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

Stratos Papadomanolakis
Carnegie Mellon University
stratos@cs.cmu.edu

Anastassia Ailamaki
Carnegie Mellon University
natassa@cmu.edu

Abstract

Database applications that use multi-terabyte datasets are becoming increasingly important for scientific fields such as astronomy and biology. Scientific databases are particularly suited for the application of automated physical design techniques, because of their data volume and the complexity of the scientific workloads. Current automated physical design tools focus on the selection of indexes and materialized views. In large-scale scientific databases, however, the data volume and the continuous insertion of new data allows for only limited indexes and materialized views. By contrast, data partitioning does not replicate data, thereby reducing space requirements and minimizing update overhead. In this paper we propose AutoPart, an algorithm that automatically partitions database tables to optimize sequential access assuming prior knowledge of a representative workload. The resulting schema is indexed using a fraction of the space required for indexing the original schema. To evaluate AutoPart, we build an automated schema design tool that interfaces to commercial database systems. We experiment with AutoPart in the context of the Sloan Digital Sky Survey database, a real-world astronomical database, running on SQL Server 2000. Our experiments corroborate the benefits of partitioning for large-scale systems: Partitioning alone improves query execution performance by a factor of two on average. Combined with indexes, the new schema also outperforms the indexed original schema by 20% (for queries) and a factor of five (for updates), while using only half the original index space.

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 (essentially, these indexes implement and ordered partition of the table). 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 non-clustered indexes) 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. For instance, new experimental or observation data are often inserted in the database and derived data are recalculated using new models. During update operations, all "replicated" new and updated data values must be sorted and written multiple times for all the indexes. Insertion and update costs increase as a function of the number of tuples inserted or modified. If update or storage constraints do not exist, 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. AutoPart receives as input a representative workload and designs a new schema using data partitioning. 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 data partitioning increases spatial locality, it improves memory and disk system performance when the covering index set cannot be built due to storage or update constraints. This paper makes the following contributions:

- We introduce AutoPart, a data partitioning algorithm. AutoPart receives as input a representative workload and utilizes categorical and vertical partitioning as well as selective column replication to design a new high-performance schema.
- To evaluate AutoPart we build an automated schema design tool that can interface to commercial systems and utilize cost estimates from the DBMS query optimizer.
- We experimentally evaluate AutoPart on the SDSS database and workload. Our experiments i) evaluate the performance improvements provided by partitioning alone, without the use of indexes and ii) quantify the performance benefits of partitioned schemas when indexes are introduced in the design.

Our experimental results confirm the benefits of partitioning: Even without the use of indexes, a partitioned schema can speed up query execution by almost a factor of two when compared to the original schema. Partitioning alone improves query execution performance by a factor of two on average. Combined with indexes, the new schema also outperforms the indexed original schema by 20% (for queries) and a factor of five (for updates), while using only half the original index space.

This paper is structured as follows: Section 2 summarizes related work. Section 3 discusses the partitioning problem in greater detail, while in Section 4 we present the AutoPart algorithm. Sections 5 and 6 discuss the AutoPart architecture and our experimental setup. Section 7 presents our experimental results and Section 8 our conclusions

2 Related work

In this section we present related work on physical design optimization. Published research is relevant to i) vertical partitioning of database relations, ii) index and materialized view selection and iii) support for partitioning.

2.1 Vertical Partitioning

Vertical partitioning is known to optimize I/O performance since the early days of relational databases. 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.

2.2 Index and Materialized view selection

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 are often face a difficult problem, since the use of every additional index increases update overhead and data redundancy.

