# Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design


Sanjay Agrawal
Microsoft Research
sagrawal@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Beverly Yang*
Stanford University
byang@stanford.edu



## ABSTRACT

In addition to indexes and materialized views, horizontal and vertical partitioning are important aspects of physical design in a relational database system that significantly impact performance. Horizontal partitioning also provides manageability; database administrators often require indexes and their underlying tables partitioned identically so as to make common operations such as backup/restore easier. While partitioning is important, incorporating partitioning makes the problem of automating physical design much harder since: (a) The choices of partitioning can strongly interact with choices of indexes and materialized views. (b) A large new space of physical design alternatives must be considered. (c) Manageability requirements impose a new constraint on the problem. In this paper, we present novel techniques for designing a scalable solution to this integrated physical design problem that takes both performance and manageability into account. We have implemented our techniques and evaluated it on Microsoft SQL Server. Our experiments highlight: (a) the importance of taking an integrated approach to automated physical design and (b) the scalability of our techniques.


## 1. INTRODUCTION

Horizontal and vertical partitioning are important aspects of physical database design that have significant impact on performance and manageability. Horizontal partitioning allows access methods such as tables, indexes and materialized views to be partitioned into disjoint sets of *rows* that are physically stored and accessed separately. Two common types of horizontal partitioning are range and hash partitioning. On the other hand, vertical partitioning allows a table to be partitioned into disjoint sets of *columns*. Like indexes and materialized views, both kinds of partitioning can significantly impact the performance of the *workload* i.e., queries and updates that execute against the database system, by reducing cost of accessing and processing data.

DBAs today also use horizontal partitioning extensively to make database servers easier to manage. If the indexes and the underlying table are partitioned identically i.e. *aligned*, database operations such as backup and restore become much easier. Therefore, there is a need to incorporate manageability while arriving at the right physical design for databases. Thus, database administrators (DBAs) in today's enterprises are faced with the challenging task of determining the appropriate choice of physical design consisting of partitioned tables, indexes and materialized views that (a) optimizes the performance of the SQL queries and updates and (b) is easier to manage at the same time.

Our goal is to optimize the performance of a database for a given representative workload, while considering alignment requirements. While there has been work in the area of automating physical database design [2,4,17,22,24], we are not aware of any work that addresses the problem of incorporating both horizontal and vertical partitioning as well as alignment requirements in an integrated manner. The novel techniques presented in this paper are motivated by the key design challenges that arise with the inclusion of horizontal and vertical partitioning, and are presented below.

**Need for an integrated approach to automating the choice of physical design**: Different aspects of physical design can interact strongly with one another. Example 1 illustrates the problems of separating the selection of different physical design choices.

**Example 1.** Consider the following query on TPC-H 1 GB data.

```
SELECT L_RETURNFLAG,        L_LINESTATUS,
          SUM (L_QUANTITY), COUNT (*)
FROM    LINEITEM
WHERE L_SHIPDATE <= '1998/12/08'
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS
```

We compare two approaches for the query above. (1) First select the best un-partitioned indexes, and in the next step horizontally partition the resulting indexes. (2) Consider indexes and horizontal partitioning together. Using the first approach we obtain as the best index, an index ($I_1$) on columns (*l_shipdate, l_returnflag, l_linestatus, l_quantity*) hash partitioned on (*l_returnflag, l_linestatus*). Using the second (integrated) approach, the best index ($I_2$) is (*l_returnflag, l_linestatus, l_shipdate, l_ quantity*) range partitioned on (*l_shipdate*). Note that the indexes $I_1$ and $I_2$ though defined over the same set of columns, differ in the ordering of columns as well as in the way they are partitioned. The execution time of the above query using $I_2$ is about 30% faster than with $I_1$.



*SIGMOD 2004*, June 13–18, 2004, Paris, France.



* Work done when the author was visiting Microsoft Research.

The reason for inferior recommendation using approach (1) is as follows. When selecting the best un-partitioned index, the alternative (*l_returnflag, l_linestatus, l_shipdate, l_quantity*) which has the column sequence as $I_2$ is considered, but is found to be much inferior than the index (*l_shipdate, l_returnflag, l_linestatus, l_quantity)*, where the columns in the index are ordered on selection and grouping columns of the query respectively. Thus, the ordering of the columns in the index is fixed after the first step. In the subsequent step, partitioning the index on the grouping/ordering columns in the query is found to be beneficial since this reduces the cost of sorting. However, since answering the query requires merging the rows from each partition of $I_1$, this adds to the query execution time. On the other hand, using approach (2) where indexes and partitioning are considered together, the superior alternative that range partitions the index on the column referenced in the range condition (which limits all the rows required to answer the query to a single partition) and orders the columns in the index on grouping/ordering columns (which saves the sort cost as well) is considered.

As demonstrated by Example 1 above, staging the solution based on different physical design features can result in poor overall physical design. Intuitively, the reason for inferior solution using the first approach is that both indexes and horizontal partitioning can speed up the same operations in the query (grouping, selections). By separating the choices we can get locked into a poor solution in the first step that cannot subsequently be undone. In this paper, we discuss the interactions among physical design structures, both within a single query as well as across queries in the workload that can cause staged solutions to perform poorly.

**Need for intelligent pruning of large search space:** With the inclusion of vertical and horizontal partitioning, the space of possible physical design alternatives that need to be considered for the given workload significantly increases. For example, each table can be vertically and horizontally partitioned in many different ways. Similarly, for each index or materialized view that we consider, we can have many variations of that structure, each horizontally partitioned in a different way. The fact that modern database systems support different ways of horizontal partitioning, such as range or hash partitioning, only adds to this combinatorial explosion. We present novel techniques that exploit workload information to intelligently prune the space of alternatives in a cost-based manner.

**Need for integrating alignment requirements into search:** A key contribution of the paper is to highlight the importance of incorporating manageability requirements while optimizing the database physical design for performance. In this paper, we focus on the *alignment* aspect of manageability. Indexes are considered as aligned if these are horizontally partitioned in the same way as the underlying tables. The challenge of incorporating alignment mainly arises from the fact that optimizing different queries can lead to physical design structures on the same table that have conflicting horizontal partitioning requirements. In this paper, we present a scheme that achieves alignment in an efficient manner.

In arriving at our solution, we leverage two important ideas from the architecture discussed in [2]. The approach in [2] restricts the search for the best physical design for a workload to (a) objects that are good for at least one query in the workload (done in *Candidate Selection step)* and (b) additional objects that are potentially good for the workload but not necessarily for individual queries (generated in *Merging step*). The best physical design for the entire workload is arrived at by searching over the set of objects described above (called the *Enumeration* step). For evaluating different physical design alternatives, the solution relies on *optimizer estimated costs* and *what-if extensions* that are available in several commercially available database servers [5, 22]. This allows the system to be robust and scalable; trying out numerous alternatives during search without physically implementing these is more efficient and does not disrupt the database server's normal mode of operation. Thus the contributions of this paper can be viewed as novel pruning and algorithmic techniques that allow the adoption of the broad architecture in [2] while expanding the scope of physical design to include horizontal and vertical partitioning as well as alignment requirements.

We extended Microsoft SQL server with the necessary interfaces to enable us to experimentally evaluate our techniques. Our experimental results show: (a) The importance of taking an integrated approach to the physical design problem (b) the impact of our pruning techniques on quality and scalability of the solution. The focus of this paper is on partitioning in a single-node environment (e.g., on an SMP), which is widely prevalent in today's enterprise databases installations [23]. We expect that some of the techniques described in this paper will also be important in physical design for multi-node environments for the same reason, i.e., interactions among different aspects of physical design.

The rest of the paper is organized as follows. Section 2 formally defines the integrated physical design problem, and describes the interactions among physical design structures. Section 3 presents an overview of the architecture of our solution, and Section 4 describes novel pruning strategies for reducing the space of alternatives introduced by vertical and horizontal partitioning. Section 5 shows how we incorporate partitioning during the *Merging* step described earlier. Section 6 presents our technique for handling alignment requirements. We present results of our experimental evaluation in Section 7, and discuss related work in Section 8. We conclude in Section 9.

# 2. BACKGROUND

## 2.1 Preliminaries

*Workload*: We model the workload as a set of SQL DML statements, i.e., SELECT, INSERT, DELETE and UPDATE statement. It can be obtained using profiling tools that are available on today's database systems. Optionally, with each statement Q in the workload, we associate a weight $f_Q$. For example, the weight may capture the multiplicity of that statement in the workload. Note that since the workload captures the inserts/updates/deletes that happen in the system, the maintenance and update costs of physical design alternatives get accounted for by our workload model.

