i) we changed the frequency distribution of queries from uniform to Zipfian ($z = 0.5$), (ii) we removed the two queries with the smallest frequency from W', and (iii) we added the two queries produced by `dbgen` that we initially excluded from W. After evaluating W', we refined the current configuration using the three workload generating alternatives of Section 4.2 and stricter storage constraints B' between 1.5 GB and 2.5 GB. As before, we also re-tuned W' from scratch using the new storage constraint, and evaluated the new W' under all the resulting configurations.

We can see in Figure 14 that the inferred workload *inferredW* performs worse than before, since the information it exploits is not up-to-date anymore. However, we note that the resulting configurations are still considerably better than the base configuration. When using the profiled workload *profiledW* the results improve, because we can extract additional information based on the execution of the new workload W' and thus assign more representative weights to the queries in the generated workload *profiledW*. Finally, using the fully logged workload *loggedW* $= W'$ is the best alternative, and the results are similar to

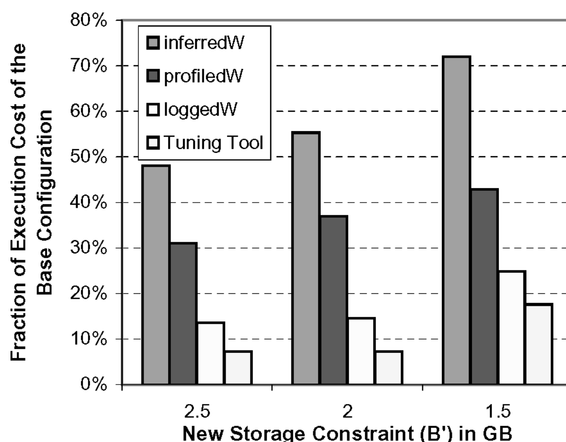
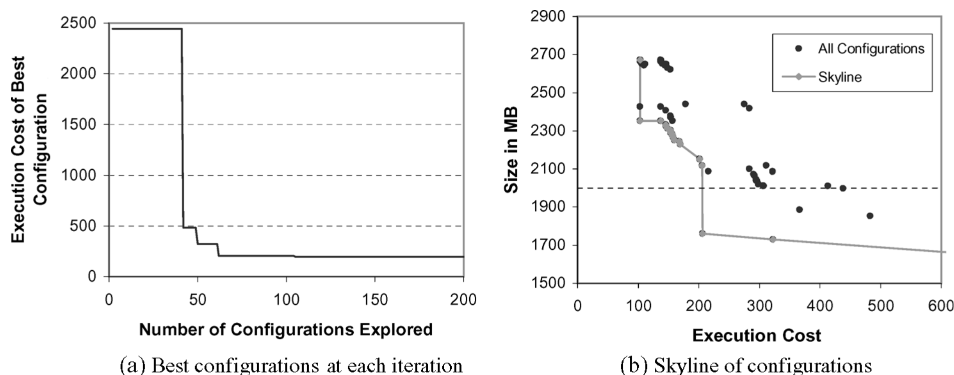


Fig. 14. Refining configurations when the underlying workload drifts.



(a) Best configurations at each iteration (b) Skyline of configurations

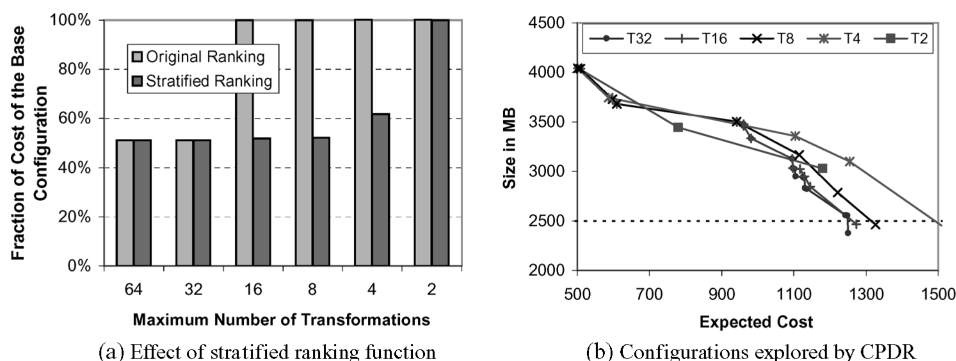
Fig. 15. Exploiting backtracking to obtain better configurations with *GreedyPDR-BT*.

those of Figure 12. The gap between using PDR with the original workload W' and tuning the DBMS from scratch are due to two reasons. First, although *loggedW* has access to the original workload W', the refinement starts from the original configuration that did not have any physical structures tuned specifically for queries in $(W' - W)$. Second, the quick refinement of PDR is always suboptimal compared to the full tuning of the physical design tool.