2.3 Support for partitioning

There has been work on extending the relational engine to support some form of data partitioning. The most recent work Fractured Mirrors [15] is a storage scheme targeting the Decomposition Storage Model (DSM). DSM 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 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 ver-

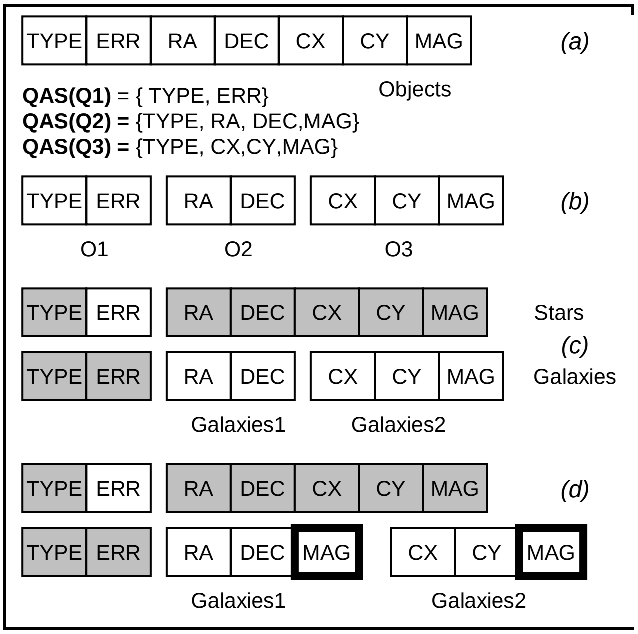

FIGURE 1: *Partitioning example. (a) original schema (b) vertical partitioning, (c) categorical (d) categorical with replication*

sions 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.

The approach taken by GMAP [23] is to decouple the logical structure of the database from the physical structures storing the data. A number of physical design alternatives are examined, including vertical partitioning of relations and replication of attributes. GMAP provides algorithms for the translation of queries expressed in terms of the logical schema so that they can efficiently access the underlying physical structures, or for updating the physical level data according to changes in the logical level. GMAP is complementary to our work: It provides a framework that can support modifications of the schema in the physical level. It does not cover the problem of *identifying* the optimal partitioning schemes for a given workload, rather it provides the primitives that allow a system to transparently support changes in its physical layer.

3 Workload-based data partitioning

In this section, we first briefly describe the vertical partitioning idea and the factors that limits its efficiency. We then explain how *categorical partitioning* and *replication* can alleviate the problem, using examples drawn from real scientific databases.

A general formulation of the vertical partitioning problem is the following: Given a set of relations $R = \{R_1, R_2, \ldots, R_n\}$ and a set of queries $Q = \{Q_1, Q_2, \ldots, Q_m\}$ determine a set of relations $R' \subseteq R$ to be partitioned and generate a set of fragments $F = \{F_1, F_2, \ldots, F_N\}$ such that:

1. Every fragment $F \in F$ stores a subset of the attributes of a relation $R \in R'$ plus an identifier column.
2. Each attribute of a relation $R \in R'$, is contained in exactly one fragment $F \in F$. (except for the primary key).
3. The sum of the query costs when executed on top of the partitioned schema, $\sum cost\ (Q, (R - R') \cup F)$ is minimized.

We expect the workload cost over the partitioned schema to be lower, because the fragments are faster to access than the original relations and queries will be able avoid accessing attributes they do not use. We define the *Query Access Set* (QAS) of a query Q with respect to a relation R (QAS(Q,*R*)), as the subset of R's attributes referenced by Q. In the ideal case, where for all query pairs Q_i, Q_j, $QAS(Q_i,R) \cap QAS(Q_j,R) = \emptyset$, the solution to the vertical partitioning problem would be to simply generate a fragment F for each distinct QAS in the workload. Then, each query would have to access a single fragment containing exactly the attributes it references, resulting in minimal I/O requirements. Realistically, however, the workload will contain overlapping QAS. In this case, such "clean" solutions to the vertical partitioning problem are not possible.

Consider the example in Figure 1(a), drawn from a simplified astronomical database. Our database consists of a single table (*Objects)* that stores astronomical objects (galaxies and stars). Our workload consists of queries Q1,Q2,Q3, shown in the figure with their QAS. Figure 1(b) shows one possible solution for vertically partitioning *Objects* into 3 fragments (O1,O2,O3). Q1 needs to access only O1, minimizing its I/O requirements. Since the attribute TYPE exists in all QAS, queries Q2 and Q3 will have to access fragment O1 in addition to O2 and O3 and perform the necessary joins. Also, since $QAS(Q2) \cap QAS(Q3) = \{MAG\}$ Q2 will have to access fragment O3 to obtain its missing attribute, performing an additional join. Alternatively, merging some of the O1,O2,O3 would result in lower joining overheads, but the queries would have to access a larger number of additional attributes and the I/O cost would increase.