*Vertical partitioning* of a table T splits it into two or more tables (which we refer to as *sub-tables*), each of which contains a subset of the columns in T. Since many queries access only a small subset of the columns in a table, vertical partitioning can reduce the amount of data that needs to be scanned to answer the query. In this paper, we will assume that each sub-table contains a disjoint set of columns of T, except for the *key* columns of T which are present in each sub-table. The key columns are required to allow "reconstruction" of an original row in T. Unlike indexes or materialized views, in most of today's commercial database systems there is no native DDL support for defining vertical partitions of a table. We simulate vertical partitioning by creating regular tables, one for each sub-table. The queries/updates in the workload that reference the original table T are rewritten to execute against the sub-table(s). Thus vertical partitioning as studied in this paper can be viewed as restricted form of *tuning the logical schema* of the database to optimize performance. Note that other access paths such as indexes and materialized views can be created over sub-tables to further improve query performance.

*Horizontal partitioning* affects performance as well as manageability. In general, any of the following access paths in a database can be horizontally partitioned: (1) A table (or sub-tables described above), which may be organized as a heap or clustered index, (2) A non-clustered index (3) A materialized view. We will refer to these un-partitioned access paths in this paper as *objects*. The horizontal partitioning of an object is specified using a *partitioning method*, which maps a given row in an object to a partition number. All rows of the object with the same partition number are stored in the same partition. As mentioned earlier, we focus on the case of single-node partitioning, i.e., all objects are present on a single machine (possibly an SMP). While multi-node partitioning can bring benefits of *availability* that is not possible with single-node partitioning, as shown in [23] and in this paper, single-node partitioning significantly impacts performance.

Today's database systems typically allow two main kinds of partitioning methods: *hash* or *range*. In hash partitioning, the partition number for a given row is generated by applying a *system* defined hash function on all columns in a specified set of partitioning column(s) of the object. A *hash partitioning method* is defined by a tuple **(C,n)**, where **C** is a set of *column types*, and **n** is the number of partitions. For example, let T be a table ($c_1$ int, $c_2$ int, $c_3$ float, $c_4$ date). The hash partition method $H_1$ = ({int, int}, 10) partitions T into 10 partitions by applying the hash function to the values of columns {$c_1$, $c_2$} in each row of T.

A *range* partitioning *method* is defined by a tuple **(c, V)**, where c is a column type, and V is an ordered sequence of values from the domain of c. For example, the range partitioning method $R_1$ = (date, <'01-01-98', '01-01-02'>) when applied to column $c_4$ on table T above , partitions T into 3 partitions, one per range defined by the sequence of dates. The first range is defined by all values ≤ '01-01-98' and the last range is defined by all values > '01-01-02'. For simplicity of exposition, we define *range partitioning over a single column* rather than a set of columns. We also do not consider other kinds of partitioning methods e.g., hybrid partitioning (consisting of range partitioning an object, and hash partition each range), and list partitioning [12].

**Definition 1**. A *physical design structure* is defined as an object and its associated partitioning method. We denote a physical design structure by (**O**,**P**,**C**) where **O** is an object (heap, index, materialized view), **P** is a partitioning method and **C** is the *ordered* set of columns of **O** on which **P** is applied. An un-partitioned object is denoted as P=ϕ,**C** =ϕ.

**Example 2.** The physical design structure (*lineitem*, ({int}, 4), {*l_orderkey*}) represents the table *lineitem* hash partitioned into 4 partitions on column {*l_orderkey*}.

Note that in Definition 1, the object O itself can be a sub-table (i.e., a vertical partition of an original table) or an index/materialized view defined on a sub-table. This allows us to consider physical design structures that combine the benefits of vertical and horizontal partitioning.

**Definition 2.** A *configuration* is a valid set of physical design structures, i.e., a set of physical design structures that can be realized in a database. Some validity constraints that apply to any given configuration: a table can be vertically partitioned in exactly one way, a (sub-)table can have at most one clustered index, a (sub-)table can be horizontally partitioned in exactly one way.

## 2.2 The Physical Design Problem

Our goal is to choose a configuration, such that the performance of a given workload is optimized, subject to a constraint on the total storage allowed for the configuration. Optionally the configuration may be constrained to be aligned i.e. all indexes are horizontally partitioned identically to the table on which they are defined. Given a statement Q in the workload, and given a configuration P we assume there exists a function Cost (Q, P) that returns the optimizer estimated cost of statement Q if the physical design of the database is P. Recently, commercial database systems support the necessary interfaces to answer such "what-if" questions without requiring the configuration P to be physically implemented. Details of such functionality can be found in [5,22], and are omitted. Figure 1 shows the physical design problem formulation.

> Given a database **D**, a workload **W**, and a storage bound **S**, find a configuration **P** whose storage requirement does not exceed **S**, such that $\Sigma_{Q\in W}$ $f_Q$. Cost(Q,P) is minimized. Optionally, **P** may be constrained to be aligned.

**Figure 1. Physical Design Problem.**

The horizontal partitioning problem has been showed to be NP hard [18]. We note that other sub-problems, e.g., index selection, materialized view selection, choosing vertical partitioning have previously shown to be NP-hard. We omit details due to lack of space.

## 2.3 Interactions among Physical Design Structures

In this section, we present the interactions arising from inclusion of horizontal and vertical partitioning that justifies the importance of selecting different physical design features (vertical partitioning, indexes, materialized views, horizontal partitioning) in an integrated manner. We believe that any solution to the physical design problem that ignores these interactions, can potentially suffer from poor quality recommendations.

**Intra-Query interactions** capture the impact of different physical design features on one another at the query processing level.

*Intersection of* conditions: Consider a query with the WHERE clause Age < 30 AND Salary > 50K. If neither condition by itself is very selective e.g. each selects 20% of records, indexing or partitioning on Age and Salary Column(s) are not very useful. However if the conjunction of their selectivities is small e.g. 5%, an index on Salary, range partitioned on the Age column, can benefit the query significantly. Note that similar interactions can also occur with two indexes (e.g., index intersection plans).

*Join* interactions: Two or more structures from different tables can share a common property than enables a faster join execution strategy; if these are partitioned identically on their respective join columns (i.e. are **co-located**), the query optimizer can select a plan that joins the corresponding partitions of the respective structures separately, and then combine the results. Co-located joins are typically much faster than non co-located joins, particularly on multi-processor systems, where joins of several different pairs of partitions can be performed in parallel. Even on a single processor the total cost of joining several smaller partitions can be much less than cost of a single large join; when each partition fits into memory, cost of join can decrease dramatically. This requires us to explore combinations of structures from different tables, partitioned identically, so that they can be used in a co-located join.

*Mutually* exclusive *structures*: This interaction is characterized by the fact that if one structure is chosen, then it eliminates (or makes redundant) other structures from consideration. For example, if we horizontally partition table T on column A, then it physically prohibits the table from being partitioned on any other column(s). Likewise, we can vertically partition T in *exactly one way*.

**Inter-Query** interactions arise from the fact that our goal is to find the best configuration for workload with certain constraints.

*Specificity vs. Generality*: Often a structure is only useful for a specific query, but not useful for any other query in the workload. A good example of this is a range partitioned table. Unless we are careful about picking boundary values of the range partitioning, we can suffer from being overly specific for a few queries, but poor for over all workload.

*Implications of Storage/Update*: A structure can be more beneficial but require more storage or have higher update cost for the workload than some other structure. This makes the task of searching for an optimal configuration in a storage-constrained and/or update intensive environment more difficult. This interaction suggests that when storage is limited, we expect structures such as partitioning and clustered indexes (both of which are non-redundant structures) to be more useful.

# 3. ARCHITECTURE OF SOLUTION

As mentioned in introduction, we adopt and extend the architecture described in [2]. For simplicity of exposition, we retain the terminology used in [2] wherever applicable. The focus of this paper is on novel techniques (and their evaluation) that make it possible to adopt this architecture for the physical design problem presented in Section 2. The issue of alternative architectures for the physical design problem is not considered in this paper, and remains an interesting open issue. Figure 2 shows the overall architecture of our solution. The four key steps in our architecture are outlined below.

