Workload-Driven Schema Design for Large Scientific Databases

Stratos Papadomanolakis, Anastassia Ailamaki
Carnegie Mellon University
{stratos,natassa}@cs.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 their 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 use of such redundant structures. In this paper we propose schema design techniques to improve query performance without redundancy. We present AutoPart, an algorithm that automatically partitions database tables utilizing knowledge of a representative workload. To evaluate AutoPart we built an automated schema design tool that interfaces to commercial database systems. We experiment with a real world astronomical database, the Sloan Digital Sky Survey. We demonstrate the benefits of partitioning for large-scale systems: Partitioning alone improves query performance by a factor of two on average. With indexes, the partitioned schema outperforms the 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. The ongoing effort to support the Sloan Digital Sky Survey (SDSS) [9, 14] 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 (www.lsst.org) 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 underlying physical design. Recent database research has produced tools that utilize workload information to automatically design indexes or materialized views [1, 4, 6] and automated design algorithms are already incorporated in commercial systems. In the case of large-scale applications like SDSS, performance is optimized through implementing a large set of *covering* indexes, since accessing the large base tables (even through non-clustered indexes) is prohibitively expensive. Large numbers of indexes, however, are expensive to store, as they *replicate* data columns

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

from the base tables. This 'data replication' incurs a significant storage management overhead, especially for multi-terabyte scientific databases. Indexing also complicates updates. For instance, new experimental or observation data is often inserted in the database and derived data is recalculated using new models. During update operations, new (or updated) data values must be sorted and inserted in every index that contains them, with a significant processing and I/O cost. The storage and maintenance overhead caused by redundant, auxiliary structures like indexes or materialized views motivates the development of physical design techniques that improve query execution performance while reducing the overheads.

We explore schema design as way to improve performance without redundancy. We present AutoPart [13], an algorithm that partitions the tables in the original database according to a representative workload. AutoPart is incorporated into an automated schema design tool that interfaces to commercial database systems. 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 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.

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. 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. Section 5 discusses the AutoPart architecture. Section 6 presents our experimental results and we summarize in section 7.

2 Related Work

In this section we present previous work on i) partitioning techniques and ii) automated physical design.

2.1 Partitioning Algorithms

Vertical partitioning is known to optimize I/O performance since the early days of relational databases [15]. To determine a suitable partitioning for a relation, initial studies exploited attribute affinity, a measure of how often queries reference combinations of attributes [8, 10, 11]. Affinity-based approaches produce reasonable results, but they dont use estimates of query costs. Subsequent work introduced such cost models in the partitioning process: An integer programming formulation of the partitioning problem [5] makes use of simplified analytical cost formulas, while a partitioning implementation using pair-wise attribute merging [7] uses a cost estimation module similar to todays query optimizers. For horizontal partitioning, a set of disjoint horizontal fragments can be specified by using the predicates in the workload [2]. The fragments are then combined to minimize an application-dependent cost function. An idea similar to the categorical partitioning technique presented in this paper, is used for the more efficient execution of group queries [3].

2.2 Automated Tools for 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, 4, 6] and the automatic declustering techniques for parallel DBMS [12] are based on the optimizers 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.

3 Workload-Driven Data Partitioning

We start with a formulation for the vertical partitioning problem: Given a set of relations $R = \{R_1, R_2,...,R_n\}$ and a set of queries $Q = \{Q_1, Q_2,..., Q_m\}$ generate a set of fragments $F = \{F_1, F_2,...,F_N\}$ such that:

- Every fragment F_i stores a subset of the attributes of a relation in R plus an identifier column.
- Each attribute of a relation in R, is contained in exactly one fragment in F. (except for the primary keys).
- The sum of the query costs when executed on top of the partitioned schema, *cost*(Q, (R-F)) is minimized.

We expect the minimal workload cost to be much less than the original cost, because in the partitioned schema queries will avoid accessing attributes they do not need. The Query Access Set (QAS) of a query Q_i with respect to a relation R_j (QAS(Q_i,R_j)), is the subset of R_j's attributes referenced by Q_i. In the ideal case, the QAS for all the queries in the workload are disjoint: then the solution to the vertical partitioning problem is 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, workloads contain overlapping QAS, making such 'clean' solutions infeasible.

