# AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning

Efstratios Papadomanolakis and Anastassia Ailamaki
CMU-CS-03-159
July 2003

School of Computer Science
Carnegine Mellon University
Pittsburgh, PA 15213

email: {stratos,natassa}@cs.cmu.edu

Keywords: relational databases, performance, self-tuning, vertical partitioning

Abstract

Database applications that use multi-terabyte datasets are becoming increasingly important for scientific fields such as astronomy and biology. To improve query execution performance, modern DBMS build indexes and materialized views on the wide tables that store experimental data. The replication of data in indexes and views, however, implies large amounts of additional storage space, and incurs high update costs as new experiments add or change large volumes of data. In this paper we explore automatic data partitioning as a tool to redesign the relational tables in a database for faster sequential access before creating indexes and views. We present AutoPart, a vertical partitioning tool that uses the optimizer's hints to determine optimal partitions for a given workload. According to our experiments, the schemas recommended by AutoPart (a) improve query execution by 12%-44% when compared to the original schema without indexes, (b) execute queries up to 37% faster and updates almost twice as fast compared to the original after indexing. Furthermore, they require half the space for indexes. Finally, we show that a form of categorical partitioning can further improve query performance up to 29%-62% without any indexes and up to 54% with indexes, while update performance improves up to 60%. AutoPart can be used with any commercial database system. Our experimental results are based on the Sloan Digital Sky Survey (SDSS) database, a real-world astronomical database, running on Microsoft's SQL Server 2000.

# 1 Introduction

Scientific experiments in fields such as astronomy and biology typically require accumulating, storing, and processing very large amounts of information [9]. The ongoing effort to support the Sloan Digital Sky Survey (SDSS) [8][17] provides a comprehensive example for both the terabyte-scale storage requirements and the complex workloads that will execute on future database systems. Similarly, the Large-aperture Synoptic Survey Telescope (LSST) [18] dataset is expected to be in the scale of petabytes (the data accumulation rate is calculated at 8 terabytes per night). Typical processing requirements on these datasets include decision-support queries, spatial or temporal joins, and versioning. The combination of massive datasets and demanding workloads stress every aspect of traditional query processing.

In environments of such scale, query execution performance heavily depends on the indexes and materialized views used in the underlying physical design. The database community has recently focused on tools that utilize workload information to automatically design indexes [1][12]. Currently, all major commercial systems ship with design tools that identify access patterns in the input workload and propose an efficient mix of indexes and materialized views to speed up query execution. Typically, the tools tend to generate a set of "covering" indexes per query to enable index-only query processing. In the case of large-scale applications like SDSS, performance depends upon a large set of covering indexes, since accessing the large base tables (even through a non-clustered index) is prohibitively expensive.

Large numbers of covering indexes are expensive to store and maintain, as data columns from the base table are replicated multiple times in the index set. Adding multiple indexes to multi-terabyte scientific databases typically increases the database size by a factor of two or three, and incurs a significant storage management overhead. In addition, indexing complicates insertions and updates, as all "replicated" new and updated data values must be sorted and written multiple times for all the indexes. The cost of inserting and updating data increases as a function of the number of tuples that are inserted or modified. The index tools currently available do not take into account insertion or update costs.

We propose to design a workload-aware schema before designing indexes, by decoupling data partitioning from ordering. By first designing a partitioned schema and then building indexes on the new database, queries can scan the base tables efficiently as well as a smaller set of indexes, thereby alleviating unnecessary storage and update statement overhead. Because vertical partitioning increases spatial locality, it improves memory and disk system performance when the most effective index set cannot be built because of storage or update constraints. If update or storage constraints do not exist, then the workload can always be processed using a complete set of covering indexes. Such a scenario, however, is unrealistic for large scale scientific databases, where both insertion and storage management costs are seriously considered.

This paper describes AutoPart, an automated tool that partitions the tables in the original database according to a representative workload. The contributions of this paper are the following:

- We develop three vertical partitioning algorithms for schema design. We consider greedy and randomized enumeration.
- We experimentally evaluate vertical partitioning on the SDSS database and workload. Our experiments (i) compare our vertical partitioning algorithms (ii) evaluate the benefits of vertical partitioning in a complete design, including indexes and an update workload (iii) evaluate a horizontal partitioning scheme called categorical partitioning, which is applied on the SDSS data.
- We incorporate our vertical partitioning approach in *AutoPart*, an automated schema design tool, that can interface to database systems supporting JDBC.

Experiments show that, when using only vertical partitioning, the new schema (a) speeds up query execution by 12%-44% when compared to the original (unindexed) schema, (b) executes updates almost twice as fast compared to an unpartitioned but heavily indexed schema. Furthermore, indexing the new schema requires half the space and improves query performance by up to 34%. Finally, categorical partitioning improves query performance up to 29%-62% without any indexes and up to 54% with indexes. In the latter case, the update performance speedup reaches 60%.

This paper is structured as follows: Section 2 summarizes related work. Section 3 discusses the vertical partitioning problem in greater detail, while in Section 4 we present the algorithms used in this study. Sections 5 and 6 discuss the AutoPart architecture and our experimental setup. Section 7 shows the results of our experiments with partitioning and Section 8 concludes our study.

# 2 Related work

Vertical partitioning is known to optimize I/O performance since the early days of relational databases. The Decomposition Storage Model (DSM) [5] first replaced the original relations by single-attribute fragments, and constructed an index on each fragment independently from the workload. DSM penalizes queries that use a large fraction of the relation's

attributes with extra joins. Fractured mirrors [15] remedy DSM performance by (a) using thick tuples to reduce the cost of DSM joins and (b) storing both the partitioned and the non-partitioned versions of the database and combining them during query optimization and execution. Our work aims at performing workload-conscious vertical partitioning on the initial database while keeping one copy of the database around, and can be combined with mirroring for even better performance.

To reduce the number of joins DSM imposes on multi-attribute queries, several studies [11][13][14] exploit *affinity* within a set of attributes (a measure of how often queries use attributes together in a representative workload). Combined with a clustering algorithm, affinity determines a reasonable assignment of attributes to vertical fragments. Attribute affinity identifies clusters by collecting statistics about the attribute usage by queries, and can therefore scale to large workloads. Its disadvantage is that it is decoupled from the system's optimizer and the query execution engine, and thus human intervention is eventually required to validate the quality of the recommended partitioned designs.

An extension to the previous approaches incorporates query processing using cost estimates given a table configuration [6]. The paper defines a set of analytical formulae that model vertical partitioning as an integer programming optimization problem. Modern practice, however, suggests that explicit analytical functions are of limited value, since they are rarely in accordance to the real cost models in modern query optimizers and cannot be easily applied on complex queries or on complex execution engines.

Similarly to today's tools for automatically evaluating database indexes, a software cost estimation module examines candidate configurations and computes their expected cost [10]. Candidate configurations are determined through a heuristic that iteratively combines attributes, minimizing the total workload cost at each step. Although the proposed scheme is simple and reduces total workload cost at each iteration, it does not incorporate workload-specific information such as the sets of attributes referenced by each query.