***Column-Group Restriction:*** The combinatorial explosion in the number of physical design structures that must be considered is a result of the large number of column-groups (i.e., sets of columns) that are, in principle, relevant for the workload. This step (which is the focus of Section 4) is a pre-processing step eliminates from further consideration a large number of column-groups that can at best have only a marginal impact on the quality of the final solution. The output of this step is a set of "interesting" column-groups (defined in Section 4) for the workload. This step is a novel extension of previous architectures.

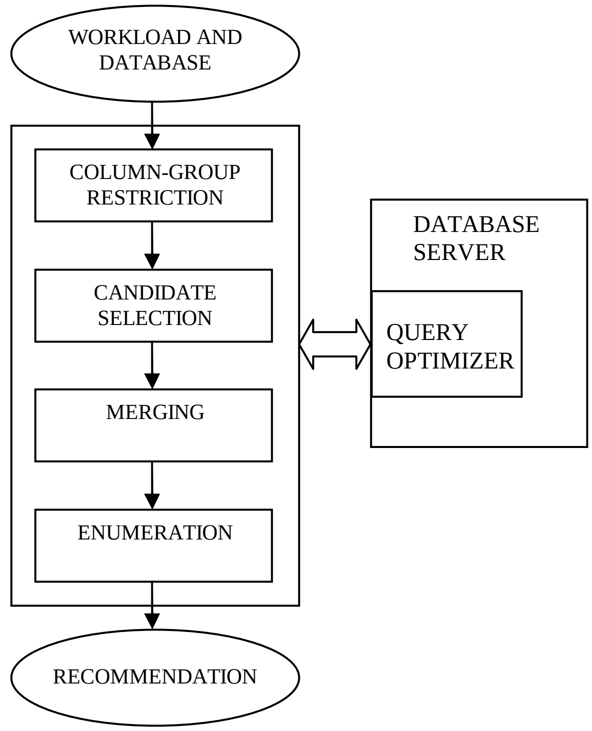


**Figure 2. Architecture of solution.**

***Candidate Selection:*** The interesting column-groups identified in the above step form the basis for the physical design structures we consider, described in Section 4.3. The Candidate Selection step selects for each query in the workload (i.e., one query at a time), a set of very good configurations for that query in a cost-based manner by consulting the query optimizer component of the database system. A physical design structure that is part of the selected configurations of one or more queries in the workload is referred to as a *candidate*. In our implementation, we use Greedy **(m,k)** approached discussed in [2] to realize this task. To recap, Greedy **(m,k)** algorithm guarantees an optimal answer when choosing up to **m** physical design structures, and subsequently uses a greedy strategy to add more (up to **k**) structures. The collocation aspect of intra-query interactions (see Section 2.3) is taken into account by selecting a value of **m** large enough ( m is the number of collocated objects in the query). Note that the specific algorithm to generate candidates is orthogonal to our solution as long it accounts for the interactions and returns a set of candidates for the query.

***Merging:*** If we restrict the final choice of physical design to only be a subset of the candidates selected by the Candidate Selection step, we can potentially end up with "over-specialized" physical design structures that are good for individual queries, but not good for the overall workload. Specifically, when storage is limited or workload is *update* intensive, this can lead to poor quality of the final solution. Another reason for sub-optimality arises from the fact that an object can be partitioned (vertically or horizontally) in *exactly one way*, so making a good decision can be crucial. The goal of this step is to consider new physical design structures, based on candidates chosen in the Candidate Selection step, which can benefit multiple queries in the workload. The Merging step augments the set of candidates with additional "merged" physical design structures. The idea of merging physical design structures has been studied in the context of un-partitioned indexes [6], and materialized views [2]. However in both these studies, the impact of vertical and horizontal partitioning on Merging was not considered. As we show in Section 5, Merging becomes considerably more challenging with the inclusion of both kinds of partitioning, and requires new algorithmic techniques.

***Enumeration:*** This step takes as input the candidates (including the merged candidates) and produces the final solution – a physical database design. The problem of exact search techniques for the physical design problem (e.g., as in [4,17,22,24]) is complementary to the focus of this paper. In our implementation, we use Greedy **(m,k)** search scheme. However, as we show in Section 6, with the additional constraint that the physical design for each table should be *aligned*, directly using previous approaches can lead to poor scalability. A contribution of this paper is describing an algorithm that improves scalability of the search without significantly compromising quality.

In Section 4, we discuss how we leverage workload to prune the space of syntactically relevant physical design structures. In Section 5, we show how we incorporate horizontal and vertical partitioning during Merging. In Section 6, we discuss how to handle alignment requirements in an efficient manner during the Enumeration step.

# 4. RESTRICTING COLUMN-GROUPS FOR CANDIDATE SELECTION

A physical design structure is *syntactically relevant* for the workload if it could potentially be used to answer one or more queries in the workload. As shown in several previous papers on physical database design e.g., [2,4,6], the space of syntactically relevant indexes and materialized views for a workload can be very large. We observe that for indexes, the space of syntactically relevant physical design structures is strongly dependent on the *column-groups*, i.e., combinations of columns referenced by queries the workload. Similarly, with horizontal partitioning, column-groups present in selections, joins and grouping conditions [17] of one or more queries need to be considered. With vertical partitioning, each column-group in a table could be a sub-table.

Once we include the options of vertical and horizontal partitioning along with indexes, even generating all syntactically relevant structures for a query/update (and hence the workload) can become prohibitively expensive. In Section 4.1, we present an efficient technique for pruning out a column-group such that *any physical design structures* defined on that column-group can only have limited impact on the overall cost of the workload.

An additional factor that must be considered is that it is much more expensive for a physical design tool to consider an alternative vertical partitioning than it is to consider an alternative index (or horizontal partitioning). The reason is (see Section 2.1.) simulating a vertical partitioning to the query optimizer requires creating regular tables, one for each sub-table in the vertical partition, and rewriting queries/updates in the workload that reference the original table to execute against the sub-table(s). Furthermore, sub-tables serve as building blocks over which we explore the space of horizontally partitioned indexes and materialized views. Thus, in large or complex workloads, where many interesting column-groups may exist, it can be beneficial to be able distinguish the relative merits of column-groups for vertical partitioning. In Section 4.2, we present a measure for determining effectiveness of a column-group for vertical partitioning that can be used by any physical design tool to filter/rank column-groups.

## 4.1 Determining Interesting Column-Groups

Intuitively, we consider a column-group as interesting for a workload W, if a physical design structure defined on that column-group can impact a significant fraction of the total cost of W. Based on this intuition, we define a metric *CG-Cost*(*g*) for a given column-group *g* that captures how interesting that column-group is for the workload. We define *CG-Cost* (*g*) as the fraction of the cost of all queries in the workload where column-group *g* is referenced. The cost of query can either be the observed execution cost of the query against the current database (if this information is available) or the cost estimated by the query optimizer. A column-group g is interesting if CG-Cost(g) ≥ **f** , where 0 ≤ **f** ≤ 1 is a pre-determined threshold.

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Example 3**. Consider a workload of queries/updates $Q_1$, … $Q_{10}$ that reference table T (A,B,C,D). A cell in the above matrix contains 1 if the query references that column, 0 otherwise. For simplicity assume that all queries have cost of 1 unit. Suppose the specified threshold **f** = 0.2. Then the interesting column-groups for the workload are {A}, {B}, {C}, {A,B}, {A,C}, {B,C} and {A,B,C}with respective CG-Cost of 1.0, 0.3, 0.9, 0.3, 0.9, 0.2, 0.2. For $Q_3$ we only need to consider physical design structures on the above 7 column-groups rather than the 15 column-groups that are syntactically relevant for $Q_3$, since {D} and all column-groups containing D are not interesting.

Note that *CG-Cost* is monotonic; for column-groups $g_1$ and $g_2$, $g_1 \subseteq g_2 \Rightarrow$ *CG-Cost* ($g_1$) ≥ *CG-Cost* ($g_2$). This is because for all queries where $g_2$ is referenced, $g_1$ is also referenced, as are all other subsets of $g_2$. Thus for a column-group to be frequent, *all its subsets* must be frequent. We leverage this monotonicity property of *CG-Cost* to build the set of all interesting column-groups of a workload in a scalable manner as shown in Figure 3, by leveraging existing algorithms for frequent-itemset generation e.g., as [1], rather than having to enumerate all subsets of columns referenced in the workload. We note that the above frequent-itemset approach has been used previously for pruning the space of materialized views [2]. NumRefColumns (in Step2) is the maximum number of columns of T referenced in a query, over all queries in the workload. The parameter to the algorithm is a threshold **f,** below which a column-group is not considered interesting.