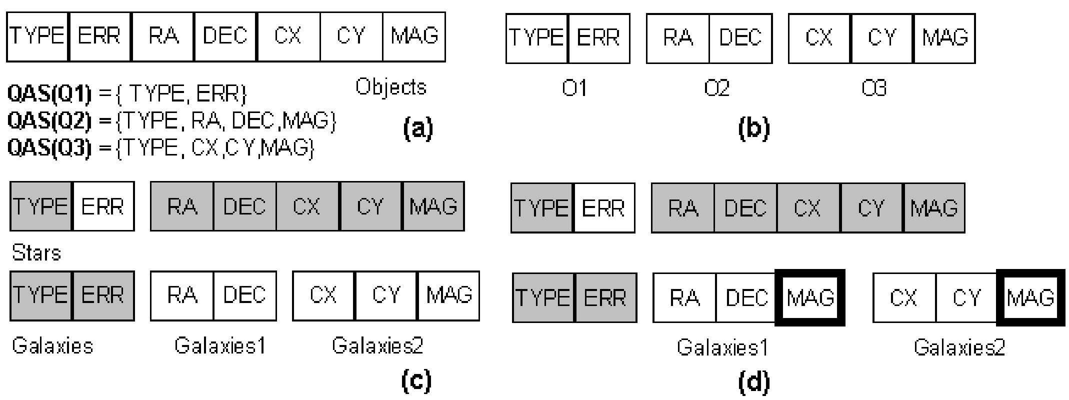


Figure 1: Partitioning example. (a) Original schema. (b) After vertical partitioning. (c) After vertical & categorical partitioning. (d) Column replication.

Consider the example in Figure 1, drawn from a simplified astronomical database. Our database consists of a single table (Objects) that stores objects like galaxies and stars. Our workload consists of queries Q_1,Q_2,Q_3. Figure 1(b) shows one possible solution for vertically partitioning Objects into fragments O_1, O_2, O_3. Q_1 needs to access only O_1, minimizing its I/O requirements. Since the attribute TYPE exists in all QAS however, queries Q_2 and Q_3 will have to access fragment O_1 in addition to O_2 and O_3 and perform additional joins. Also, Q_2 will have to access fragment O_3 to obtain its missing attribute. The alternative approach of merging some of the O_1,O_2,O_3 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.

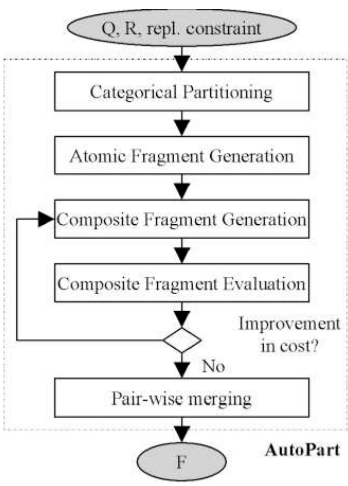
Overlapping QAS in a workload make it impossible to avoid additional joins. Often, however, much of the overlap implied by overlapping QAS is not real. In the previous example, assume that Q_1 restricts its search to objects of type 'Star' (TYPE = 'Star'), whereas Q_2 and Q_3 only care about objects of type 'Galaxy' (TYPE =

'Galaxy'). In this case, considering only QAS leads to 'false sharing' as Q_1 will process a completely disjoint set of tuples than Q_2 and Q_3. By *categorically* partitioning Objects we remove the overlap between QAS(Q_1) and {QAS(Q_2),QAS(Q_3)}, since the queries now access only the categorical fragments (Figure 1 (c)). The fact that Q_1 needs to access attributes {TYPE, ERR} together does not affect Q_2, Q_3. With this form of partitioning queries benefit from the elimination of unnecessary accesses to objects of the wrong class and also from the removal of categorical columns (in this example we can remove TYPE from the categorical fragments altogether).Note that even in the categorically partitioned schema of Figure 1 (c), there is still an overlap between QAS(Q_2) and QAS(Q_3) on MAG. We reduce the impact of such overlaps, that cannot be removed by categorical partitioning, by allowing the replication of attributes belonging to the intersection of two or more QAS. In our example, we replicated attribute MAG in the two fragments, Galaxy1 and Galaxy2, in order to remove the remaining joins (Figure 1 (d)). The resulting schema has no additional joins or unnecessary data accesses.