The method of choice for modern, state-of-the-art automatic design tools is the combination of heuristic search methods with the system's own query optimizer. Index selection tools for relational databases [1][2][12] and the automatic declustering techniques for parallel DBMS [16] are based on the optimizer's cost estimates. The index selection problem is closely related to vertical partitioning: since the base table structure is perceived as a static property of the database, a viable alternative for reducing the I/O requirements of a query workload is the use of covering, multi-attribute indices to facilitate index-only data access. Such indexes essentially are ordered vertical partitions. The only difference is that indexes are redundant structures, therefore a popular column is replicated multiple times in the final design. Given that the original relations are not necessarily optimized for a particular workload, these tools often face a difficult problem, since the use of every additional index increases update overhead and data redundancy.

## 3 Query-Based Vertical Partitioning

Vertical partitioning stores a relation as multiple fragments, each containing a subset of the initial relation's attributes. A well-designed partitioned schema improves performance, because queries access smaller tuples that contain a higher percentage of interesting data. Our approach relies on intelligently partitioning the database schema using workload information, to take advantage of the potential performance improvement. As a second step, we use existing tools to select appropriate indexes on the partitioned schema. The partitioned design requires less indexing overhead, resulting in higher update performance and less storage for indexes.

To achieve lossless decomposition, a unique identifier column is replicated in all the vertical fragments of a relation. The original relation can now be removed and replaced by its fragments. Whenever information needs to be reconstructed from multiple fragments, however, query plans must include additional joins. In order to avoid reconstruction costs, it is necessary to partition relations in a workload-conscious fashion.

The rest of this section discusses the issues involved in designing an automated partitioning tool, and explains the basic aspects of our approach. The issues are:

- Why not rely on simple partitioned designs that use the Decomposition Storage Model (DSM)?
- Which enumeration algorithms are needed to evaluate the large numbers of alternative partitioned designs?
- Which tables should we partition?
- How does vertical partitioning affect the creation of indexes, and how can we efficiently combine the two?

### 3.1 Using single-attribute relations

The Decomposition Storage Model (DSM) [5][15] creates single-attribute relations, so that a query can access only the attributes that it needs and no unnecessary attributes need ever be retrieved. In addition, it requires no prior analysis of the workload. It assumes that the system's query optimizer and query execution engine will cooperate to efficiently join mul-

tiple columns from the same original relation, if this is required by a query. Warehousing products such as SybaseIQ [21] successfully use DSM-style partitioning and further optimize data access using variant indexes [22]. DSM, however, is not always the best option in the context of standard relational systems.

In DSM, queries accessing large numbers of attributes suffer from extensive reconstruction costs. The problem persists even with more efficient storage/processing schemes [15] (assuming no replication). Therefore, it is difficult to execute queries that reference large numbers of attributes.

Our approach suggests a departure from this extreme design point by examining workload-aware partitioning algorithms that can manage sets of attributes, while being able to resort to single-attribute relations if appropriate.

### 3.2 Enumerating alternative designs

Enumerating all the alternative partitioned designs for a given schema is impractical. Even when partitioning a single table, the number of possible partitions is higher than exponential on the number of attributes in the table. In addition, optimizing even for a single query that accesses multiple tables might require an exhaustive search of all possible combinations. Assuming that the query plan sequentially scans tables T1 and T2, partitioning changes the relative costs of accessing the tables and may affect the entire execution plan. An optimization algorithm cannot rely on decomposing the design problem into sub-problems, for example partitioning first T1 and then T2, since deciding on a partitioning configuration for T1 restricts the set of possible plans when evaluating T2.

Reaching an optimal solution becomes even more difficult when considering multiple queries, where assigning an additional attribute to fragment $T_i$ of table T to optimize one query means removing it from another fragment $T_j$, potentially affecting the performance of another query. The results can be more dramatic if $T_i$ is broken into two fragments, in which case the plan is fundamentally restructured. Similar complexity appears for index selection: researchers suggest that one cannot optimally select indexes using a restricted local search (even for a single query), since the effectiveness of each new index depends on the set of the indexes that have already been chosen [1][2][7][12]. Furthermore, any local decision has an impact on global parameters, such as the overall storage and the performance of update statements. A proposed solution [1][2] is to exhaustively search all combinations for a very small number of indexes and then using a suboptimal, greedy selection strategy for the rest.

In order to deal with the above complexities, we also generate a restricted search space of alternative configurations and explore it in a greedy fashion, picking the lowest cost solution discovered so far. The enumeration algorithm proceeds from state to state by applying moves (such as splitting a table into two fragments) and evaluating the workload cost at each step. To more effectively explore the search space and evaluate the greedy approach, we also examine randomized search algorithms.

### 3.3 Selection of tables to partition

When designing a vertical partitioning algorithm, we need to decide which tables to partition. The obvious solution is to consider all the tables in the database. Tables that are not expensive to access, however, but are heavily referenced by the workload can generate large numbers of candidate designs without a significant performance improvement. Therefore, there is a trade-off between performance improvement and search algorithm complexity.

One way of identifying the tables to partition is to evaluate the workload cost (with the optimizer) once on the original schema to measure the contribution of accessing each table. This method has the disadvantage that the table access costs are computed based on the assumption that no indexes exist. The presence of an index might reduce the impact of accessing a table and change the relative rankings. A different solution is to design a partitioning algorithm that considers candidate partitions for all possible tables, only in its initial steps. Cost analysis during the algorithm's operation will reveal the candidates that have negligible contribution to the workload cost and prune them.

Our algorithms are designed to receive as input the tables that need to be partitioned. To make a selection, we analyzed the total cost of accessing each table, for our experimental workload and database. We found that our workload is dominated by the cost of accessing the two largest tables in the database. The access costs for the two largest tables are 47% and 7% of the overall workload cost, while the rest of the tables account for less than 1%.

### 3.4 Integration with indexes

As index selection is an important aspect of physical design, we must determine whether partitioning can reduce the need for indexing, thereby alleviating the index maintenance or storage overhead. For this purpose, the search algorithm should also evaluate configurations consisting of alternatives for both tables and indexes. A combined search space is useful, for example, because partitioning might disable an actually useful index. In addition, including an index at some state of a partitioning algorithm might change the relative costs for the workload queries and lead to a new set of candidate config-

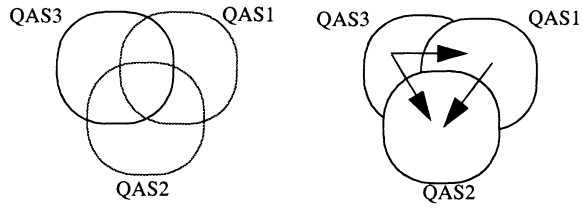


**FIGURE 1:** *Overlapping QAS (left) and attribute assignment (right).*

urations. A similar argument is made in [1], about the need for the joint enumeration of candidate indexes and materialized views. The prohibitive factor in this case is the immense search space, since designing the indexes is already a complex problem.

In this paper we consider vertical partitioning and indexing as two different steps in a design sequence. In the first step we perform vertical partitioning assuming no indexes exist and in the second step we perform index selection, using existing automated tools. Our experimental results suggest that the partitioned schema can still support those indexes that are essential for performance (for example those used to speed up expensive joins). In addition, for the lower cost queries in the workload, many of the 'covering' indexes can be removed because of the performance benefits of partitioning.