In practice, in several large real and synthetic workloads, we have observed that even relatively small values of **f** (e.g., 0.02) can result in dramatic reduction in number of interesting column-groups (and hence overall running time of the physical design tool) without significantly affecting the quality of physical design solutions produced. This is because in practice, we observe distributions of *CG-Cost*() that are skewed; there are large number of column-groups that are referenced by a small number of inexpensive queries and these get pruned by our scheme. Our experimental evaluation of the effectiveness of the above technique is described in Section 7.

Let $G_1$ = {g | g is a column-group on table T of cardinality 1, and column $c \in g$ is referenced in the workload and *CG-Cost* (g) ≥ **f**}; i = 1<br>
**While** i < T.NumRefColumns and $|G_i| > 0$<br>
  i = i + 1; $G_i$ = {}<br>
  Let G ={g | g is a column-group on table T of size i, and $\forall s \subset g$, |s|=i-1, $s \in G_{i-1}$}<br>
    **For** each $g \in G$<br>
      **If** *CG-Cost* (g) ≥ **f** **Then** $G_i = G_i \cup \{g\}$<br>
    **End For**<br>
**End While**<br>
**Return** ($G_1 \cup G_2 \cup \ldots G_{T.NumRefColumns}$)

**Figure 3. Algorithm for finding interesting column-groups in the workload for a given table T.**

Finally, we note that a pathological case for the above pruning algorithm occurs when *CG-COST* of (almost) all column-groups in the workload is below **f**. In such cases, one possible approach is to dynamically determine **f** so that enough column-groups are retained.

## 4.2 Measuring Effectiveness of a Column-Group for Vertical Partitioning

As described earlier, evaluating a vertical partitioning can be an expensive operation for any physical design tool. We now present a function that is designed to measure the effectiveness of a column-group for vertical partitioning. Such a measure can be used, for example, to filter or rank interesting column-groups (identified using the technique presented in Section 4.1). Intuitively, a column-group *g* is effective for vertical partitioning if *all* columns in the group almost always *co-occur* in the workload. In other words, there are only a few (or no) queries where any one of the columns is referenced but the remaining columns are not.

**Definition 5**. The **VP-CONFIDENCE** (or **VPC** for short) of a column-group **g** is defined as:

$$\frac{\sum_{c \in g} width(c).\left|Occurrence(c)\right|}{\sum_{c \in g} width(c) \;.\left|\bigcup_{c \in g} Occurrence(c)\right|}$$

where c is a column belonging to column-group *g*, width(c) is the average width in bytes of c, and *Occurrence*(c) is the set of queries in the workload where c is referenced.

**Example 4**. In Example 3, for query $Q_1$, the set of columns referenced is {A,B}. Note that the interesting column-groups that could be considered for vertical partitioning for $Q_1$ are {A,B} and {A,B,C} since both these column-groups contain all columns referenced in $Q_1$. Assume that all columns are of equal width. Then **VPC**({A,B}) = 13/20 = 0.65, whereas **VPC**({A,B,C}) = 22/30 = 0.73. Thus using the **VPC** measure, we would prefer to vertically partition it on {A,B,C} than on {A,B}.

We note that VPC is a **fraction** between 0 and 1. The intuitive interpretation of VPC(*g*) is the following: If a vertical partition on *g* were defined, VPC(*g*) is the fraction of the scanned data would actually be useful in answering queries where one or more columns in *g* are referenced. Hence, column-groups with high VPC are more interesting. If the VPC of a column group *g* is high enough (e.g., VPC(*g*)=.9), then it is unlikely that the optimal partitioning will place the columns of *g* into separate sub-tables otherwise the cost of queries that reference *g* will significantly increase due to the cost of joining two or more sub-tables to retrieve all columns in *g*. The definition can be extended to incorporate *cost* of queries by replacing *Occurrence*(c) with the total cost of all queries in *Occurrence*(c).

## 4.3 Leveraging Column-Groups for Generating Physical Design Structures

We use the interesting column-groups (these are ranked by VPC) to generate relevant physical design structures on a per-query basis as follows. In the first step, we find **vertical partitions** per table. We apply the **VPC** measure to all interesting column-groups that contains *all the columns* referenced in query, and consider only the top **k** ranked by **VPC.** Each vertical partitioning considered has a sub-table corresponding to one such column-group. The remaining columns in the table form the second sub-table of the vertical partition. Note that every vertical partition generated by the above algorithm consists of exactly two sub-tables. We also consider the case where the table is not vertically partitioned. In the next step, for each vertical partitioning considered above (including the case where the table is not partitioned), we restrict the space of indexes and respective horizontal partitioning to the joins/selection/grouping/ordering conditions in the query where the underlying column-groups are interesting. For range partitioning, we use the specific values from the query as boundary points. For hash partitioning, we select number of partitions such that it is a multiple of number of processors and each partition fits into memory (we assume uniform partition sizes). We use the approach outlined in [2] to determine the materialized views we consider for a query. The considerations for horizontally partitioning materialized views are exactly the same as for tables i.e. partition on interesting joins/selection/grouping column(s) when the view is used to answer the query. We omit details due to lack of space.

## 5. INCORPORATING PARTITIONING DURING MERGING

The Merging step of our solution (described in Section 3) becomes much more complex when we have partitioning for two reasons. (1) Merging vertical partitions can become very expensive as each merged vertical partition potentially requires a new set of (partitioned) indexes and materialized views to be simulated as well. (2) The benefits of collocation have to be preserved during merging of horizontally partitioned indexes.

Although not the focus of this paper, an important aspect of generating new merged candidates is defining the space of merged candidates explored. Given a set of structures, which we refer to as *parent* structures, our goal is to generate a set of merged structures, each of which satisfies the following criteria. First, the merged structure should be usable in answering all queries where each parent structure was used. Second, the cost of answering queries using the merged structure should not be "much higher" than the cost of answering queries using the parent structures.

For exploring the space of merged structures, we adopt the algorithm from [2]. To recap, the algorithm iterates over the given set of candidates as follows. In each iteration, the algorithm generates all candidates that can be obtained by merging a pair of candidates. It then picks the best merged structures and replaces their parent candidates, and repeats this process. Thus the algorithm returns the "maximal" merged structures that can be obtained by repeatedly merging pairs of structures. Our focus is on the problem of *how to merge a pair of physical design structures* in the presence of vertical and horizontal partitioning. We expect that the intuition underlying the techniques we present would be useful in any scheme that performs merging.

The overall Merging step can be described as follows. First, we generate interesting vertical partitions of different tables by merging vertical partitions that are output of Candidate Selection. Next, for each single vertical partition (including the new merged ones), we merge all indexes and materialized views that are relevant for that vertical partition while taking horizontal partitioning into account. If indexes on the same (sub-) table are horizontally partitioned on the same column(s), we merge the respective partitioning methods to arrive at a more generic partitioning method. We describe these steps in detail below.

## 5.1 Merging Vertical Partitions

Given the best vertical partitioning for individual queries, the goal of Merging is to find new vertical partitionings that are useful across queries in the workload. A vertical partitioning that is best for one query may significantly degrade the performance of another query. Merging vertical partitions is a challenging problem in itself. Since each vertical partition itself is a *set of column-groups* (i.e., sub-tables), merging two vertical partitionings requires us to merge two *sets* of column-groups.

**Example 5**. Consider table T(A,B,C,D) from Example 3 and two vertical partitionings of T, $VP_1$ = {(A, B, C), (D)} and $VP_2$ = {(A, B), (C, D) }. For the merged vertical partitioning, we could consider (among other alternatives) {(A, B, C, D)} or {(A, B), (C), (D)} or {(A, C), (B, D)}.

For the example above, we highlight, in the following table, the impact of a few different vertical partition alternatives on two queries $Q_1$ and $Q_4$ from example 3.