4 The AutoPart Algorithm

We define a *fragment* of a relation R to be a new relation containing a subset of R's attributes. We distinguish between two kinds of fragments: *atomic* fragments are the 'thinnest' possible, and are accessed atomically in the workload: there are no queries that access only part of an atomic fragment. In addition, atomic fragments are disjoint and their union is equal to R. We define a *composite* fragment as the union of two or more atomic fragments. AutoPart generates a set of atomic or composite fragments than minimizes the input workload's cost.

The general structure of AutoPart is shown in Figure 2 (a). The first step is to categorically partition the input relations. Section 4.1 describes our categorical partitioning algorithm in more detail. Then the algorithm vertically partitions the categorical fragments. Section 4.2 describes our vertical partitioning approach.




```
categorical_partitioning(relation R, queries Q, size N)
  1. A := categorical attributes ∈ R.
  2. Let X = collection of predicates in Q of the form xi:{Ai ∈ d ⊂ Di}
  3. XM := complete_minimal_set({xi})
  4. horizontal fragments Y := minterm_fragments(X)
  5. if |Y| ≤N then
     5a. let A := A - {ai} /* remove attribute ai */
     5b. Yi := merge(Y,ai) /* merge the hor. fragments defined using ai*/
     5c. Select ai that leads in the minimal |Yi|
     5d. Set Y := Yi
6. Repeat step 5 until |Y| satisfies size constraint
7. For each fragment F ∈ Y, remove all attributes in F that take a single value.
```

(a) (b)

Figure 2: (a) Outline of the AutoPart algorithm, (b) Categorical Partitioning algorithm

4.1 Categorical Partitioning

The categorical partitioning step (Figure 2 (b)) 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 algorithm first identifies the set of categorical attributes A_i in the relation R and their corresponding domains D_i (step 1), by consulting the system's designer or the system catalog. Each query in workload Q 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 subsets, which will be assigned to different horizontal fragments. We use the methodology developed in [2].

We can express every query predicate involving each of those attributes in the form $x_i : \{A_i \in d \subset D_i\}$. Let X = $\{x_i\}$ be the collection of such predicates, and assume that it is minimal and complete, according to [2]. Then, the min-term predicates Y(X) [2] 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 A_i that take a unique value in the horizontal fragments determined, they are removed (Step 5).

4.2 Vertical Partitioning

The vertical partitioning algorithm first examines an initial solution consisting only of atomic fragments. Using only atomic fragments probably has a negative impact on performance, because of the large number of additional joins that must be performed.

The algorithm improves the initial schema by forming composite fragments that reduce the joining overhead but increase I/O cost: queries accessing a composite fragment don't necessarily reference all the attributes in it. The Composite Fragment Generation module determines a set of composite fragments that should be considered for inclusion in the schema, while the Composite Fragment Selection module evaluates the available options and chooses the fragments that are found to provide the highest improvements for the workload.

AutoPart iterates through the composite fragment generation and selection steps, each time expanding the fragments selected in the previous steps. The composite fragments considered may contain attributes that are also included in other fragments in the partitioned schema, thus allowing for attribute replication. The storage space available for replication can be controlled by an *storage constraint* parameter provided to the algorithm. The storage constraint that maximizes performance depends on the properties of the workload. One can identify the optimal value for the storage constraint by running AutoPart multiple times.

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. A detailed description of AutoPart's components can be found in [13].

5 System Architecture

This section describes the functional blocks of the automated schema partitioning tool, depicted in Figure 3. The system was implemented 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 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. 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.

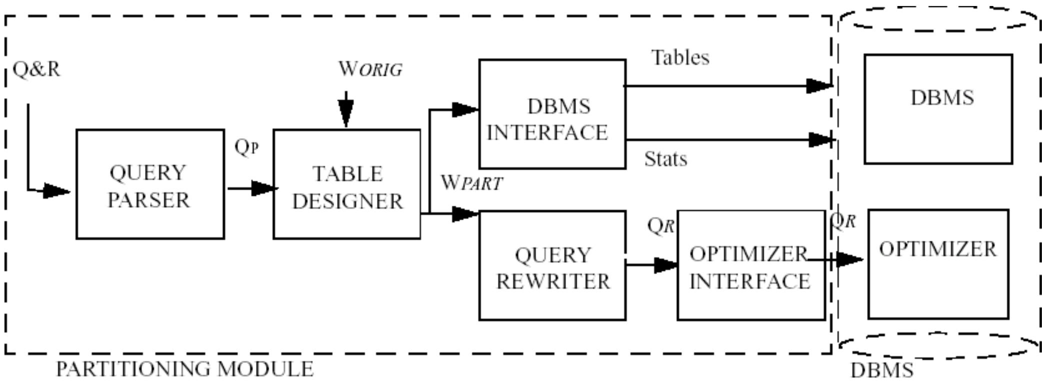


Figure 3: Partitioning tool architecture

- 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, without actually loading the partitioned tables with data.
- 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.