7.2 Effect of Backtracking

In this section we evaluate the effect of backtracking on the quality of the resulting configurations. We generated a new 22-query TPC-H workload with `dbgen` and tuned the DBMS for optimal performance using a physical design tool (we obtained a 2.7-GB configuration). We then initiated a PDR session (using *GreedyPDR-BT*) with a storage constraint of 2 GB and a time limit of 5 min.

Figure 15(a) shows the expected cost of the current best configuration after each new configuration is evaluated by *GreedyPDR-BT*. Before 1 min, after evaluating close to 50 configurations, *GreedyPDR-BT* found the first solution,



(a) Effect of stratified ranking function

(b) Configurations explored by CPDR

Fig. 16. Evaluating the constrained physical design problem CPDR.

with an expected cost of 500 units (this would have been the final solution of the original algorithm *GreedyPDR*). After that, *GreedyPDR-BT* started the backtracking process and the best configuration kept improving down to 205 units at the end. Figure 15(b) shows each evaluated configuration and the cost/space skyline. We note that all the configurations in the skyline do not belong to the same refinement sequence, but are instead taken from different iterations of the inner loop of *GreedyPDR-BT*.

7.3 Constrained PDR Problem

In this section we evaluate the constrained PDR (CPDR) problem defined in Section 5.3. For that purpose, we generated an optimal 4-GB configuration using a physical tool for a 44-query workload generated by concatenating two different executions of `dbgen`. We then generated CPDR instances with a storage constraint of 2.5 GB and different bounds on the number of allowed transformations.

Figure 16(a) shows the fraction of cost of each resulting configuration with respect to the base one, both when using the original penalty function and the modification of Section 5.3.1. We see that, the larger the number of allowed transformations $maxT$, the better the resulting configurations. For $maxT \leq 16$, the CPDR algorithm with the original penalty function failed to obtain any solution (we then show the corresponding bar at 100%). In contrast, by using the modified penalty function, algorithm *GreedyCPDR* was able to find feasible solutions even for $maxT = 4$. The reason is that *GreedyCPDR* effectively biases the search strategy toward feasible configurations in the constrained space. To complement the analysis, Figure 16(b) shows a typical sequence of configurations explored during the execution of *GreedyCPDR* for varying values of $maxT$.

8. RELATED WORK

In recent years there has been considerable research on automated physical design in DBMSs. Several pieces of work (e.g., Agrawal et al. [2000, 2006]; Chaudhuri and Narasayya [1997, 1999]; Valentin et al. [2000]; Zilio et al. [2004]) detailed solutions that consider different physical structures, and some

of these ideas were later transferred to commercial products (e.g., Agrawal et al. [2004]; Dageville et al. [2004]; Zilio et al. [2004]). This line of work, while successful, fails to address the common scenarios discussed in the introduction (which we collectively refer to as *physical design refinement*). In contrast to previous references, this work presents a new and complementary paradigm that considers the current physical database design and evolves it to meet new requirements.

Previous work in the literature adopted an ad hoc approach regarding the transformations that can be exploited for physical database design. Chaudhuri and Narasayya [1999] introduced a concept of index merging that is similar to what we define in this work, but does not generalize this notion to indexes over views. Similarly, Agrawal et al. [2000] exploited a few transformations to combine the information in materialized views without giving a formal and complete framework. Goldstein and Larson [2001] presented an overview of related work on view matching, which shares some of the technical details with our work, specifically with respect to view merging. We believe our work is the first to consider a unified approach of primitive operations over indexes and materialized views that can form the basis of physical design tools.

Some of the ideas in this work are inspired by Bruno and Chaudhuri [2005], which presented a relaxation-based approach for physical design tuning. This reference introduced the concept of relaxation to transform an optimal configuration obtained by intercepting optimization calls to another one that fits in the available storage. Unlike this work, the main focus in Bruno and Chaudhuri [2005] was to obtain an optimal design from scratch for a given workload and therefore the notion of transformations was of secondary importance. Specifically, Bruno and Chaudhuri [2005] considered transformations for indexes and materialized views as different entities, and did not provide a unifying framework.