## 4 Vertical partitioning algorithms

In this section we present the vertical partitioning approach used in this paper. Our algorithms are based on identifying a query's Query Access Set (QAS) which is the set of attributes of a relation that a query accesses. The basic assumption is that the cost of a query is minimized when there exists a *fragment* of the original relation that contains exactly the attributes in its QAS.

Specifically, given a database schema $W$, a table $R \in W$ and a set $Q_R$ of queries that involve attributes in $R$, we define the *Query Access Set* of query $q$ with respect to $R$ (QAS($q$,$R$)) as the subset of $R$'s attributes accessed by query $q$. Every QAS in the workload can be viewed as a distinct access pattern. We call the generation of a fragment $R_i$ that contains exactly the attributes in QAS($q$,$R$) a *materialization* of QAS($q$,$R$).

Ideally, for every two queries $q_i$ and $q_j$ *in* $Q_R$, QAS($q_i$,$R$)$\cap$QAS($q_j$,$R$)=$\emptyset$; therefore we can partition $R$ into fragments $R_i$, $i$=1,...,$|Q_R|$+1 such that (a) the union of all $R_i$ equals $R$, (b) no two fragments have attributes in common, and (c) $\forall i$, $q_i \in Q_R \Rightarrow R_i$=QAS($q_i$,$R$), i.e., each fragment is a materialization of a QAS (the last fragment contains $R$'s attributes not used in $Q_R$). Such a partitioning has the attractive properties that (a) no query will ever need to join two fragments to reconstruct its QAS, and (b) each query will use all data in pages of the fragment that contains its QAS.

Realistically, however, queries access overlapping attribute sets. In this case there are queries that will have to perform additional joins or access extraneous attributes. What is required is an algorithm that will balance the I/O and record reconstruction costs for the workload. Consider for example the situation with the overlapping QAS of three queries, shown in the left side of Figure 1. An exhaustive algorithm would evaluate all the ways of combining the QAS fragments shown. We design our search space based on the observation that performance benefits can be obtained by allowing a subset of the queries to access their entire QAS, materialized in a single fragment. The remaining attributes can be distributed among the other queries so that the sequential scan or joining costs are balanced.

Our algorithm works in principle by selecting a *winner* query at each step. The winner is allowed to materialize its QAS in a separate new fragment. The new fragment will contain all the attributes in the query's QAS except for those belonging to the QAS of queries that were winners in previous steps. A possible solution to the example three query problem, including the additional joins, is shown in the right side of Figure 1.

Section 4.1 describes the specifics of the search space we consider for the vertical partitioning problem. Section 4.2 describes our primary algorithm, PRIOQ-MERGE, which traverses the search space in a greedy fashion. Section 4.3 describes adaptations of randomized algorithms to traverse the search space.

### 4.1 Search space design

We consider a state in the physical design algorithm to be a partially partitioned schema. The initial state corresponds to the original schema, where no queries have been considered for optimization yet, e.g. no QAS have been materialized. In an intermediate state, some QAS have already been selected and materialized, resulting in a set $P$ of fragments. The attributes that do not belong to a materialized QAS remain merged in a large table $T$. Thus a state for each partitioned table consists of the following:

- a set $P$ of already selected fragments.

- the table $T$ containing the remaining attributes.
- the QAS already materialized.
- the QAS that are yet to be selected.

In the final state of the algorithm all the fragments have been computed, all the QAS have been considered and $T$ is empty, unless there are attributes not used by any query in the workload.

## 4.2 Enumeration: PRIOQ-MERGE

We define one basic move to transition from one state to the next. SPLIT is a transformation that generates a new fragment containing the attributes referenced by a QAS that is not yet resolved. The new fragment contains the attributes that do not yet exist in any of the fragments generated in a previous move (do not exist in $P$). The attributes are removed from $T$. Intuitively, SPLIT 'chunks' a set of attributes from the original relation and stores them in a separate fragment.

The enumeration algorithm tries all possible SPLIT moves accessible from a given state and it evaluates the workload performance for each of them. The winner state is the one that has the lowest overall workload cost. The new fragment is added to $P$. Note that the winner QAS might benefit multiple queries in the workload. In this case, the QAS 'covered' by the winner QAS (those that no longer reference any attributes in $T$) no longer need to be considered. Selecting a winner at each step greedily enforces a *PRIO*ritization among the *Q*ueries, since those that are selected first benefit most.

The problem with the basic SPLIT mechanism is that it will potentially consider fragments that contain a very small number of attributes. 'Thin' fragments can be 'attached' to existing fragments in $P$, reducing the number of joins without significantly increasing the I/O cost of accessing the merged fragment. Thus, we extend our enumeration algorithm so that it considers *MERG*ing the new fragment with one of the existing fragments.

```
Greedy (workload W, relations R)
  Determine distinct QAS in workload W
  S0 := initial state //original tables
  S := S0
  while (cost cannot be further improved and
         exist QAS that are not considered)
    Neighbors := ∅
    For each QAS not yet considered
      Neighbors := Neighbors ∪ SPLIT (S,QAS,R)
      For each fragment F in state S
        Neighbors:= Neighbors ∪ MERGE(S,QAS,R,F)
        Choose min cost state Smin in Neighbors
    S := Smin //Materialize QAS selected for this state
  Return overall minimum cost S
```

**FIGURE 2:** *The PRIOQ-MERGE (Greedy) Algorithm.*

The PRIOQ-MERGE algorithm is shown in Figure 2. The Neighbors list is a list of states derived from a state S. It is populated by applying the SPLIT and MERGE procedures. The SPLIT procedure takes as input the starting state and a candidate QAS from relation R. It generates a new state, where a new fragment has been created. The MERGE procedure generates a new state where a new fragment is merged with one of the existing fragments in $P$. Once all the "neighboring" states have been generated, the algorithm selects the once that has the lowest cost.

## 4.3 Enumeration: Randomized Algorithms

The purpose of employing randomized algorithms is to explore the search space described in Section 4.1 without necessarily choosing the lowest cost solution at each step. The basic idea behind randomized enumeration is to allow more states to be evaluated, even those that do not necessarily belong to a lowest cost path, because they could also lead to low cost solutions.

Randomized algorithms have been used in the database literature [19] to address complex optimization problems, especially those involving query optimization for large numbers of joins. In this case a randomized search is necessary to deal with the large space of possible join orders.