For $Q_1$ (references only columns A and B) {(A,B),(C,D)} is the best among these as no join is needed and no redundant data is scanned; $Q_1$ is answered using (A,B) only. However the same vertical partitioning is much worse for $Q_4$ (references only columns A and C) as now both (A, B) and (C, D) needs to be scanned and joined to get required columns.

| Vertical partitioning | $Q_1$ | $Q_4$ |
|---|---|---|
| {(A,B,C,D)} | Joins: No<br>Extra Data Scan:Yes | Joins: No<br>Extra Data Scan:Yes |
| {(A,B),<br>(C,D)} | Joins: No<br>Extra Data Scan :No | Joins: Yes<br>Extra Data Scan:Yes |
| {(A,C),<br>(B,D)} | Joins: Yes<br>Extra Data Scan:Yes | Joins: No<br>Extra Data Scan:No |
| {(A),(B),<br>(C),(D)} | Joins: Yes<br>Extra Data Scan :No | Joins: Yes<br>Extra Data Scan:No |

This simple example highlights the fundamental issues when merging vertical partitions. As a consequence of merging vertical partitions, some queries can become more expensive due to (a) more joins that need to be done or (b) more redundant data that needs to be scanned. Thus, if the vertical partition is the entire table itself, we have optimal join characteristics but we could potentially incur a lot of redundant data scan. At the other extreme, if table is completely partitioned i.e. each column forms separate partition, we can have lots of joins but no redundant data scan.

**Input**: Two vertical partitionings $VP_1 = \{t_{11}, t_{12}, \ldots, t_{1n}\}$, $VP_2 = \{t_{21}, t_{22}, \ldots, t_{2m}\}$ for of a given table T. $T_s$ is the set of all columns in T.
**Function**: QUERIES (VP) over a vertical partition VP returns all queries for which VP was a candidate.
**Function**: COST (VP,W) returns cost of vertical partition VP for set of queries W.
**Output**: A merged vertical partitioning.

```
1.   S = {} // S is a set of sub-tables on T.
2.   For i = 1 to n
         For j=1 to m
               S = S ∪ {t1i ∪ t2j} ∪ {Ts –(t1i ∪ t2j)}
               S = S ∪ {t1i ∩ t2j} ∪ {Ts –(t1i ∩ t2j)}
         End For
     End For
3.   W = QUERIES (VP1) ∪ QUERIES (VP2)
4.   For all subsets VP of S that form valid vertical
     partitioning of T, return the VP with the minimal Cost
     () over W.
```

**Figure 4. Merging a pair of vertical partitionings of a table.**

Our algorithm for merging two vertical partitionings of a given table T is shown in Figure 4. The algorithm measures the impact of merging on the workload in terms of joins and redundant data scans. Steps 1-2 defines the exact space of sub-tables that are explored. We restrict the space of sub-tables over which the merged vertical partition can be defined to those that can be generated via *union* or *intersection* of sub-tables in the parent vertical partitionings. The union operation can decrease the number of joins required (and hence reduce join cost) to answer one or more queries, while intersection can decrease the amount of data scanned (and thereby decrease scan cost) in answering a query. We also require the complement of sub-tables to be present as the final output must be a valid vertical partition i.e. all the columns of table must occur in some sub-table. A desirable property of this approach is that whenever a column-group occurs in a single sub-table in both parents, it is guaranteed to be in the same sub-table in the merged vertical partitioning as well. For example, if the parents are $VP_1$ = {(A,B), (C,D)} and $VP_2$ = {(A,B,C), (D)}, then we are guaranteed that the column-group (A,B) will never be split across different sub-tables in the merged vertical partitioning.

While in principle COST (VP,W) can use the optimizer estimated cost as described in Section 2, for efficiency, we adopt a simpler Cost model that is computationally efficient and is designed to capture the above trade-off in join and scan cost. Thus, we instead define the COST (VP, W) for a vertical partition VP and workload W as the sum of scan cost of data and join cost for all queries $q \in W$. The join cost is modeled as linear function of individual sub-tables that are joined. Step 3 defines the set of queries over which we cost the generated vertical partitions to be candidates of either of the parents. In step 4 we enumerate the space of valid vertical partitions defined by the sub-tables above and return the one with the least cost.

Finally, we note that candidate indexes on $VP_1$ and $VP_2$ may also be candidates on the merged vertical partitioning. The exception to this is indexes whose columns appear in different sub-tables in the merged vertical partitioning. In Example 5 above, an index on columns (C,D) cannot exist in the vertical partitioning {(A,C),(B,D)} since C and D belong to different sub-tables.

## 5.2 Merging Horizontally Partitioned Structures

The inclusion of horizontal partitioning introduces new challenges during the Merging step. First, it is no longer sufficient to simply merge the *objects* (e.g., indexes) themselves, but we also need to merge the associated *partitioning methods* of each object. This is non-trivial since how we merge the objects may depend on the partitioning methods, and vice-versa. The underlying reason for this problem is that (as discussed in Section 2.3), the indexing columns and the partitioning columns can be interchangeably used. We illustrate this issue with the following example:

**Example 6**. Consider two structures: $I_1$ is an index on column A hash partitioned on column B and $I_2$ is an index on column A hash partitioned on C. If we merge the indexes and the partitioning methods separately, we would never be able to generate a structure such as index on (A,B) partitioned on C, which may be able to effectively replace both $I_1$ and $I_2$.

Second, when merging two partitioning methods, the number of possible partitioning methods for the merged structure can be large. Third, when merging two structures, we need to pay attention to the fact that one (or both) of the partitioned structures being merged may be used in co-located joins; since if a merged structure has a different partitioning method than its parent structure, it may no longer allow a co-located join. We now discuss our approach for each of these issues.

### *5.2.1 Merging Index and Partitioning Columns*

We will assume that the two structures being merged are $I_1 = (O_1, P_1, C_1)$ and $I_2 = (O_2, P_2, C_2)$, as per the notation defined in Section 2.1. Intuitively, there are two key ideas underlying our approach. First, we exploit the fact that the partitioning columns and the index columns of the parent structures, i.e., the columns in $O_1$, $O_2$, and the columns $C_1$ and $C_2$ can be interchangeably used. Second, we restrict the space considered to those merged structures such that the benefit of *at least one* of the parent structures is retained. This is similar to the idea of index preserving merges described in [6], with the extension that the partitioning columns may also need to be preserved in the merged structure. Based on the above ideas, the space of merged structures we consider is as described below. We denote a merged structure as $I_{12} = (O_{12}, P_{12}, C_{12})$. Also, we denote columns in an object O as Cols (O).

The *partitioning* columns of $I_{12}$, i.e., $C_{12}$ can be one of the following: (a) $C_1$ (b) $C_2$ or (c) $C_1 \cap C_2$ (d) Cols ($O_1$) (e) Cols ($O_2$), (f) Cols ($O_1$) $\cap$ Cols ($O_2$). Thus in addition to the original partitioning (resp. indexing) columns, we also consider the intersection of the partitioning (resp. indexing) columns of $I_1$ and $I_2$, which is a more "general" partitioning. For example, if a table T partitioned on (A,B) is used to answer a query on T with a GROUP BY A,B clause and the table partitioned on (A,C) is used to answer a different query on T with a GROUP BY A,C clause, then, T partitioned on (A) can be used to answer both queries (partial grouping). We observe that $C_1 \cap C_2$ may be empty. In such cases the merged structure is un-partitioned.

The *index* columns of a merged structure, i.e., $O_{12}$ can have as its leading columns any one of: (a) Cols ($O_1$) in sequence, (b) Cols ($O_2$) in sequence (c) columns of $C_1$ in order (d) columns of $C_2$ in order. This ensures that the merged structure will retain the benefit (either indexing or partitioning benefit) of at least one of its parents. To the leading columns, we append all remaining columns from $O_1$, $O_2$, $C_1$ or $C_2$, which are not already part of the partitioning columns of $C_{12}$.

Thus our procedure considers all combinations of merged structures that can be generated from the space of possible $C_{12}$ and $O_{12}$ as defined above. The exact procedure for determining the partitioning method $P_{12}$ of the merged structure is described in Sections 5.2.2. Finally, similar to [2,6] we disallow a merged structure that is "much larger" in size (defined by a threshold) than the original candidate structures from which it is derived since such a merged structure is likely to degrade performance of queries being served by the original candidate structures.