The previous example demonstrates that overlapping QAS in a workload reduce the efficiency of vertical partitioning, because it is impossible to avoid additional joins for some of the queries in the workload. Often, however, much of the overlap implied by comparing the QAS is not real. Consider, for instance, that in the previous example Q1 restricts its search to objects of type "Stars", whereas Q2 and Q3 only care about objects of type "Galaxies". In this case, considering only QAS leads to "false sharing" as Q1 will process a completely disjoint set of tuples than Q2

[redacted]

amount of storage available for attribute
, which implicitly bounds the degree of replica-
ed. The output is a set F of fragments, which
the execution of Q. This section presents an
nd details of the interesting stages of the parti-
orithm.

minology

del, a relation R is represented by a set of
whereas a fragment F of R is represented by a
R. We distinguish between two kinds of frag-
mic fragments are the “thinnest” possible frag-
the partitioned relations, and are accessed
there are no queries that access only a subset
ic fragment. In addition, atomic fragments are
d their union is equal to R. A composite frag-
nstructed by the union of two or more atomic
The query extent of a fragment F is the set of
t reference it (if F is atomic) or the intersection
of queries that access each of its atomic compo-
is composite).

orithm overview

l structure of our algorithm is shown in Figure
step of the algorithm is to identify the categor-
ates in Q, and to partition the input relations
y to avoid the “false sharing” between queries
overlapping QAS but access different object
the second step, the algorithm generates an ini-
of the partitioned schema, consisting only of
gments of the partitioned relations. The perfor-
his initial version of the solution is determined
ng overhead (since atomic fragments may often
ingle attribute).

rformance of this initial schema is improved by
mposite fragments that reduce the joining over-
resulting schema but increase I/O cost: queries
a composite fragment don’t necessarily refer-
attributes in it. Composite fragments can either
eir constituent atomic fragments in the parti-
ema, or just be appended to the schema (assum-
eplication constraint is not violated). The
Fragment Generation module of our algorithm
a set of composite fragments that should be

[redacted]

and selection steps multiple times, each time
the fragments selected in the previous steps.
tion of fragments with an increasing number of
based on the results of previous iterations, is a
ristic, applied also in index/materialized view

```
/* schema PS is the best partial solution so far */
  1. schema PS := AF
  /*Composite fragment generation*/
  2. for each composite fragment F ∈ SF(k-1)
    2.a E(f) := {composite_fragments (F,A∈ AF) ∪
                 composite_fragments (F, A ∈ AF) having
                 query extent > X }
    2.b CF(k) := CF(k)∪ E(f)
  /*Composite fragment selection */
  3. for each composite fragment F ∈ CF(k)
    3.schema $S_F$ := add_fragment (F,PS)
    3.b if size($S_F$) > B then continue with the next F
    3.c compute cost($S_F$, $Q$ )
  4.select $F_{min}$ = arg_max (cost ($S_F$, $Q$ ))
          with cost ($S_{Fmin}$,$Q$ ) < cost (PS,$Q$ )
  5. if no solution was found then goto 9 /* exit */
  6. PS := $S_{Fmin}$
  7. SF(k) := SF(k) ∪ $F_{min}$
  8. remove $F_{min}$ from CF(k)
  9. repeat steps 3-8
  /* proceed with next iteration*/
  10. k++
  11. goto 2 /*generate new fragments */
```

FIGURE 3: *The AutoPart algorithm*

selection [1,2], to reduce the number of combination considered by the selection module. Note that the composite fragments considered may contain attributes that are also included in other fragments in the partitioned schema, thus allowing for attribute replication. When the workload cost cannot be further improved by the incorporation of composite fragments, the resulting schema is passed through a sequence of pair-wise merges of fragments, attempting to further improve performance.

The following sections present the various components of the algorithm in more detail. The pseudocode for the partitioning algorithm is shown in Figure 3.