More recently, Bruno and Chaudhuri [2006b, 2007] used the notion of merging indexes while investigating new directions in physical design tuning. Specifically, Bruno and Chaudhuri [2006b] provided quick lower and upper bounds on the expected benefit of a comprehensive tuning tool, and considered merging indexes as a crucial component in the main algorithm. On the other hand, Bruno and Chaudhuri [2007] proposed an alternative approach to the physical design problem. Specifically, it introduced algorithms that are always-on and continuously modify the current physical design reacting to changes in the query workload. Specifically, these techniques analyze the workload and maintain benefit and penalty values for current and hypothetical indexes (including merged indexes) and modify the current configuration in response to changes in the query workload.

This article extends the work in Bruno and Chaudhuri [2006a]. In addition to an expanded treatment of items in the original submission, this work addresses several new issues. First, we identified a novel approach (i.e., *profiled workloads*) to obtain knowledge on usage of the database that balance the accuracy of the resulting physical configuration and the required overhead (Section 4.2). In Section 5 we discussed a geometric interpretation of the PDR problem and our original solution that offers a different perspective to the PDR problem.

Based on this interpretation, we explored a new search alternative using backtracking for the original PDR problem in Section 5.1. We then introduced two alternative problem formulations to the PDR problem. Specifically, in Section 5.2 we introduced the Dual-PDR Problem (which minimizes the space used by the final configuration while not exceeding a bound in cost), and in Section 5.3 we introduced the constrained-PDR problem (which limits the number of transformations that may be applied to the original configuration). Finally, we formally defined the *Physical Design Scheduling* problem, an essential step in implementing physical design changes in Section 6.

The literature on query optimization is vast, and sometimes addresses problems that implicitly use the merge/reduce building blocks described in this paper. For instance, the multiquery optimization problem has a long history (see, e.g., Finkelstein [1982]; Park and Segev [1988]; Sellis [1988]; Roy et al. [2000]). The objective is to exploit common subexpressions across queries to reduce the overall evaluation cost (even though some queries in isolation might execute suboptimally). While detecting exact matches is already an improvement, many techniques extend exact matches with subsumed expressions. In other words, for a given pair of expressions e_1 and e_2, these techniques try to obtain the most specific expression that can be used to evaluate both e_1 and e_2. While the main issues in these references revolve around matching efficiency and greedy techniques to incorporate common subexpressions into regular query optimization, the problem statement can be easily reworded in terms of view merging and reduction. As another example, Ross et al. [1996] and Mistry et al. [2001] addressed the problem of materialized view maintenance. These authors showed how to find an efficient plan for the maintenance of a set of materialized views. Specifically, they exploited common subexpressions among the views and reached an analogous conclusion to that of Section 6: creating additional materialized views can reduce the total maintenance cost. In contrast to the physical design scheduling problem of Section 6, these references were concerned with the cost of *maintaining* the set of materialized views, where we pay attention to the complementary issue of how to *transition* between a given configuration and a new one. In any case, the work in Ross et al. [1996] and Mistry et al. [2001] can be rephrased in the context of merging and reduction operations and the search conducted on the closure of the original set of views.

Finally, the PDS problem of Section 6 is similar to the register allocation problem studied in the compiler literature [Chaitin et al. 1981]. The register allocation problem consists of allocating a large number of program variables into a small number of CPU registers in order to keep as many operands as possible in registers and thus maximize the performance of compiled programs. In both problems we have to schedule the use of scarce resources (disk vs. CPU registers) to minimize the execution cost of some program (physical restructuring vs. arbitrary compiled code). However, there are significant differences between both problems. First, the structures that we need to allocate in PDS can be of vastly different sizes while on the register allocation problem the size of the registers is fixed. Also, PDS allows more flexible schedules since we only are concerned with the initial and final configuration, but we can change the

order of intermediate operations arbitrarily. Finally, in PDS we might have to consider additional temporary structures that are not part of either the initial or final configuration to maximize performance. Thus, the solutions for the register allocation problem are not directly applicable to PDS, but there might be opportunities for further research for a better understanding of the relationship between the two problems.

9. CONCLUSIONS

In this work we introduce and study in depth several variants of the physical design refinement problem, which fills an important gap in the functionality of known physical design tools. Rather than building new configurations from scratch when some requirements change, we enable the progressive refinement of the current configuration into a new one that satisfies storage and update constraints. We do so by exploiting two new operators (*merging* and *reduction*) that balance space and efficiency. The configurations obtained via physical design refinement are also easily explained to the DBAs. We believe that this new functionality is an important addition to the repertoire of automated physical design tools, giving DBAs more flexibility to cope with evolving data distributions and workloads.

A. PROOFS

THEOREM A.1. (THE PDR PROBLEM IS NP-HARD).