6 Experimental Evaluation

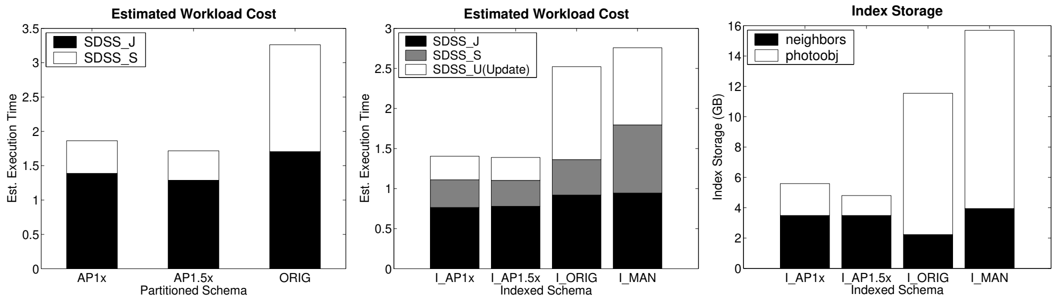
We present experimental results on (a) the performance of AutoPart without indexes and (b) the benefits of partitioning in the presence of indexes and updates.

6.1 Experimental Setup

Our experiments use the Sloan Digital Sky Survey (SDSS) database [9, 14], running on SQL Server 2000. The machine we use for the experiments is a workstation with a single 1-GHz Athlon CPU, 512 MB of main memory and two 120-GB IDE disks. The two largest tables in the SDSS database are *PHOTOOBJ* (22GB), a 'catalog' table with hundreds of attributes for each astronomical object, and *NEIGHBORS* (5GB), which stores spatial relationships between neighboring objects. The SDSS workload [9, 14] is a set of 36 representative SQL statements. In order to simulate the addition of new data, we added two insertion (*SQL INSERT*) statements. *PHOTOOBJ* and *NEIGHBORS* are almost exclusively responsible for the I/O costs in the SDSS workload and we therefore use AutoPart only on them. In the following sections we present performance results in terms of the optimizer estimates of the workload's cost. Table 1 documents the alternative schemas we experimented with. We detail our experimental setup in our recent study [13].

ORIG	Original non-partitioned schema without indexes.
AP1.5x	AutoPart schema with column replication (allowing 50% additional space).
AP1x	AutoPart generated schema without column replication.
I_ORIG	Indexed non-partitioned schema (using the Index Tuning Wizard)
I_MAN	Manually indexed non-partitioned schema
I_AP1.5x	Indexed AutoPart schema with column replication (using the Index Tuning Wizard)
I_AP1x	Indexed AutoPart schema with no column replication (using the Index Tuning Wizard)

Table 1: Description of the schemas used in our experiments



(a) (b) (c)

Figure 4: (a) Workload costs without indexes, (b) With indexes, (c) Storage allocated for indexes

6.2 Evaluation of Partitioning

This section demonstrates that AutoPart significantly improves query execution without the use of any indexes. We derive two partitioned schemas, AP1.5x and AP1x, with and without replication respectively. In the attribute replication case, we set a storage bound for the replication columns equal to 1/2 the original database size. Experimenting with higher storage bounds did not further improve performance.

We distinguish between two groups of SDSS queries. The join-bound group (SDSS_J) consists of four queries, whose execution is bounded by expensive joins among several instances of *PHOTOOBJ* and *NEIGHBORS*. The scan-bound group (SDSS_S) includes queries dominated by table scans. This distinction is necessary because the two query classes do not benefit equally from partitioning. We expect SDSS_J queries to benefit less because their dominant operators are the joins, which cannot be improved by partitioning. SDSS_S queries on the other hand, benefit significantly since they are bound by I/O.

Figure 4 (a) shows the estimated workload performance for the two query groups. Partitioning improves SDSS_S queries by 2 or 3 times, even without the use of any indexes. In addition, column replication is beneficial for performance, with the schema using replication (AP1.5x) performing 10%better than the schema without (AP1x). SDSS_J queries improve much less (19% to 24%), since partitioning does not benefit join performance.