4.3 Categorical Partitioning

The categorical partitioning step first generates horizontal fragments of the partitioned relations. The partitioning depends on the existence of categorical attributes in the relations and in the workload. Categorical attributes are attributes that take a small number of discrete values and are used to identify classes of objects. The basic motivation for categorical partitioning is that if queries operate on distinct classes of objects, those classes can be stored in separate horizontal fragments. The algorithm used for categorical partitioning of a relation R, under a query workload Q is shown in Figure 4. The algorithm first identifies the set of categorical attributes $\{A_i\}$ in R and their corresponding domains $\{D_i\}$ (step 1). This information can be provided either by the system's designer or the system catalog. Each query containing predicates on those attributes, defines a horizontal subset of R, containing all the objects that satisfy the predicates. The purpose of the algorithm is to determine a suitable collection of non-overlapping such fragments, which will be assigned to different horizontal fragments.For this, we use the methodology developed in [24]

```
categorical_partitioning(relation R, queries Q, size N)
  1. A := categorical attributes e R.
  2. Let X = collection of predicates in Q of the form
     $x_i$:{$A_i$ ∈ d ⊂ $D_i$}
  3. XM := complete_minimal_set({$x_i$})
  4. horizontal fragments Y := minterm_fragments(X)
  5. if |Y| ≤ N then
    5a. let A := A - {$a_i$} /* remove attribute ai */
    5b. Yi := merge(Y,$a_i$) /* merge the hor. fragments
```

We can express every query predicate involving each of those attributes in the form x_i: {Ai ∈ d ⊂ Di}. Let X = {xi} be the collection of such predicates, and assume that it is *minimal* and *complete*, according to [24]. Then, the min-term predicates Y(X) [24] computed in Step 4 define a collection of non-overlapping horizontal fragments that can be used to define the horizontal fragments of R. If there exist categorical attributes Ai that take a unique value in the horizontal fragments determined, they can be removed (Step 7).

Note that this collection of fragments can be modified, for example by suitably merging the horizontal fragments determined in step 4. Such a merging is shown in steps 5a-5b. The purpose of merging could be to derive a more suitable collection of horizontal fragments In Figure 4 we restrict the number of horizontal fragments generated to a number less than N. (This for example could express the user's desire not to over-partition the horizontal schema, in order to keep its definition manageable).

4.4 Composite fragment generation

The composite fragment generation stage provides, in each iteration, a new set of composite fragments to be considered for inclusion in the schema. It is described by step 2 in Figure 3.

The input to the stage for iteration k is the set of composite fragments SF(k-1) that were actually selected in the previous iteration. For the first iteration (k=1) the input to the stage is the set AF of atomic fragments.

As explained in section 4.1, the algorithm reduces the total number of composite fragments evaluated for inclusion by essentially extending only those fragments that were selected in the previous iteration. Those fragments can be extended in two ways:

1. By combining them with fragments in AF.
2. By combining them with fragments in SF(k-1)

The number of fragments generated in the initial steps of the algorithm is in the worst case quadratic to the number of atomic fragments. Depending on the size of the AF set, this number could be very large. It is possible to reduce the number of fragments generated, by selecting only those that will have the largest impact in the workload. Intuitively, a composite fragment is useful if it is referenced by many queries. The query extent of a fragment is a measure of a fragment's importance. Step 2.a prunes the fragments that are referenced by less than X queries in the workload. Pruning based on the query extent criterion reduces the set of fragments considered during the initial steps of the algorithm.

4.5 Greedy fragment selection

Given the collection of composite fragments provided by the generation stage, the selection stage greedily picks a subset of those for inclusion in the partitioned schema. The selection stage is described by steps 3-8 in Figure 3.

For each iteration, the selection module starts with the best "partial" schema found so far, PS, and a set of composite fragments CF(k) that must be evaluated for inclusion in the schema (step 3). The algorithm incorporates each candidate fragment in the current partial solution PS and computes the workload cost on the resulting schema (Steps 3.a, 3.b, 3.c). The fragment that minimizes workload cost is selected and permanently added to PS (Steps 6-8). The procedure is repeated until the workload cost cannot be further improved by fragments in CF(k).