In this study we consider two randomized search algorithms, Iterative Improvement and Simulated Annealing. Iterative Improvement selects the next state at random, provided that it provides a lower cost solution (not necessarily the lowest). The innermost loop stops when a local optimum has been reached and the quality of the solution cannot be further improved. The random traversal can be repeated multiple times and the algorithm maintains a list of the lowest-cost solu-

```
Iterative Improvement (workload W, relations R)
  Determine distinct QAS in workload W
  S0 := initial state
  while (time limit has not been reached)
    S:=S0
    while (cost cannot be further improved and exist
          QAS that are not considered)
      generate Neighbors list  using SPLIT, MERGE
      S1:= random state ∈ Neighbors
      If  (cost(S1) < cost (S))
        S:= S1
      Save minimum cost seen so far
  Return overall minimum cost
```

**FIGURE 3:** *The Iterative Improvement Algorithm*

```
Simulated Annealing (workload W, relations R)
  Determine distinct QAS in workload W
  S0 := initial state
  Temperature T := T0
  While (T above  threshold) // system not 'frozen'
    S:=S0
    While (cost cannot be further improved AND
          exist QAS that are not considered)
      generate Neighbors list  using SPLIT, MERGE
      S1:= random state ∈ Neighbors
  If  (cost(S1) < cost (S))
        S:= S1
      Else    //Accept up-hill moves with a probability
        S:= S1 with probability inversely proportional to T
      Save minimum cost seen so far
    Reduce temperature T // reduce system temperature
  Return overall minimum cost
```

**FIGURE 4:** *The Simulated Annealing Algorithm*

tions obtained so far. The outermost loop stops whenever a time limit has been reached at which point the best solution found so far is returned to the user. The pseudocode for our Iterative Improvement algorithm is shown in Figure 3.

Simulated Annealing works in a similar way, only that it initially allows moves that actually increase the overall cost. The motivation for this is that a state that might appear to have higher cost, might eventually lead to lower cost solutions. This property is useful, in cases where the search space has many local minima that must be avoided, in order for the global minimum to be reached. The algorithm is designed so that more up-hill moves are allowed at the initial steps. As the algorithm progresses, it tends to select only down-hill solutions. The algorithm decides whether to accept a random up-hill move based on a probability.

This probability is determined by a value called the 'temperature'. As the algorithm progresses, the temperature is reduced, so that it is harder for up-hill solutions to be selected. The 'temperature' analogy is borrows from natural processes involved in crystal formation, where the temperature of those systems is reduced, until they reach equilibrium.

Our adaptation of simulated annealing for the vertical partitioning problem is shown in Figure 4.

## 5 System architecture

This section describes the functional blocks of the automated schema partitioning tool, which is depicted in Figure . The system implementation was done using Java (JDK 1.4) and JDBC and the DBMS is SQL Server 2000.

***QUERY PARSER.*** This module receives as input the original queries ($Q$) and the tables to partition ($\{R\}$). Its output is the queries in a parsed representation ($Q_P$)

***TABLE DESIGNER.***The Table Designer module is the heart of the schema design tool. It receives as input the set of parsed queries ($Q_P$) and the original schema definition ($W_{ORIG}$), and applies the vertical partitioning algorithms of Section 4. Its output is a set of candidate partitioned schemas ($\{W_{PART}\}$ ) to be evaluated by the query optimizer.

***QUERY REWRITER.*** The rewriter uses each partitioned schema definition ($W_{PART}$) and the set of parsed queries ($Q_P$) to produce a set of equivalent rewritten queries ($Q_R$) that can access the fragments in $W_{PART}$.

***DBMS INTERFACE.*** This is a JDBC interface to the database currently hosted by the SQL Server. The interface executes table and statistics creation statements according *to* $W_{PART}$. To accurately estimate query costs, our tool provides the query optimizer with the correct table sizes and statistics for the partitioned schema. Since it is impractical to populate the tables for each candidate schema, we estimate table sizes and copy the estimates to the appropriate system catalog tables, for the optimizer to access. In addition, we compute statistics for each column in the original, unpartitioned tables and reuse that information for the evaluated partitions. To test our *virtual* table generation method, we actually implement the partitions recommended by our tool and find that the cost estimates obtained by it match those obtained from the real database.

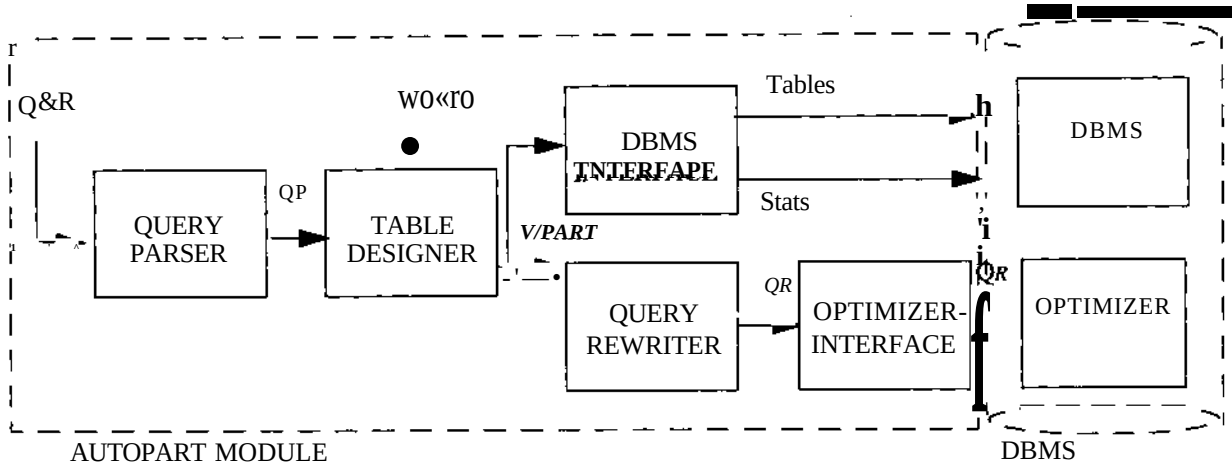


FIGURE 5: *AutoPart Architecture.*

We found that in order for the virtual and real cost estimates to agree, the statistics must be generated using full data scans and not by random sampling.

*SYSTEM CATALOG* The DBMS system catalog stores nformation like table sizes, row sizes and statistics. To facilitate query cost estimation, we update the system catalog tables with information reflecting the new schemas.

***OPTIMIZER INTERFACE.*** This JDBC interface receives as input the rewritten queries *($Q_R$)* and uses the query optimizer to obtain query plan information and cost estimates.

We deployed our partitioning tool as a web application that runs independently of the database server component. We provide the input (query workload and tables to be partitioned) through a simple web interface. Our tool can (through standard JDBC) access remote databases to obtain the original schemas, modify their structure and obtain cost estimates for alternative solutions.

## 6 Experimental setup

Our experiments use the Sloan Digital Sky Survey (SDSS) dataset [8][17]. The SDSS database consists of 39 tables. The database is structured around a central "catalog" table, *PHOTOOBJ* (22GB), which describes each astronomical object using 369 mostly numerical attributes. The second largest table is *NEIGHBORS* (5GB), which is used to store spatial relationships between neighboring objects. It essentially contains pairs of references to neighboring *PHOTOOBJ* objects and additional attributes, such as distance. Both tables are clustered on their primary key, which consists of application-specific object identifiers.

The SDSS workload consists of 35 SQL queries. Most of them are sequential scans that process *PHOTOOBJ* and apply predicates to identify collections of astronomical objects of interest. 6 queries (the most expensive ones) have a spatial flavor, joining *PHOTOOBJ* with *NEIGHBORS.* Only 68 of the 369 attributes in the *PHOTOOBJ* table and 5 out of the 8 attributes in *NEIGHBORS* are actually referenced in the workload.