PROOF. We provide a reduction from knapsack. The knapsack problem takes as inputs an integer capacity B and a set of objects o_i, each one with value a_i and volume b_i. The output is a subset of o_i whose combined volume fits in B and sum of values is maximized. Consider an arbitrary knapsack problem with capacity B and elements $\{o_1, \ldots, o_n\}$. We create a PDR(C, W, B) instance as follows. First, we associate each o_i with the view V_i = `SELECT x FROM` T_i `WHERE x=0`, where T_i is a single column table that contains b_i tuples with value zero and a_i tuples with value one. We then define the initial configuration $C = \{V_i\}$ and the representative workload also as $W = \{V_i\}$. Since all views refer to different tables, there is no possibility of merging views. Additionally, each index is defined over a single column, so no reduction is possible either. The PDR(C, W, B) problem then reduces to finding the best subset of the original indexes over views. Now, if V_i is not present in the final configuration, we have to scan the base table T_i to obtain the zero-valued tuples and answer q_i. Base table T_i is $a_i + b_i$ units of size, which is a_i units larger than the view size (there are only b_i tuples in T_i that satisfy $x = 0$). Assuming that scan costs are linear, the value of having V_i in the result (i.e., the time we save by having such an index) is a_i and its size is b_i. After solving this PDR(C, W, B) problem, we generate the knapsack solution by mapping the subset of views in the result to the original objects o_i. □

PROPERTY 4.4. *Let C be a configuration, IV_1 and IV_2 be indexes in* closure(C)*, and $IV_M = IV_1 \oplus IV_2$. If $IV_M \notin$ closure$(C -$ base$(IV_1))$, then PDR(C, B) cannot include both IV_1 and IV_M.*

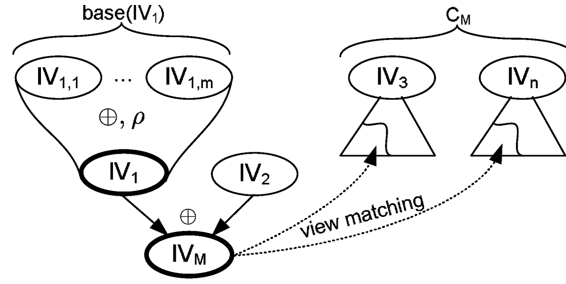


Fig. 17. Pruning indexes over views from the PDR search space.

PROOF. Suppose that both IV_1 and IV_M belong to PDR(C, B). Consider the indexes in C whose inferred queries are evaluated using IV_M (we call this set C_M in Figure 17). For each index $IV \in C_M$, it must be the case that IV_M matches either IV or some reduction of IV. Let us define the set C'_M as composed of the indexes in C_M (or their corresponding reductions) that are matched by IV_M. Now consider replacing IV_M in PDR(C, B) by $IV'_M = \oplus_{IV \in C'_M} IV$. We next show that this alternative configuration, denoted PDR'(C, B), is better than PDR(C, B). We first show that PDR'(C, B) is not larger than PDR(C, B). For that purpose, we note that IV'_M is obtained by merging elements in C'_M, which are all subsumed by IV_M. Therefore, $IV_M \oplus IV'_M = IV_M$ (the merged IV_M cannot incorporate anything that is not already captured by IV_M). Additionally, by our hypothesis, $IV'_M \neq IV_M$. The reason is that indexes in $base(IV_1)$ do not belong to C_M (the optimizer should have found better execution plans by replacing usages of IV_M with better alternatives that use IV_1). Therefore, $IV'_M \in closure(C_M) \subseteq closure(C - base(IV_1))$ and cannot be equal to IV_M. We then have that $IV'_M \oplus IV_M = IV_M$ and $IV'_M \neq IV_M$. Consequently, IV'_M is strictly smaller than IV_M and thus PDR'(C, B) is smaller than PDR(C, B). All queries inferred from indexes in $(C - C_M)$ cannot execute slower in PDR'(C, B) because all supporting indexes are present. Queries inferred from indexes in C_M would execute faster in PDR'(C, B) because the optimizer would replace usages of IV_M in the execution plans with more efficient alternatives that use the smaller IV'_M. PDR'(C, B) is also more efficient than PDR(C, B), which proves the property. □

PROPERTY 4.5. *Let C be a configuration, IV_1 and IV_2 be indexes in* closure(C)*, and $IV_M = IV_1 \oplus IV_2$. If* (*i*) size(IV_M) > size(IV_1) + size(IV_2)*, and* (*ii*) *for each $IV_k \in$* closure(C) *such that $IV_M = IV_M \oplus IV_k$, it still holds that* size(IV_M) > size(IV_1) + size(IV_2) + size(IV_k)*, then $IV_M \notin PDR(C, B)$.*