The function add_fragment (Figure 5, used in step 3.a) removes all the subsets of a new fragment before adding it to the schema. This "recycling" of fragments simplifies the management of the storage space during the execution of the algorithm. If we were simply appending the new fragments to the partial solution, then the algorithm would quickly run out of space and then a separate process for removing fragments would have to be used. Using this replacement strategy , our algorithm works naturally when no replication is allowed in the partitioned schema.

Cost evaluation: Cost models

The selection module makes decisions based on the workload cost. We implemented AutoPart to utilize both a simple analytical cost model and the detailed cost estimation provided by the query optimizer of database systems. A simple model for the cost of a query on a partitioned schema is presented in Figure 6. The model captures only the parameters necessary for partitioning, like the I/O cost of scanning a table and the cost of joining two or more fragments to reconstruct a portion of the original data. In our model the I/O cost of scanning a fragment F is proportional to the number of its attributes (Step 4.a), since the number of rows in the fragments of the same relation is constant. The scaling factor S_R accounts for differences in relation sizes. The cost of joining two fragments is for simplicity considered constant and equal to J. The value of J must be carefully chosen to reflect the relative cost of joining compared to performing I/O. We computed the value of J by observing the query plans generated by the query optimizer, for various partitioned schemas.Our experimentation suggests that a value for J between 5 and 10 gives good approximations of the workload cost.

```
procedure add_fragment (schema S, fragment F)
  for each fragment $F_1 \in S$
    if ($F_1 \subset F$) then remove $F_1$ from S
      S := S $\cup$ { F}
    return S
```

FIGURE 5: *Procedure to add new fragments*

```
procedure cost (workload Q, schema S)
  1.repeat for each query q in Q
  2. Let R = a relation referenced in q
  /* compute the set of partitions Q will access */
  3. P := plan (Q,S)
  4. for each fragment f in P
    4.a scan_cost := scan_cost + $S_R$ * | F|
  5. join_cost = (|P|-1) * J
  6. $cost_R$ := scan_cost + join_cost
  7. repeat steps 2-6 for every relation referenced in q
  8. $cost_Q$ := sum of $cost_R$ + count(R) * J'
  9. return the total cost for all queries
```

FIGURE 6: *The model used for cost estimation*

An alternative to analytical models is the system's query optimizer. Modern optimizers utilize detailed knowledge of the query execution engine internals and of the data distributions to provide realistic cost estimates. The use of the query optimizer accounts for all the factors involved in query execution that our simple model ignores, like those affecting the joining costs. The use of the optimizer removes the constant join cost assumption of our model and takes into account factors like the existence of different join algorithms and the influence of predicate selectivities.The main disadvantage of using the query optimizer compared to an analytical cost model is that a call to the optimizer is time consuming.