### *5.2.2 Merging Partitioning Methods*

**Merging Range Partitioning Methods:** Given a pair of range partitioning methods $P_1 = (S, V_1)$, $P_2 = (S, V_2)$ we wish to find the best partitioning method $P_{12} = (S, V_{12})$ to be used with the merged object $O_{12}$. The best partitioning method is one such that the cost of all queries answered using $(O_1,P_1,C)$ (denoted by the set $R_1$) as well as $(O_2,P_2,C)$ (denoted by the set $R_2$) increases as little as possible when answered using $(O_{12},P_{12},C)$. The naïve approach of considering all possible partitioning methods $P_{12}$ that can be generated by considering any subset of the values in $V_1 \cup V_2$ is infeasible in practice.

Observe that if we do not partition an index, all queries need to scan the entire index resulting in high scan cost, but only pay a small cost in opening/closing the single partition. At the other extreme, if we partition the index into as many partitions as number of distinct values, each query is served by scanning the least amount of data required but may access a large number of partitions i.e. high cost of opening/closing partitions. Both these extremes can be sub-optimal. We need to balance the scan and partition opening/closing costs to arrive at a good compromise.

We now present an efficient algorithm **MergeRanges** for finding a merged range partitioning method. We model the cost of scanning a range partitioned access method, denoted by *Cost-Range,* for any range query Q as follows: (1) The cost of scanning the subset of partitions necessary to answer Q. This cost is proportional to the total size of all the partitions that must be scanned. (2) A fixed cost per scanned partition corresponding to the CPU and I/O overhead of opening and closing the B+-Tree for that partition. Note that computing Cost-Range is computationally efficient and does not require calls to the query optimizer. Thus our problem can be stated as finding a $V_{12}$ such that $\Sigma_{Q\in\ (R1\ \cup\ R2)}$ Cost-Range(Q, ($O_{12}$, (S,$V_{12}$),C)) is minimized, where $R_1$ (resp. $R_2$) is the set of queries for which the input objects are candidates. *MergeRanges* is a greedy algorithm that starts with $V_{12}$, a simple merging of sequences $V_1$ and $V_2$. In each iteration, the algorithm merges the next pair of adjacent intervals (from among all pairs) that reduces *Cost-Range* the most. The algorithm stops when merging intervals no longer reduces *Cost-Range.* Note that performance of *MergeRanges* varies quadratically with the number of boundary points. However in practice, we have observed the algorithm to perform almost linearly; the data distribution tends to be skewed causing the algorithm to converge much faster. *MergeRanges* does not guarantee an optimal partitioning; however our experimental evaluation suggests that it produces good solutions in practice, and is much more efficient compared to a solution that arrives at an optimal solution by considering all range partitioning boundary values exhaustively. We omit experiments and counter examples due to lack of space.

**Merging Hash Partitioning Methods:** For merging a pair of objects ($O_1$,$P_1$,C) and ($O_2$,$P_2$,C) where $P_1$ and $P_2$ are hash partitions on C with number of partitions as $n_1$ and $n_2$ respectively, the number of partitions of the merged object $O_{12}$ is determined by number of processors, available memory and number of distinct values in C.

### *5.2.3 Co-Location Considerations in Merging*

The merging described thus far does not pay attention to merged structures on other tables – this can result in loss of co-location (see Section 2.3) for merged candidates. Consider a simple case where we have 2 candidate indexes $I_1$ and $I_2$ on table $T_1$ that we wish to merge. Assume that $I_1$ is used in a co-located join with index $I_3$ on table $T_2$, i.e., both $I_1$ and $I_3$ have the same partitioning method (say $P_1$). Now $I_1$ and $I_2$ get merged to produce $I_{12}$ with a partitioning method $P_{12}$. Since $P_{12}$ is potentially different from $P_1$, the merged index $I_{12}$ can no longer be used in a co-located join with $I_3$, thereby greatly reducing its benefit. Thus to preserve the benefits of co-location, we need to also consider a merged structure where $I_{12}$ is partitioned on $P_1$. For each merged structure O, we consider all partitioning methods of any other structure that could be used in a co-location join with O. In our experiments (see Section 7) we show how co-location consideration in Merging is crucial for good quality solutions.

## 6. INCORPORATING ALIGNMENT REQUIREMENTS

As discussed in Section 3, the input to the final search step in our solution (i.e., the Enumeration step), is the set of candidate structures that have been found to be best for individual queries in the workload, or those introduced via the Merging step. In general, the input structures (for any given table) can be partitioned differently since different queries in the workload may be served best by different partitioning requirements. However, due to manageability reasons we may be required to provide as a solution, a configuration where all structures on each table are *aligned*, i.e., partitioned identically.

We assume that a specific search strategy (Greedy (**m, k**)) described in Section 3 is used in the Enumeration step. To recap, Greedy **(m,k)** algorithm guarantees an optimal answer when choosing up to **m** physical design structures, and subsequently uses a greedy strategy to add more (up to **k**) structures. The key challenges in meeting these alignment requirements are outlined below. For simplicity, let us assume that the structures $I_j$ (j=1, .. *n*) horizontally partitioned using partitioning method $P_j$ are input candidates to Enumeration and all structures are on same table and are partitioned on same columns, and that each $P_j$ is distinct. If we use Greedy (**m, k**) unmodified on the above *n* structures, we will end up picking exactly one of the structures $I_j$ above (adding any more will violate alignment), and thus can lead to a poor solution. An alternative way to address the alignment issue is to generate new structures which are the cross product all structures in I with all partitioning methods in P. These new structures, along with the *n* original structures (resulting in a total of $n^2$ structures) are then passed into the Greedy(m,k) algorithm. This approach, which we refer to as **Eager Enumeration,** although will result in a good quality solution, can cause a significant increase in running time as the number of input structures have been considerably increased (potentially by a quadratic factor). Thus, the challenge is to find a solution that is much more scalable then Eager Enumeration and at the same time does not significantly compromise the quality.

Our solution leverages the following observation. If we have a candidate structure **C** that enters the search step, and we alter only its partitioning method (to enforce alignment), then the resulting structure **C'** is *at most as good* in quality as **C** for any query in the workload where **C** was a candidate. Furthermore, for the given workload, suppose the cost of updating **C** is no higher than the cost of updating **C'** and the size of **C** is no larger than the size of of **C'**. Then, we note that **C'** can be introduced *lazily* during Greedy (m,k) without compromising the quality of the final answer, i.e., we would get the same answer as we would have obtained using Eager Enumeration. We have observed, that the above assumptions on update and storage characteristics typically hold for new structures that need to be generated to enforce alignment since: (1) the partitioning columns of C and C' often are the same (e.g., partitioning on join columns is common) and (2) differences that arise from specific partitioning method typically have small impact on update characteristics or size. (e.g., changing number of hash partitions does not significantly change size of structure or cost of updating structure).

Our solution builds on this observation and interleaves *generation of new candidates and configurations with search*. We call this **Lazy Enumeration.** The pseudo code for lazy enumeration with respect to the Greedy **(m,k)** search is described in Figure 5. We assume that all input structures are on same table and can differ on partitioning columns and methods. The extensions for different tables are straightforward and are omitted.

Steps 1-6 of the algorithm describe how we generate the optimal configuration **r** of size up to **m** from the input set of structures such that all structures in **r** are aligned. The idea is to get the best configuration of size up to **m** *without taking alignment into account*. If structures in the best configuration are not aligned, then only we generate new structures and configurations from structures in the best configuration that are aligned. In steps 8-10, we add structures greedily to **r** one at time to get up to **k** structures. If the added structure **a** causes the alignment to get violated, we generate **a'**, a version of **a** that is aligned with structures in **r,** and add **a'** to (and remove **a** from) the set of structures from which structures are picked greedily.

**Input:** **I**=$\{I_j \mid I_j=(O_j,P_j,C_j)$, $1 \le j \le n$, $I_j$ is a physical design structure$\}$, workload **W**, **k** and **m** in Greedy (**m,k**)
**Output:** A configuration with least cost for **W** having all structures aligned.