6.3 Indexing a Partitioned Schema

In this section we extend our comparison by building indexes on the partitioned and the original schemas. We design indexes using the Index Tuning Wizard in SQL Server 2000. We allow unlimited storage for indexes, but add updates (SDSS_U) to the input workload, in order to compare the efficiency of the designs. We also consider a set of hand-tuned indexes, designed by the experts maintaining the SDSS database, which were included with our SDSS distribution.

Figure 4 (b) shows the total workload cost when using the indexed original (I_ORIG, I_MAN) and partitioned schemas (I_AP1.5x and I_AP1x) for all statement groups. I_ORIG is the indexed schema generated by the Index Tuning Wizard and I_MAN is the manual design. Due to the effectiveness of partitioning the partitioned schemas use fewer and smaller indexes. Because of the reduced. indexing requirements, workload (SDSS_U) shows a dramatic 5x improvement for the partitioned schemas. Despite the fewer indexes, queries in the SDSS_S class improve by 20% compared to I_ORIG, because of the combined effect of partitioning and indexing. The performance improvement is similar for the replication and the no-replication case, because indexes mitigate any performance differences. Overall, partitioning improves query execution performance even in the presence of indexes, by 45% compared to I_ORIG.

Our experiment highlights the effectiveness of automated design techniques, since I_MAN is shown to perform worse than I_ORIG for the query workload (although it does better for updates). In addition, it confirms the benefits of automated partitioning as a physical design technique: We find that our partitioned-only schema

(AP1.5x), Figure 4 (a)) has similar performance to the I_MAN indexed design, without the overhead!

Figure 4 (c) details the total storage allocated for the I_ORIG, I_MAN and the two partitioned schemas, broken down into the storage required to index the two main tables. The 'covering' indexes built on the non-partitioned schemas require additional space larger than the original dataset size. The partitioned schemas require about half the space. Both I_ORIG and I_MAN rely on heavily indexing *PHOTOOBJ* for performance. In comparison, because of the performance benefits of partitioning *PHOTOOBJ*, the partitioned schemas require minimal storage to index *PHOTOOBJ* (4 and 7 times less than I_ORIG). Instead of heavily indexing *PHOTOOBJ*, the partitioned schemas allocate some more space for the efficient indexing of *NEIGHBORS*.

7 Summary

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. We propose AutoPart, an algorithm that automatically partitions database tables utilizing prior knowledge of a representative workload. 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. This paper describes our algorithm, the system architecture, and experimental results using the Sloan Digital Sky Survey database.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. *VLDB* (2000), 496–505.

[2] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *SIGMOD* (1982), 128–136.

[3] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. *VLDB* (1997) 476–485.

[4] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. *VLDB* (1997), 146–155.

[5] Douglas W. Cornell and Philip S. Yu. A vertical partitioning algorithm for relational databases. *ICDE* (1987) 30–35.

[6] G.Valentin, M.Zuliani, D.Zilio, and G.Lohman. DB2 advisor: An optimizer smart enough to recommend its own indexes. *ICDE* (2000), 101.

[7] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. *SIGMOD* (1979), 93–101.

[8] Jeffrey A. Hoffer and Dennis G. Severance. The use of cluster analysis in physical data base design. *VLDB* (1975), 69–86.

[9] J.Gray, D.Slutz, A.Szalay, A.Thakar, J.vandenBerg, P.Kunszt, and C.Stoughton. Data mining the SDSS skyserver database. Technical Report MSR-TR-2002-01, Microsoft Research, 2002.

[10] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[11] Shamkant B. Navathe and Mingyoung Ra. Vertical partitioning for database design: a graphical algorithm. *SIGMOD* (1989), 440–450.

[12] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. *SIGMOD* (2002), 558–569.

[13] S.Papadomanolakis and A.Ailamaki. AutoPart: Automated schema design for large scientific databases using data partitioning. *SSDM* (2004).

[14] Alexander S. Szalay, Jim Gray, Ani R. Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS skyserver: public access to the sloan digital sky server data. *SIGMOD* (2002), 570–581.

[15] Toby J. Teorey and James P. Fry. The logical record access approach to database design. *ACM Comput. Surv.*, 12(2):179–211, 1980.