4.6 Pairwise merging

The final part of the algorithm (Figure 7) is intended to

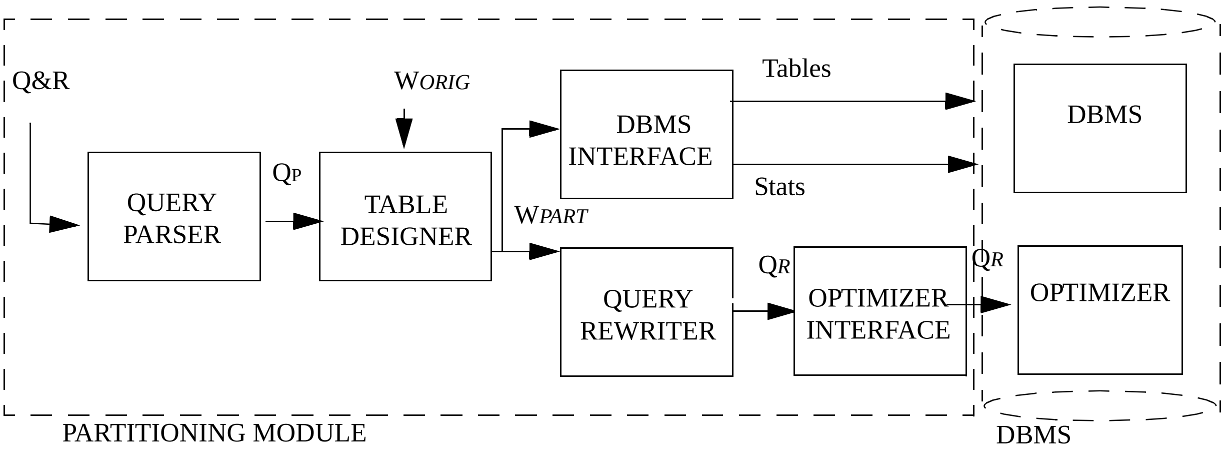

FIGURE 8: *Partitioning tool architecture.*

improve the solution obtained by the greedy fragment selection through a process of pairwise merges. The algorithm merges pairs of fragments from the solution obtained so far and evaluates the impact of the merge on the workload. Merges that improve workload cost are incorporated in the solution. The loop in steps 1-5 terminates when the solution cannot be further improved. Note that merging does not increase the size of the solution. We use the pair-wise merging process to capture the most important of those composite fragments that were not considered by the algorithm, because they were omitted by the fragment selection process.

```
procedure pairwise (queries Q, schema S)
    1. for each pair $F_i$, $F_j$ in S
        1.a $F_{ij}$ := merge ($F_i$,$F_j$)
        1.b $S_{ij}$ := S - $F_i$ -$F_j$
        1.c $S_{ij}$:= $S_{ij} \cup F_{ij}$
    2. find $i_{min}$, $j_{min}$ such that cost(Q,$S_{ij}$) is the minimal
    3. if cost not improved then goto 6 /*exit*/
    4. S := S $i_{min}$, $j_{min}$
    5. repeat steps 1-5
    6. return S
```

FIGURE 7: *The pairwise merging procedure*

5 System Architecture

This section describes the functional blocks of the automated schema partitioning tool, depicted in Figure 8. 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.

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 catalog stores information 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) database [8][17], running on SQL Server 2000. 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 from 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. For a fair comparison, we modified the database tables before our experiments, so that they only contain the attributes actually referenced in the workload.

To realistically evaluate the full impact of data partitioning one needs to include maintenance operations in the workload. The update workload (SDSS_U) used in our experiments consists of two insertion statements (SQL INSERT), that simulate the insertion of new data in the system's two largest tables. The statements we use simply append 800,000 and 5,000,000 tuples in the *PHOTOOBJ* and *NEIGHBORS* tables respectively, corresponding to 6% and 4.5% of their current contents.

The SDSS database comprises 39 tables. We used our partitioning algorithm to partition the two largest ones, *PHOTOOBJ* and *NEIGHBORS*, that are almost exclusively responsible for the workload's I/O costs. We present our performance results in terms of the *estimated execution time* provided by the query optimizer. The speedup of a query is defined as

s = 1 - (query_cost_optimized)/ (query_cost_original)

7 Experimental results

In this section we present experimental results on (a) the performance of our data partitioning algorithm and (b) the benefits of partitioning in the presence of indexes and maintenance workloads.

7.1 Evaluation of partitioning

This section demonstrates that the combination of categorical partitioning and attribute replication can generate schemas that can significantly improve query execution, even without the use of any indexes. We derive two partitioned schemas, CVP_x0 and CVP_x0.5 through categorical and vertical partitioning, without and with replication respectively. In the attribute replication case, we set a storage upper bound for the replication columns equal to 1/2 the original database size.