1. Let **S** be the set of all configurations of size up to **m** defined over structures in I.
2. Let **r** be the configuration in **S** with the minimal cost for **W**. If no such configuration exists, return an empty configuration. If all structures in **r** are aligned go to 7.
3. Let **P** be the set of all partitioning methods and columns of structures in **r**.
4. Let **T** be the set of all structures generated by partitioning structures in **r** using partitioning methods and columns in **P**. Note that **T** defines a cross product set of (o,p,c) where (o,*,*) is a structure in **r** and (p,c) is a partitioning method and column in **P**.
5. Let **S'** be the set of all configurations defined over structures in **T** where each configuration in **S'** is aligned and is of same size as size of **r**.
6. **S** = **S** $\cup$ **S'**, **S** = **S** – {**r**}. Go to 2. //Remove **r** from search
7. Let **V** = **I**. // Initialize **V** to be the same as **I**
8. If size of **r** $\geq$ **k**, return **r**. Pick a structure **a** from **V** such that configuration **r** $\cup$ {**a**} has the minimal cost for **W**. If no such structure can be picked, return **r**. If **a** is aligned with structures in **r**, go to 9, else go to 10.
9. **V** = **V** – {**a**}, **r** = **r** $\cup$ {**a**}. Go to 8. // add **a** to **r**
10. **V** = **V** – {**a**}, Let **a'** be the structure generated by partitioning **a** using (p,c) where structures in **r** are partitioned on columns c using partitioning method p. **V** = **V** $\cup$ { **a'**}. Go to 8.

**Figure 5. Extending Greedy (m,k) for handling alignment**

Note that we introduce a structure with a new partitioning only when the original structure would have been picked by the search scheme in the absence of alignment requirement. The savings in running time from the above algorithm arise due to the greedy nature of Greedy(m,k), which makes it unnecessary to introduce many new structures lazily that would otherwise have been introduced by the Eager Enumeration approach. In Section 7, we compare the Eager and Lazy Enumeration strategies on different workloads and show that the latter is much more scalable than Eager Enumeration without significantly compromising quality.

## 7. EXPERIMENTS

We have implemented the techniques presented in this paper on a commercial database server that has necessary server extensions to simulate indexes/materialized views/partitions. We simulate the effect of vertically partitioning a table as described in Section 2.1.

Next we present experiments conducted on our prototype implementation. We show that: (1) Our integrated approach to physical design is better than an approach that stages the physical design based on different features. (2) Column-group pruning (Section 4) is effective in reducing the space of physical design structures (3) Collation must be considered during Merging and (4) The Lazy Enumeration technique discussed in Section 6 to generate aligned indexes performs much better compared to the eager strategy without compromising quality.

***Setup:*** All experiments were run on a machine with an x86 1GHz processor with 256 MB RAM and an internal 30GB hard drive and running a modified version of a commercial relational DBMS. We use TPC-H database [21] in our experiments. We use notation TPCH1G to denote TPC-H data of size 1GB and TPCH22 for TPC-H 22 query benchmark workload.

***Importance of Selecting Structures Together:*** Here we study the importance of selecting physical design structures together. Figure 6 compares the reduction in quality (difference in optimizer estimated cost for TPCH22 workload) compared to our approach that selects vertical partitions, indexes and horizontal partitions in an integrated manner (TOGETHER). We compare our approach to (a) IND-ONLY where we select indexes only (no horizontal or vertical partitioning) and (b) VP->IND->HP where we first vertically partition the database, then select indexes and subsequently horizontally partition the objects. We vary the total available storage from 1.3 GB to 3.5 GB.

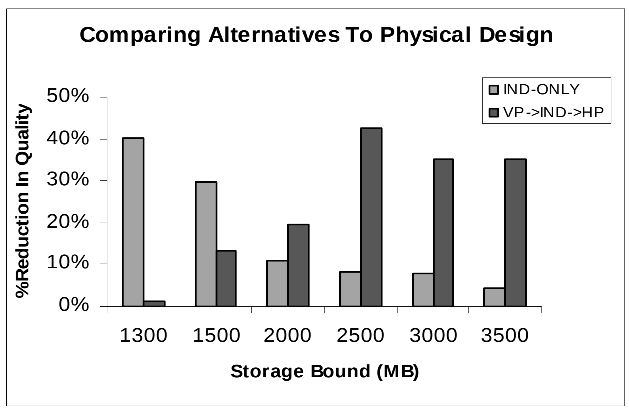


**Figure 6. Quality vs. Storage of physical design alternatives.**

First, we observe that at low to moderate storage (1.3GB to 2.0GB), TOGETHER is much superior to IND-ONLY. This is because unlike indexes (which are redundant structures) both kinds of partitioning incur very little storage overhead. Second, VP->IND->HP is inferior across all storage bounds to TOGETHER. The reason is that the best vertical partitioning chosen in isolation of indexes ends up precluding several useful indexes. Likewise, picking indexes without regard to co-location considerations of horizontal partitioning also results in missing out good solutions. Note that at large storage bounds (where importance of partitioning is diminished), TOGETHER is still better than IND-ONLY (but not by much), and much better than VP->IND->HP. Results were similar for updates and materialized views and have been omitted due to lack of space.

***Effectiveness of Column-Group Restriction:*** We study the effectiveness of the column-group based pruning (Section 4). We use two workloads TPCH22 and CS-WKLD and varied the threshold (**f**) for pruning from 0.0 (no pruning) to 0.1. CS-WKLD is a 100 query workload over TPCH1G database consisting of SPJ queries; the specific tables and selection conditions are selected uniformly at random from the underlying database. Figure 7 shows the reduction in quality for different values of **f** for the two workloads compared to f = 0.0. We observe that using column-group based pruning with **f** less than 0.02 has almost no adverse effect on quality of recommendations. For TPCH22 there was a 2% quality degradation at **f** = 0.02 compared to **f** = 0.0. We observe that the quality degrades rapidly for **f** > 0.02 for CS-WKLD because poor locality forces us to throw away many useful column-groups. Figure 8 shows the *decrease* in total running time of tool as **f** is varied, compared to the time taken for **f** = 0.0. We observe that the running time decreases rapidly as f is increased. For TPCH22, we observe about 20% speedup. This is not surprising since the space of column-groups is strongly correlated with the space of physical design that we explore. This experiment suggests that a value of **f** around 0.02 gives us about the same quality as f = 0.0 and in much less running time.

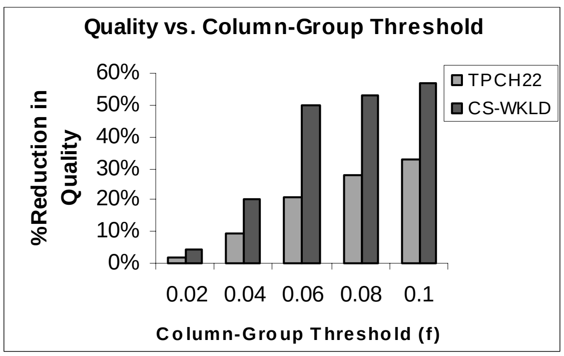


**Figure 7. Variation of Quality with Threshold f**

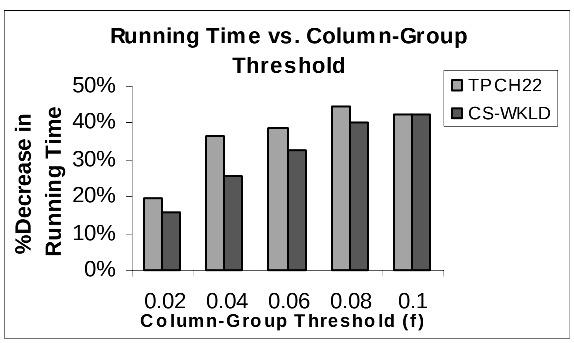


**Figure 8. Variation of Running Time with Threshold f**

***Importance of Co-location in Merging:*** Here we compare our algorithm for Merging (see Section 5) with a variant of this algorithm (NOCOL) that does not take co-location into consideration. We use 4 workloads of 25 queries each on TPCH1G database – WKLD-COL-n where n is % of queries with join conditions. We use n = 20, 40, 60 and 80. The specific values in the filter condition of the queries are generated randomly and the range lengths are selected with Zipfian distribution [13] (skew 1.0). Figure 9 shows the percentage reduction in quality of NOCOL of the workloads compared to our Merging scheme. We observe that as the % of join queries increases (e.g., at n=60), ignoring co-location causes the quality to drop significantly. The smaller difference at n=80 is because the overall workload cost has increased with more join queries causing the relative difference to become smaller.