Using the SDSS database, we perform the following experiments. First, we compare the performance of the 3 vertical partitioning algorithms introduced in section 4. Second, we compare the performance of an indexed design to that of a vertically partitioned and indexed design. We use the Microsoft's SQL Server 2000 Index Tuning Wizard (ITW) to derive the index configurations for the two schemas. We append a maintenance workload to the standard 35 SDSS queries and use the resulting mixed workload as input to the ITW. The purpose of using updates is to show the benefits of vertical partitioning when indexes become expensive. We compare the performance of both the standard and the update queries, and the storage consumed by indexes in each case.

Third, we show that additional performance benefits can be achieved by partitioning the *PHOTOOBJ* table into three disjoint segments using *categorical* attributes. The use of categorical partitioning is justified by the fact that most of the SDSS queries restrict the astronomical objects of interest to distinct classes, like galaxies or stars and do not need to access the entire *PHOTOOBJ* table. We demonstrate that categorical partitioning further improves the performance of the partitioned schema.

The following sections describe our experimental setup in more detail. Section 6.1 describes the update workload used to compare between the conventional and the partitioned designs. Section 6.2 describes categorical partitioning.

### 6.1 The update workload

The SDSS query workload includes only selection statements. However, to realistically evaluate competing designs in the presence of indexes, one needs to include a set of maintenance statements (insertions, updates) in the workload, to make the use of indexes more expensive.

The update workload (SDSS_U) used in our experiments consists of two insertion statements (SQL INSERT) and 10 update (SQL UPDATE) statements. The insertion statements simply append 800,000 and 5,000,000 tuples in the *PHOTOOBJ* and *NEIGHBORS* tables.

The update statements process tuples exclusively on the *PHOTOOBJ* table. To obtain them we modified existing SDSS queries, keeping their WHERE clause and modifying the SELECT clause, so that all the referenced attributes receive a zero value. The unique object identifier attribute for *PHOTOOBJ* is never modified, because modifications would require unnecessary (and costly) maintenance to the table's clustered index.

We used our rewriting tools to modify the update workload for use in the partitioned schema. Each insertion statement was replaced by a set of statements, each inserting new data to a fragment of the original table. In the partitioned case, the cost of inserting new data to a relation is equal to the sum of the insertion costs for each of its fragments. The update statements were similarly rewritten, replicating each initial statement when necessary to update tuples from multiple fragments.

### 6.2 Categorical partitioning

*Categorical partitioning* decomposes tables into horizontal fragments, containing disjoint sets of tuples. The partitioning is based on the value of one or more *categorical attributes* (i.e., attributes with limited value domains). The idea is to improve performance for queries that select tuples based on categorical attribute values. SDSS queries use two categorical attributes (*type* and *status*) to classify *PHOTOOBJ* tuples into Galaxies, Stars and Others. Most of the SDSS queries select either Galaxies or Stars, while a few access the entire table. Objects belonging to a particular class can be easily retrieved using indexes. However, making them explicit through categorical partitioning produces a schema with reduced access costs that potentially requires less indexing. Also, there is no overhead in recombining data from multiple partitions, since a union operation is required instead of a join.

We used categorical partitioning to improve the performance of the vertically partitioned schema. Each fragment of the *PHOTOOBJ* table is partitioned into Galaxies, Stars and Others. The queries (and the update statements) are then rewritten using union statements to combine the horizontal segments when necessary.

## 7 Experimental results

In this section we present experimental results on (a) the performance of three vertical partitioning algorithms, (b) the benefits of partitioning in the presence of indexes and maintenance workloads, and (c) the improvements obtained by the application of categorical partitioning..

### 7.1 Evaluation of Vertical Partitioning

This section aims at demonstrating how much vertical partitioning alone can improve workload performance (with no indexes or categorical partitioning), when compared to the original schema. In addition, we compare the traditional "greedy search" approach to the performance obtained when using randomized search algorithms. All the reported execution times are normalized cost estimates provided by the optimizer. The workload speedups shown are relative to a trivially partitioned initial SDSS database (ORIG), in which the tables only include the attributes actually used by queries.

Vertical partitioning improves the performance of queries that spend a significant fraction of their execution time scanning large tables. Depending on the dominant operator in their execution plan, SDSS queries are categorized into two groups. The first group, SDSS_J, consists of four queries, whose execution is bounded by expensive joins among several instances of *PHOTOOBJ* and *NEIGHBORS*. These queries account for 47% of the total workload cost. Table scans account for 13%-52% of the queries in the SDSS_J group. The second group, SDSS_S, includes 31 SDSS queries, primarily dominated by table scans. We expect vertical partitioning to significantly improve the performance of queries in the SDSS_S group

Although each algorithm enforces a different partitioned design, the greedy and randomized search algorithms perform similarly for SDSS queries. Table 1 shows average SDSS workload speedups when using the vertical partitioning algorithms described in Section 4. *PHOTOOBJ* was partitioned in 5, 4, and 8 fragments for the PQM_II, PQM_SA and PQM_Greedy algorithms respectively, whereas *NEIGHBORS* was not partitioned. Despite the differences, attribute groups that appear together in most of the queries are never separated in any of the partitioned schemas, but may be combined with other attributes to form fragments. Minor differences in the attribute placement amongst various designs lead

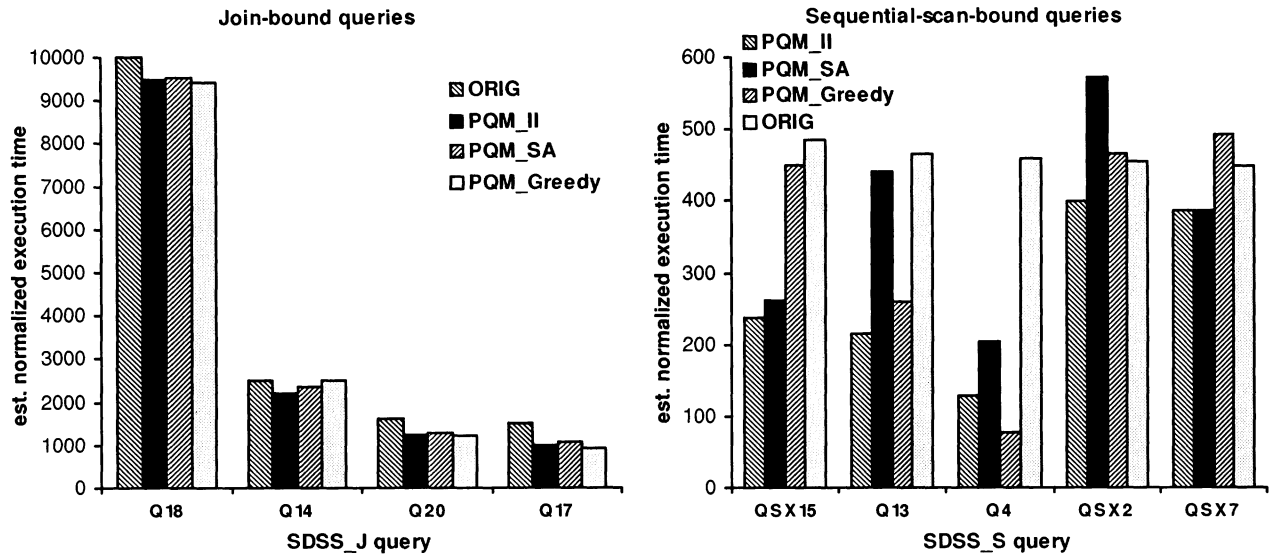


**FIGURE 6:** *Estimated query execution times when using the original and three vertically partitioned schemas.*

to differences in individual query costs, yet the overall performance is the same. Queries in the SDSS_J group are improved by 12%-14%, while queries in the SDSS_S class are improved by 37%-44% on the average. The overall workload speedups are 21%-25%.