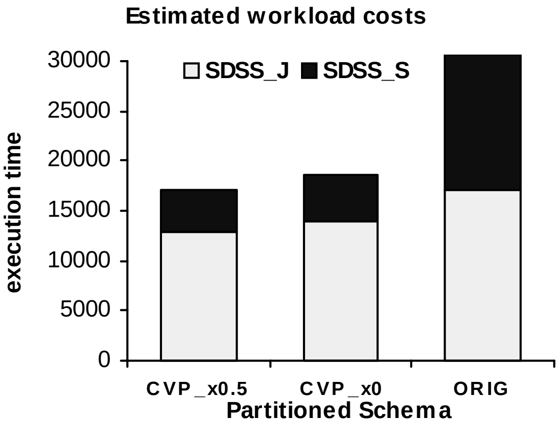

FIGURE 9: *Comparison of workload costs for the two partitioned schemas and the original, for two query classes*

The 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. The second group, SDSS_S, includes 31 SDSS queries, dominated by table scans. Queries in the SDSS_J group do not benefit much from partitioning, since the joins are their dominant operators. On the other hand, we expect vertical partitioning to significantly improve performance of queries in the SDSS_S group.

Figure 9 shows the estimated workload performance distinguishing and the two query classes, SDSS_S and SDSS_J. As expected the replicated schema (CVP_x0.5) performs better than the one without replication. (CVP_x0) The overall performance improvement is 47% and 43% respectively. Queries in the SDSS_J class benefit less, 19% and 24% respectively, while the improvements for the SDSS_S class queries are 69% and 72%. We observe that attribute replication after partitioning did not make significant difference in the overall execution time (8%).

Figure 10 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 from 2% (Q17, CVP_x0) to 56% (Q17, CVP_x0.5). The performance improvement for queries in the SDSS_S group is often impressive (an order of magnitude for Q7).

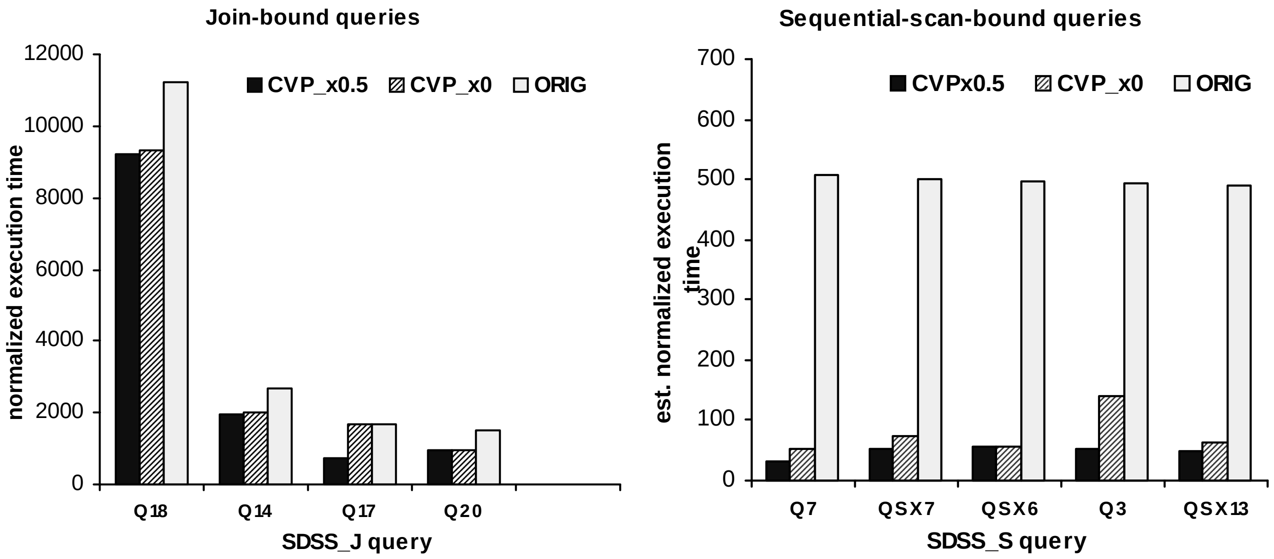

FIGURE 10: *Query execution times when using the original and the partitioned schemas.*

Partitioning improved the performance of all the SDSS queries, by significantly reducing their I/O costs. The schema with attribute replication offers better performance, at the expense of additional space.