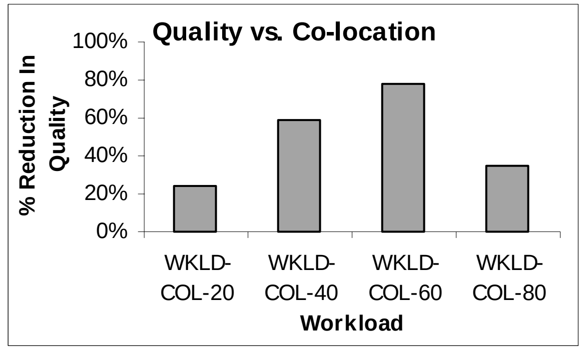


**Figure 9. Impact of co-location considerations on merging**

***Effectiveness of Lazy Enumeration for handling alignment requirements:*** Here we compare the Lazy Enumeration technique that we use to generate aligned indexes to Eager Enumeration, discussed in Section 6. Table 2 compares the two techniques. We use TPCH22 workload on TPCH1G database. We also use 200 APB queries on APB database [14] that is about 1.2 GB; the APB queries are complex decision support queries. We observe that on TPCH22 Lazy Enumeration performs much better, it is about 90% faster than Eager Enumeration and the loss in quality is very small ~1%. The reason for this is that Eager Enumeration generates and evaluates the goodness of lot more indexes compared to the lazy strategy. In the latter, new candidate indexes are generated on demand. We observe similar behavior for APB benchmark queries. This shows that Lazy Enumeration is much more scalable and almost as good in quality compared to Eager Enumeration.

**Table 2. Comparing quality and performance of Eager and Lazy Enumeration Techniques**

| Workloads | Speed up compared to Eager Enumeration | Loss in Quality compared to Eager Enumeration |
|---|---|---|
| TPCH22 | 90% | 1% |
| APB | 50% | 0% |

# 8. RELATED WORK

To the best of our knowledge, ours is the first work to take an integrated approach to the problem of choosing indexes, materialized views, and partitioning, which are the common physical design choices in today's database systems. The problem of how to automatically recommend partitioning a database across a set of nodes in shared-nothing parallel database system was studied in [17,25]. However, the key differences with our work are: (1) Their work does not explore the interaction between choice of partitioning and choice of indexes and materialized views. Thus, they implicitly assume that the two tasks are staged (i.e., done one after the other). As shown in this paper, in a single-node environment, such an approach can lead to poor quality of recommendations. (2) Our work also presents techniques for recommending appropriate range and hash partitioned objects. In [24], the problem of determining appropriate partitioning keys for a table (in a multi-node scenario) as well as indexes is considered.

Our work is a significant extension of this work in the following ways: (1) In addition to partitioning of tables, we also consider partitioning of indexes and materialized views. (2) We also consider range partitioning. (3) The focus in [24] was on the search problem (a branch-and-bound strategy). While this is an important aspect of the problem, we have argued in this paper for scalable techniques for selecting candidate physical design structures, which enable the search strategy to scale in practice. Recently, Zeller and Kemper [23] showed the importance of horizontal partitioning in a single-node scenario (for a large scale SAP R/3 installation), which is the focus of this paper. Their study showed the benefits of single-node partitioning for joins, and exploiting parallelism of multiple CPUs. The problem of allocating data fragments (horizontal partitions) of the fact table using bitmap indices in a data warehouse environment (star schemas) is studied in [20]. They explore the issues of how to fragment the fact table, as well as physical allocation to disks (e.g., degree of declustering, allocation policy).

There have been several papers [7,8,10,14] describing techniques for vertically partitioning a table for a given workload. Relative to this body of work, this paper is different in or more of the following ways. (1) We study the interaction of vertical partitioning with other physical design features. (2) Our approach is cost-based and takes into account usage of structures by the query optimizer. (3) We do not restrict our space to only binary partitioning (see the Merging step). Our techniques assume DBMS architectures prevalent in today's database systems. We note that recent studies [3,16] have looked into new DBMS architectures where improved cache performance is possible by storing data in a column-wise manner. We view this work as complementary to ours as refinements in the cost model based on these ideas can lead to better choice of vertical partitioning.

The problem of generating more general candidates based on merging indexes [6] and materialized views [2] have been studied earlier. However the problem of merging is significantly more complex in the presence of partitioning (Section 5). Our techniques for merging both range and hash partitioned objects as well as vertical partitioning are novel contributions of this paper. Finally, we note that there have been several papers (e.g., [9]) on the problems of index selection and materialized view selection, the latter mostly in the context of OLAP and Data cube. However, these studies differ in one or more of the following ways from our work. (1) They do not take into account the workload. (2) Their approach is disconnected from query optimizer. (3) Class of queries considered does not consider the full generality of SQL.

## 9. CONCLUSION

In this paper, we present techniques that enable a scalable solution to the integrated physical design problem of indexes, materialized views, vertical and horizontal partitioning for both performance and manageability. Our work on horizontal partitioning focuses on single-node partitioning. In the future we will investigate how our techniques can be adapted to handle performance and availability requirements in multi-node partitioning.

## 10. ACKNOWLEDGMENTS

We thank Lubor Kollar, Arun Marathe, Campbell Fraser and Alex Boukouvalas in the Microsoft SQL Server team for helping us with the necessary server extensions. We are grateful to Nico Bruno, Surajit Chaudhuri, Christian Konig and Zhiyuan Chen for their valuable discussions and feedback.

## 11. REFERENCES

[1] Agrawal, R., Ramakrishnan, S. *Fast Algorithms for Mining Association Rules in Large Databases*. Proc. of VLDB 1994.

[2] Agrawal, S., Chaudhuri, S., and Narasayya, V. *Automated Selection of Materialized Views and Indexes for SQL Databases*. Proceedings of VLDB 2000.

[3] Ailamaki A., Dewitt D.J., Hill M.D., and Skounakis M. *Weaving Relations for Cache Performance*. VLDB 2001.

[4] Chaudhuri, S., Narasayya, V. *An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server*. VLDB 1997.

[5] Chaudhuri, S., and Narasayya, V. *AutoAdmin "What-If" Index Analysis Utitlity*. Proc. of ACM SIGMOD 1998.

[6] Chaudhuri, S., and Narasayya, V. *Index Merging*. Proceedings of ICDE 1999.

[7] Cornell D.W., Yu P.S. *An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases*. IEEE Transactions on Software Engg, Vol 16, No 2, 1990.

[8] De P., Park J.S., and Pirkul H. *An Integrated Model of Record Segmentation and Access Path Selection for Databases*. Information Systems, Vol 13 No 1, 1988.

[9] Gupta H., Harinarayan V., Rajaramana A., and Ullman J.D. *Index Selection for OLAP*. Proc. ICDE 1997.

[10] Navathe S., Ra M. *Vertical Partitioning for Database Design: A Graphical Algorithm*. Proc. of SIGMOD 1989.

[11] http://otn.oracle.com/products/oracle9i/index.html.

[12] http://research.microsoft.com/~gray/dbgen/.

[13] http://www.olapcouncil.org/research/bmarkco.htm.

[14] Papadomanolakis, E., and Ailamaki A. *AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning*. CMU Technical Report. CMU-CS-03-159, July 2003.

[15] *Program for TPC-D data generation with Skew*. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[16] Ramamurthy R., Dewitt D.J., and Su Q. *A Case for Fractured Mirrors*. Proceedings of VLDB 2002.

[17] Rao, J., Zhang, C., Lohman, G., and Megiddo, N. *Automating Physical Database Design in a Parallel Database*. Proceedings of the ACM SIGMOD 2002.

[18] Sacca D., and Wiederhold G. *Database Partitioning in a Cluster of Processors*. ACM TODS,Vol 10,No 1, Mar 1985.

[19] Stohr T., Martens H.., and Rahm E.. *Multi-Dimensional Aware Database Allocation for Parallel Data Warehouses*. Proceedings of VLDB 2000.

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

[21] Valentin, G., Zuliani, M., Zilio, D., and Lohman, G. *DB2 Advisor: An Optimizer That is Smart Enough to Recommend Its Own Indexes*. Proceedings of ICDE 2000.

[22] Zeller, B., and Kemper, A. Experience Report. *Exploiting Advanced Database Optimization Features for Large-Scale SAP R/3 Installations*. Proceedings of VLDB 2002.

[23] Zilio, D. *Physical Database Design Decision Algorithms and Concurrent Reoganization for Parallel Database Systems.* PhD Thesis, Dept. of Comp. Sc., Univ. of Toronto, 1998.

[24] Zilio, D., Jhingran, A., Padmanabhan, S. *Partitioning Key Selection for Shared-Nothing Parallel Database System*. IBM Research Report RC 19820. 1994.