Figure 6 shows normalized execution times for queries in the SDSS_J (left) and in the SDSS_S (right) groups. When compared to ORIG, query performance in the SDSS_J group, improves by a factor of 5% (Q18, PQM_SA) to 38% (Q17, PQM_Greedy). Execution performance for queries in the SDSS_S group, however, range from 11% to 83% (Q4, PQM_II). Note that the performance of individual queries varies depending on the partitioning each algorithm enforces. PQM_II, for instance, runs QSX15 47% faster than PQM_Greedy, because the latter happens to join 3 fragments, whereas the former accesses only a single fragment. Similarly, PQM_II runs Q13 51% faster than PQM_SA.

When designing a vertically partitioned schema to accommodate the needs of a large group of queries, the algorithms resolve trade-offs and recommend partitioning schemes that improve the workload performance as much as possible, occasionally incurring a low performance hit for low-cost queries. PQM_II does not incur a slowdown in any query, PQM_Greedy and PQM_SA only slow down QSX2 (that references 20 attributes) by a factor of 10% and 26% respectively.

***Table 1: Workload speedups for the SDSS_S and SDSS_J queries when using three vertical partitioning algorithms***

| VP algorithm | SDSS_J | SDSS_S | Overall |
|---|---|---|---|
| PQM_II | 14% | 44% | 25% |
| PQM_SA | 12% | 37% | 21% |
| PQM_Greedy | 13% | 43% | 24% |

## 7.2 Indexing a Vertically Partitioned Schema

This section evaluates the performance improvement when creating indexes on the original, non-partitioned schema (ORIG), and on the vertically-partitioned schema created using PQM_Greedy (VP). The indexed schemas are denoted as I_ORIG and I_VP, respectively. We designed indexes using the ITW with a 200GB storage constraint (10 times the size of the dataset) so index storage was practically unlimited. In addition to the 35 read-only SDSS queries, the workload includes 12 insert/update queries that represent database maintenance operations (SDSS_U). The objective is to evaluate the vertically partitioned schema and the indexes on it when the workload includes update statements.

Figure 7 (next page) shows the total workload cost when using the I_ORIG and I_VP schemas, for all the statement groups (SDSS_J, SDSS_S, and SDSS_U). When using the I_VP schema read-only statements run 17.2% faster, whereas updates run almost twice as fast. Overall, I_VP improves the workload execution time by a factor of 30%.

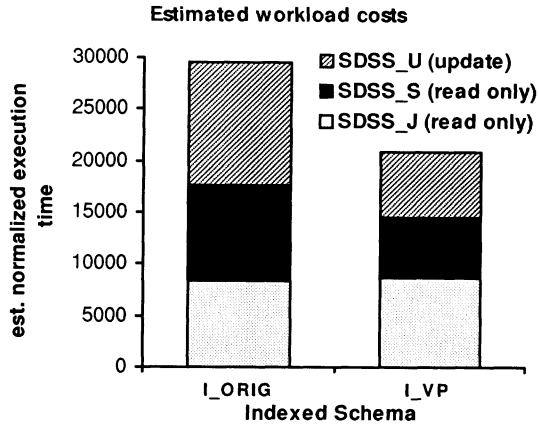


**FIGURE 7:** *Query and update workload costs using the original and the vertically partitioned indexed schemas*

Figure 8 shows execution time estimates for representative queries in the SDSS workload. The leftmost graph shows the four queries in the SDSS_J group. When compared to I_ORIG, I_VP runs queries Q17 and Q20 2% and 20% faster respectively, while it slows down Q14 and Q18 by 2% and 7% respectively. Q14 has a higher cost in the partitioned schema, because it references two fragments that need to be joined. This can be solved with a materialized view (i.e., more redundancy). Although the I_ORIG and I_VP query plans have identical structure, the optimizer proposes a different index on *NEIGHBORS* when using I_VP. This forces an additional hash-aggregate operator to implement *distinct* in the query's select clause, hence the additional 7%.

The rightmost graph in Figure 8 shows execution times for five (out of 31) queries in the SDSS_S group. The largest speedup is 83% for query Q4, whereas queries QSX3 and QSX13 run 79% and 75% faster. The reason is that, in order to reduce update costs, the I_ORIG schema does not include the necessary indexes for these queries to run faster. Q12 and Q13 are the only queries I_VP slows down when compared to I_ORIG. Those queries can be executed efficiently using indexes in I_ORIG, while they must join multiple fragments in I_VP paying a high overhead. Again, a materialized view would trade off the slowdown with an update overhead. Despite the join overhead in the last two cases, the I_VP schema exhibits an average speedup of 37% for the SDSS_S query group.

Figure 9 (next page) shows the detailed costs for the update statements in the workload. The insertion into *NEIGHBORS* (NI) is three times more expensive in I_ORIG, because it updates two indexes on Neighbors, compared to one index for I_VP. The insertion into *PHOTOOBJ* (POI) is slightly more expensive in the I_VP schema because although the