7.2 Indexing a partitioned schema

This section shows the benefits of partitioning even when compared to an unpartitioned schema with indexes. We designed indexes using the Index Tuning Wizard in SQL Server 2000. We allowed unlimited storage for indexes, but we added updates (SDSS_U) to the input workload. The cost of the SDSS_U workload increases considerably with every new index built, since that index would require the updated data to be properly ordered. Since the partitioned schemas are already optimized for the particular workload, they will require much less indexing effort, offering better performance for both retrieval and update statements

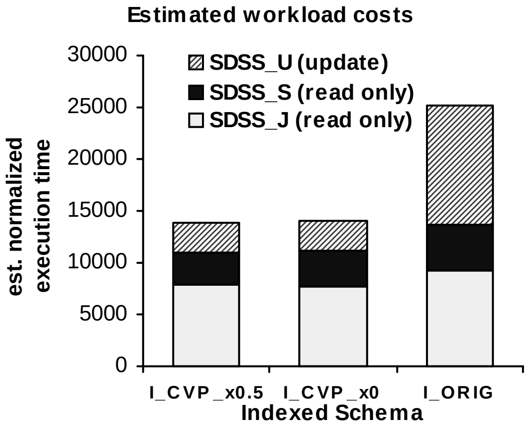

FIGURE 11: *Query and update workload costs using the original and the vertically-partitioned indexed schemas.*

Figure 11 .shows the total workload cost when using the indexed original and partitioned schemas, for all the statement groups (SDSS_J, SDSS_S, and SDSS_U). When using the I_CVP_x0 and I_CVP_x0.5 schemas, read-only statements run 20% faster compared to the original schema, whereas the insertion statements are more than 5 times faster. Overall, the partitioning improves query execution performance even in the presence of indexes, by approximately 45%.

Figure 12 shows the total amount of storage allocated for the I_ORIG and the two partitioned schemas, broken down into the storage required to index the two main tables. The partitioned schemas require about half the storage space for indexes, compared to the original schema. According to Figure 12, the original schema relies on heavily indexing *PHOTOOBJ* for performance. In comparison, because of the performance benefits of partitioning *PHOTOOBJ,* the partitioned schemas require 7 and 4 times less storage of indexing. Instead of heavily indexing

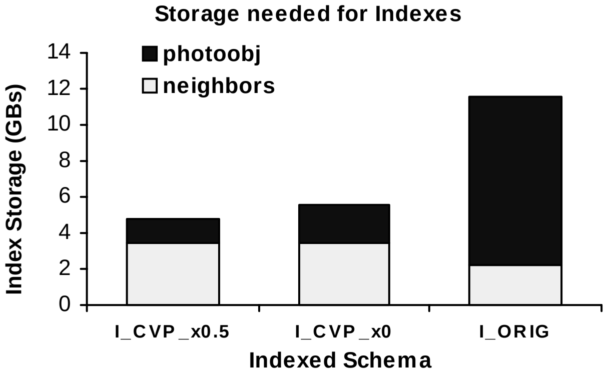

FIGURE 12: *Amount of storage needed for indexes when using the original and vertically partitioned schemas.*

PHOTOOBJ, the partitioned schemas allocate some more space for the efficient indexing of *NEIGHBORS*.

Our experimental results in this section show that partitioning can improve query execution performance, requiring less indexing overhead.

8 Conclusions

Database applications that use multi-terabyte datasets are becoming increasingly important for scientific fields such as astronomy and biology. In such environments, physical database design is a challenge that involves complex query processing needs as well as space limitations. We propose AutoPart, an algorithm that automatically partitions database tables utilizing prior knowledge of a representative workload. Using a data partitioning and replication, AutoPart suggests an alternative, high-performance schema that executes queries faster than the original one and can be indexed using a fraction of the space required for indexing the original schema. To evaluate AutoPart, we build an automated schema design tool that interfaces to commercial database systems. The paper describes our algorithm, the system architecture, and experimental results using the Sloan Digital Sky Survey database.

9 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

[23] O.Tsatalos,M.Solomon,Y.Ioannidis, “The GMAP: A Versatile Tool for Physical Data Independence”, VLDB 1994

[24] S.Ceri,M.Negri, G.Pelagatti. “Horizontal Data Partitioning in Database Design”, SIGMOD 1982