PROOF. Suppose that IV_M belongs to PDR(C, B) configuration but both (i) and (ii) do not hold. Since (i) does not hold, replacing IV_M by both IV_1 and IV_2 results in a smaller configuration. Additionally, every query inferred from an index in $base(IV_1) \cup base(IV_2)$ can be answered more efficiently by either IV_1 or IV_2 than it is by IV_M. There might be, however, some query inferred from an index IV_k that is not in $base(IV_1) \cup base(IV_2)$, and IV_k might greatly benefit from IV_M (see Figure 17). If that is the case, there is an IV'_k reduced from IV_k such

that $IV_M \oplus IV'_k = IV_M$. Since (ii) does not hold, we have that the combined size of IV_1, IV_2 and IV_k is smaller than that of IV_M, so we can replace IV_M by all IV_i and obtain a better configuration. In conclusion, IV_M cannot belong to PDR(C, B) if (i) and (ii) do not hold. □

REFERENCES

AGRAWAL, S., CHAUDHURI, S., KOLLAR, L., MARATHE, A., NARASAYYA, V., AND SYAMALA, M. 2004. Database tuning advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*.

AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. 2000. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases* (VLDB).

AGRAWAL, S., CHU, E., AND NARASAYYA, V. 2006. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

BRASSARD, G. AND BRATLEY, P. 1996. *Fundamental of Algorithmics*. Prentice Hall, Englewood Cliffs, NJ.

BRUNO, N. AND CHAUDHURI, S. 2005. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

BRUNO, N. AND CHAUDHURI, S. 2006a. Physical design refinement: The "Merge-Reduce" approach. In *Proceedings of the International Conference on Extending Database Technology* (EDBT).

BRUNO, N. AND CHAUDHURI, S. 2006b. To tune or not to tune? A lightweight physical design alerter. In *Proceedings of the International Conference on Very Large Databases* (VLDB'06).

BRUNO, N. AND CHAUDHURI, S. 2007. An online approach to physical design tuning. In *Proceedings of the International Conference on Data Engineering* (ICDE).

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. In *Computer Languages 6*, 1, 47–57.

CHAUDHURI, S., GUPTA, A. K., AND NARASAYYA, V. 2002. Compressing SQL workloads. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

CHAUDHURI, S. AND NARASAYYA, V. 1997. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Databases* (VLDB).

CHAUDHURI, S. AND NARASAYYA, V. 1999. Index merging. In *Proceedings of the International Conference on Data Engineering* (ICDE).

DAGEVILLE, B., DAS, D., DIAS, K., YAGOUB, K., ZAIT, M., AND ZIAUDDIN, M. 2004. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the 30th International Conference on Very Large Databases* (VLDB).

FINKELSTEIN, S. J. 1982. Common subexpression analysis in database applications. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

GOLDSTEIN, J. AND LARSON, P.-A. 2001. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

GRAEFE, G. 1995. The Cascades framework for query optimization. *Data Eng. Bull. 18*, 3.

KÖNIG, A. C. AND NABAR, S. U. 2006. Scalable exploration of physical database design. In *Proceedings of the International Conference on Data Engineering* (ICDE).

MISTRY, H., ROY, P., SUDARSHAN, S., AND RAMAMRITHAM, K. 2001. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

NILSSON, N. J. 1971. *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, NY.

PARK, J. AND SEGEV, A. 1988. Using common subexpressions to optimize multiple queries. In *Proceedings of the International Conference on Data Engineering* (ICDE).

ROSS, K. A., SRIVASTAVA, D., AND SUDARSHAN, S. 1996. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD).

SELINGER, P. G. ET AL. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data*.

SELLIS, T. K. 1988. Multiple-query optimization. *ACM Trans. Database Syst. 13*, 1, 23–52.

VALENTIN, G., ZULIANI, M., ZILIO, D., LOHMAN, G., AND SKELLEY, A. 2000. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering* (ICDE).

ZILIO, D. ET AL. 2004. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the 30th International Conference on Very Large Databases* (VLDB).

ZILIO, D., ZUZARTE, C., LIGHTSTONE, S., MA, W., LOHMAN, G., COCHRANE, R., PIRAHESH, H., COLBY, L., GRYZ, J., ALTON, E., LIANG, D., AND VALENTIN, G. 2004. Recommending materialized views and indexes with IBM DB2 design advisor. In *International Conference on Autonomic Computing*.

Received November 2006; revised March 2007; accepted June 2007