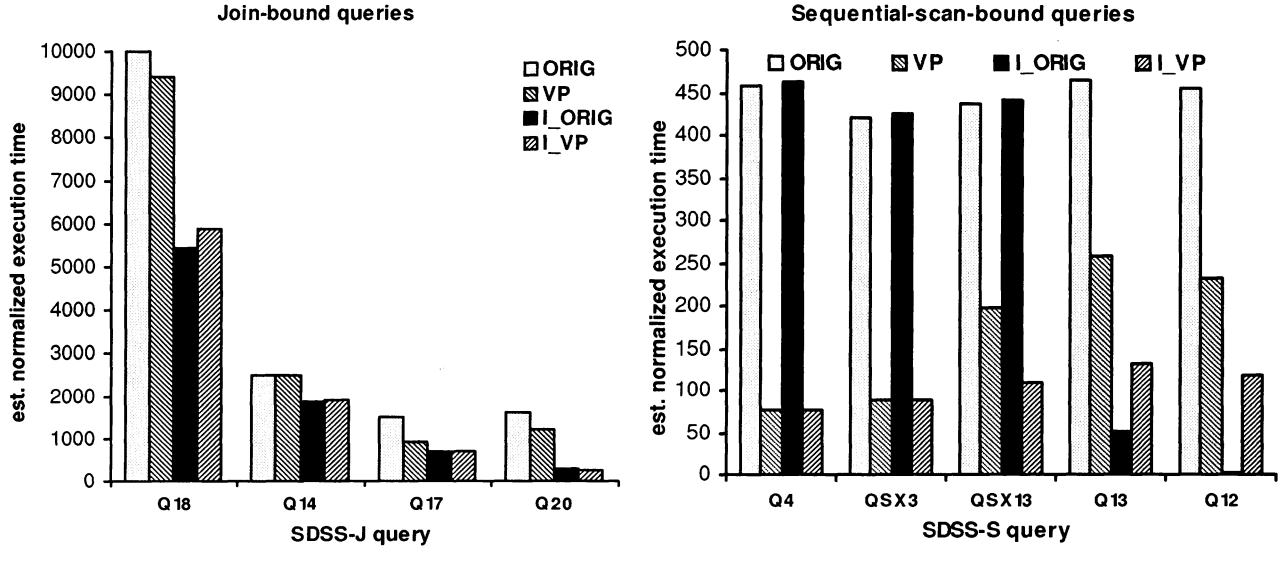


**FIGURE 8:** *Estimated query execution times when using the original and the indexed schemas.*

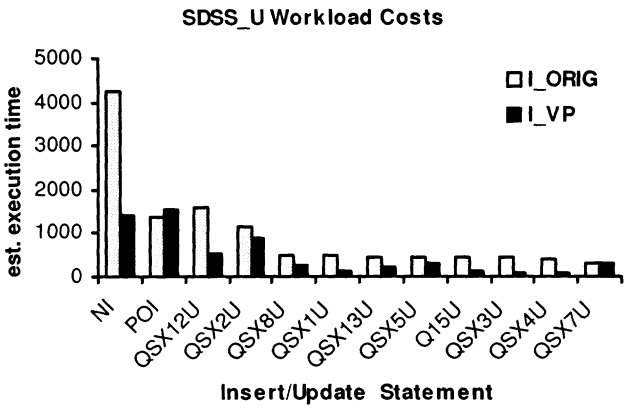


**FIGURE 9:** *Estimated update execution times using the original and the vertically partitioned indexed schemas.*

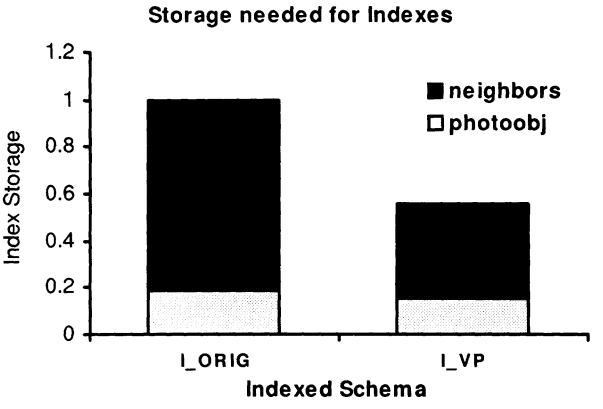


**FIGURE 10:** *Amount of storage needed for the indexes when using theoriginal and the vertically partitioned schemas.*

ndexes are the same, tuples are inserted in 8 different fragments. Query QSX12U is again three times more expensive in I_ORIG, for two reasons: First, its retrieval component (identifies the tuples to be updated) is about 4 times more expensive in I_ORIG, since the entire un-partitioned *PHOTOOBJ* table has to be scanned, while in I_VP only a fragment is scanned. Second, I_ORIG updates two indexes (as opposed to only one in I_VP), thereby slowing down the update process by a factor of three. The same holds for statements QSX1U to QSX4U which run from 30% to 80% faster in I_VP. On average, the SDSS_U queries run twice as fast using I_VP than when using I_ORIG.

Figure 10 compares the storage requirements for the two designs. The normalized values for the index storage are broken down into the components required to index the *PHOTOOBJ* and *NEIGHBORS* tables. The I_ORIG design requires almost twice the storage size for the indexes on *NEIGHBORS*, since the ITW recommends two indexes instead of one for the I_VP design. Overall, the I_VP schema requires 44% less storage space for indexes.

## 7.3 Evaluation of Categorical Partitioning

This section evaluates workload performance when using categorical partitioning on the already vertically partitioned schema. We decomposed every fragment of the *PHOTOOBJ* relation into three categorical segments: Galaxies, Stars and Others. Categorical partitioning reduces the cost of queries that access only a single type of *PHOTOOBJ* tuples. We used our automated tools to rewrite the queries and the update statements to match the new schema, as described in Section 6.2. The fully partitioned schema was indexed using the ITW. We expect the fully partitioned schema to perform better also after indexing, and to incur lower indexing storage costs.

Figure 11 (left) shows the estimated query execution times for five queries in the SDSS_S group when using the original (ORIG), vertically partitioned (VP) and the fully partitioned (VCP) schemas, without indexes. Q7 has the largest performance benefit from VCP, as its performance increases by 75% compared to VP and 91% compared to the original design. This happens because it only accesses a single segment in VCP. Queries QSX6 and QSX15 show how categorical partitioning can improve queries that do not benefit much from vertical partitioning. They exhibit 72% and 70% speedups compared to VP and 72% and 73% compared to ORIG. VCP has no effect on QSX3 and slows down Q15b by 8%, because these queries access all segments and Q15b joins multiple vertical fragments. Nevertheless, the average speedup of VCP is 36% for SDSS_S and 18% for SDSS_J (not in the graph) when compared to VP, and 62% and 29% when compared to ORIG. On average, categorical partitioning speeds up ORIG by 45% and VP by an additional 25%.

Figure 11 (right) shows the estimated query execution times for five queries in the SDSS_S group when using the indexed original (I_ORIG), vertically partitioned (I_VP) and fully partitioned (I_VCP) schemas. The largest speedup observed compared to I_VP is 79% for query QSX6, which has 84% better performance compared to I_ORIG. Queries QSX7, QSX8 exhibit similar behavior, being 66% and 68% faster compared to I_VP and 52% and 77% faster compared to I_ORIG. Most of the queries in the SDSS_S class exhibit similar behavior, because they access only one of the three categorical segments. Queries Q16 and QSX1 are exceptions, being 40% and 1.6 times slower in the I_VCP schema compared to I_VP. The behavior of Q16 is due to the index configuration in the two designs: Q16 does not utilize any indexes in I_VCP, while it efficiently uses an index in the I_VP design. QSX1 has lower performance because of the overhead involved in joining three horizontal and three vertical segments. Despite the exceptions, the overall speedup for the SDSS_S query group is 34% compared to I_VP and 54% compared to I_ORIG. For the four queries in the SDSS_J group,

the average speedup is 4% compared to I_VP (most of their cost is join overhead, and partitioning does not help). Overall, categorical partitioning provides a 28% improvement compared to I_ORIG and a 13% compared to I_VP.

c

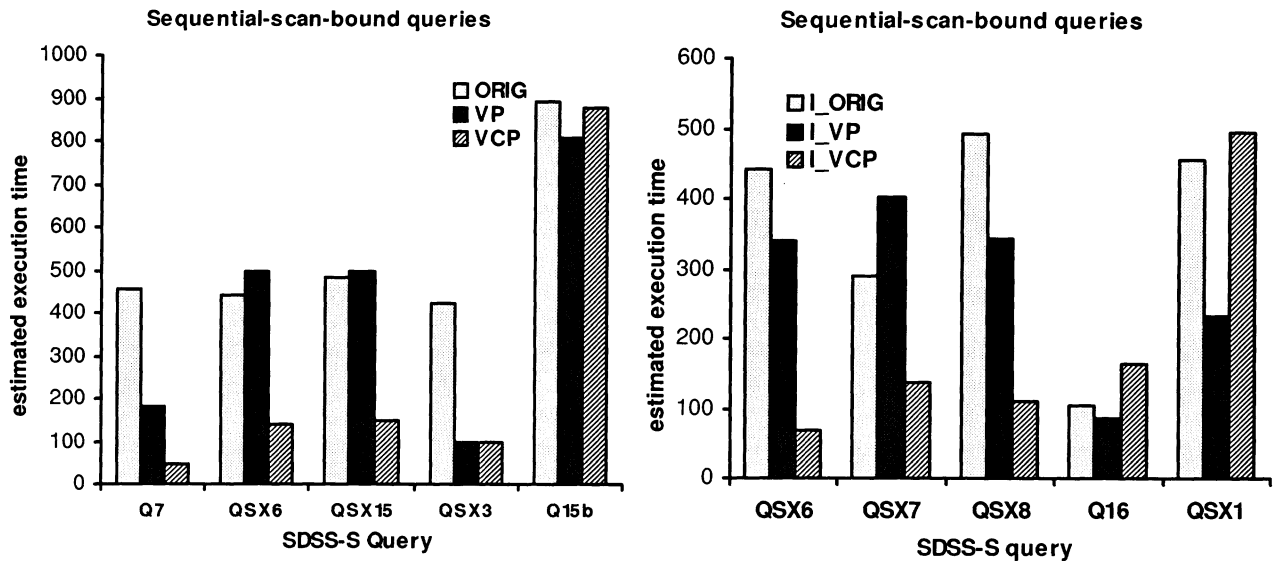


**FIGURE 11:** *Estimated execution times for queries in the SDSS_S class without indexes (left) and SDSS_S class after indexing (right) when using the original, vertically partitioned and fully partitioned schemas.*

Figure 12 shows execution times for the statements in the SDSS_U group when using the I_VCP schema. The highest improvement is observed for the *PHOTOOBJ* insertion statement (POI). In the I_VP design, 2 wide indexes on the most frequently accessed fragment have to be updated, while in the I_VCP case the insertion affects 5 thin indexes, built on multple horizontal parts. The resulting speedup is 25%. The I_VCP schema incurs a 3%-70% speedup to the rest of the queries, compared to I_VP and a 25% to 85% compared to I_ORIG, which is due to both more efficient update operations on the indexes and to more efficient searches for the updated tuples. The overall update workload speedup is 13% compared to I_VP and 60% compared to I_VCP.

## 8 Conclusions

This paper introduces AutoPart, a new tool for automated physical design that modifies the original database schema into a partitioned one according to a set of representative queries. AutoPart implements three vertical partitioning algorithms that use greedy and randomized enumeration of alternative schemas, as well as a categorical partitioning algorithm. AutoPart interfaces to a query optimizer to obtain cost estimates and evaluate alternative schemas. Our experiments with verti-

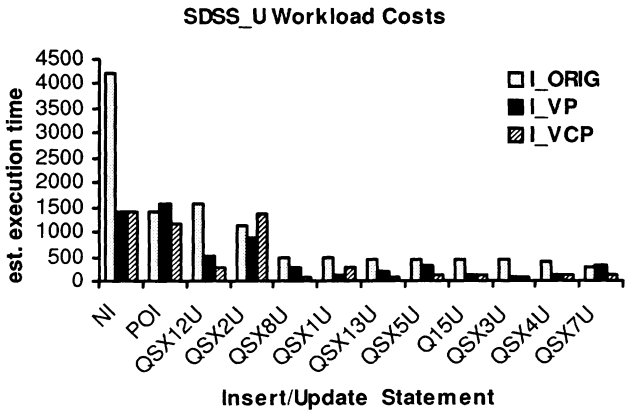


**FIGURE 12:** *Execution time estimates for insertions and updates using a vertically vs. a fully partitioned schema.*

cal partitioning on SDSS, a scientific astronomical database application, show that full (vertical and categorical) data partitioning improves query performance by up to 45% average (29% and 62% for the two query classes) no indexes. For indexed schemas, the respective query and update speedups are 28% and 60%. In conclusion, AutoPart is a versatile schema design tool that significantly improves workload performance and can be used with any commercial database system.

## References


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

[2] Chaudhuri S., Narasayya V., "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server". Proceedings of VLDB 1997.

[3] Chaunduri S., Gupta A., Narasayya V., "Compressing SQL Workloads". SIGMOD 2002.

[4] Chauduri S, Narasayya V., "Index Merging". Proceedings of ICDE 1999.

[5] G. P. Copeland and S. F. Khoshafian. A Decomposition Storage Model. SIGMOD 1985.

[6] Cornell D.W., Yu P.S., "A Vertical Partitioning Algorithm for Relational Databases", ICDE 1987.

[7] S. Finkelstein, et. al. "Physical Database Design for Relational Databases". TODS 13(1) (1988).

[8] "Data Mining the SDSS SkyServer Database," J. Gray, D.Slutz, A. Szalay, A. Thakar, P. Kuntz, C.Stoughton, MSR TR 2002-1, pp1-40, 2002.

[9] Jim Gray, Alex Szalay, "The World Wide Telescope: An Archetype for Online Science ," MSR TR 2002-75.

[10] Hammer M., Niamir B. "A Heuristic Approach to Attribute Partitioning". SIGMOD 1979.

[11] Hoffer J.A., Severance D.G., "The Use of Cluster Analysis in Physical Database Design". VLDB 1975.

[12] Lohman G., Valentin G., Zilio D., Zuliani M., Skelly A., "DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes". ICDE 2000.

[13] Navathe S., Ceri G., Wiederhold G., Dou. J, "Vertical Partitioning Algorithms for Database Systems. ACM TODS, 9(4), 680-710, 1984.

[14] Navathe S., Ra M., "Vertical Partitioning for Database Design: A Graphical Algorithm". SIGMOD 1989.

[15] Ravishankar Ramamurthy, David J. DeWitt, Qi Su. "A Case for Fractured Mirrors", VLDB 2002.

[16] Rao J., Zhang C., Lohman G., Megiddo N., "Automating Physical Database Design in a Parallel Database System". SIGMOD 2002.

[17] A. Szalay, J. Gray, A. Thakar, P. Kuntz, T. Malik, J. Raddick, C.Stoughton. J. Vandenberg, "The SDSS SkyServer – Public Access to the Sloan Digital Sky Server Data", SIGMOD 2002.

[18] http://www.lsst.org

[19] Y.Ioannidis, Y.Kang, "Randomized Algorithms for Optimizing Large Join Queries", SIGMOD 1990

[20] Chaco, software for partitioning graphs, http://www.cs.sandia.gov/~bahendr/chaco.html

[21] SybaseIQ, http://www.sybase.com/products/archivedproducts/sybaseiq

[22] P.O'Neil, D.Quass, "Improved Query Performance with Variant Indexes", Proceedings of SIGMOD 1997