RJ 10208 (95082) March 16, 2001
Computer Science

Research Report

AUTOMATING PHYSICAL DATABASE DESIGN IN A PARALLEL DATABASE: THE DB2 PARTITIONING ADVISOR

Jun Rao

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Chun Zhang

University of Wisconsin
Madison, Wisconsin

Guy Lohman
Nimrod Megiddo

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library.CyberDig.nsf/home.

Research Division
Almaden ▪ Austin ▪ Beijing ▪ Haifa ▪ T. J. Watson ▪ Tokyo ▪ Zurich

Automating Physical Database Design in a Parallel Database: the DB2 Partitioning Advisor

Jun Rao
IBM Almaden Research Center
junrao@almaden.ibm.com

Chun Zhang
University of Wisconsin at Madison
czhang@cs.wisc.edu

Guy Lohman
IBM Almaden Research Center
lohman@almaden.ibm.com

Nimrod Megiddo
IBM Almaden Research Center
megiddo@almaden.ibm.com

Abstract

Physical database design is important for query performance in a shared-nothing parallel database system, in which the base data is partitioned among multiple independent nodes. Given a workload, we seek to determine automatically how to partition the base data across multiple nodes to achieve overall optimal performance. Previous attempts use heuristic rules to make those decisions. These approaches fail to consider all of the interdependent aspects of query performance typically modeled by today's sophisticated query optimizers.

This paper presents a comprehensive solution to the problem that has been tightly integrated with the optimizer of a commercial shared-nothing parallel database system—IBM Universal Database Enterprise Extended Edition (EEE). Our approach uses the query optimizer itself both to recommend candidate partitions for each table that will benefit each query in the workload, and to evaluate various combinations of these candidates in an intelligent way that scales with the number of tables and queries in the workload. We compare a steepest-ascent heuristic with a genetic algorithm, which quickly converge to an optimal solution on a TPC-H workload that is 35% better than the human-selected partitions. Our experimental results demonstrate the effectiveness of this technique and the scalability of our algorithm for large workloads.

1 Introduction

Database systems increasingly rely upon parallelism to achieve high performance and large capacity [DG92]. Most of the major database vendors – such as IBM [Cor00a], NCR [Cor00c], Oracle [Cor00d], Sybase [Cor00e], etc. – have support for parallelism. Rather than relying upon a single monolithic processor, parallel systems exploit fast and inexpensive microprocessors to

achieve high cost-effectiveness and improved performance. The popular shared-memory architecture of symmetric multiprocessors is relatively easy to parallelize, but cannot scale to hundreds or thousands of nodes, due to contention for the shared memory by those nodes. Shared-nothing parallel systems, on the other hand, interconnect independent processors via high-speed networks. Each processor stores a portion of the database locally on its disk. These systems can scale up to hundreds or even thousands of nodes, and are the architecture of choice for today's data warehouses that typically range from tens of terabytes to over 100 terabytes of online storage. High throughput and response times can be achieved not only from inter-transaction parallelism, but also from intra-transaction parallelism for complex queries.

Because data is partitioned among the nodes in a shared-nothing system, and is relatively expensive to transfer between nodes, selection of the best way to partition the data becomes a critical physical database design problem. A suboptimal partitioning of the data can seriously degrade performance, particularly of complex, multi-join "business intelligence" queries common in today's data warehouses. For example, a join between two tables can be made cheaper if all rows of those two tables having the same value of the join columns are co-located on the same node. Selecting the best way to store the data is complex, since each table could be partitioned in many different ways to benefit different queries, or even to benefit different join orders within the same query. This puts a heavy burden on database administrators, who have to make many trade-offs when trying to decide how to partition the data, based upon a wide variety of complex queries in a workload whose requirements may conflict.

Previous work in the literature tried to choose partitioning heuristically or to create a performance model separate from the optimizer. Heuristic rules unfortunately cannot take into consideration the many interdependent aspects of query performance that modern query optimizers do.

In this paper, we automate the process of selecting the optimal partitioning for all tables in a database, for a given workload, by exploiting the sophisticated cost model of the query optimizer itself. We use the optimizer's cost estimates to both suggest possible partitionings and to compare them in a quantitative way that considers the interactions between multiple tables within the workload. Our approach therefore avoids redundancy – and possible inconsistency – between the Partitioning Advisor and the query optimizer. It also lowers the cost of ownership for customers, by simplifying the design tasks typically performed by increasingly rare and expensive database administrators. Our Partitioning Advisor has been prototyped in IBM's DB2 Universal Database for Unix, Windows, and OS/2 as part of the Self-Managing And Resource Tuning (SMART) project, and is completely operational. Our tool can be used to decide the initial data partitioning before loading the data into the database and to choose data repartitioning when the workload or the

underlying data has changed.

The rest of the paper is organized as follows: We discuss related work in Section 2. Section 3 gives an overview of our approach. We then describe the four components in our system–recommend partitions, evaluate partitions, cost estimation and enumeration algorithms–from Section 4 to Section 7. Our experimental results are presented in Section 8. We conclude in Section 9.

2 Related Work

Physical database design in relational DBMSs has been studied for decades. How to "horizontally partition" rows of tables was among that early work. In a shared-nothing environment, horizontal partitioning takes the form of declustering (i.e., partitioning) tables across many nodes to support a high degree of intra-query parallelism for complex queries, effectively providing a static form of load balancing [CNW83, SW85, CABK88, Gha90, Zil98]. However, none of the previous work used the query optimizer in a database server to evaluate alternative solutions. Most of the approaches tried to come up with some cost models of their own to estimate the benefit of different partitions. Therefore, the partitioning decision made by these authors might not be consistent with – or as accurate as – the detailed cost model used by the optimizer.

A substantial amount of research has been conducted on dynamic load balancing in parallel shared-nothing database systems (a lot of the references can be found in [RM93]). The potential for dynamic load balancing is limited for operations (such as scans) where the execution location is statically determined by the partitioning and the allocation of the database among processing nodes. As a result, most dynamic load balancing work focuses on operators such as joins, which typically work on derived data. Our work complements that in dynamic load balancing by recommending the data partitioning for the stored data. All the strategies used in dynamic load balancing can be applied to achieve further query improvement.

Database design is one aspect of database management, the automation of which has become increasingly important as the cost of people grows, while the costs of hardware and software decrease. "Self-managing" databases have become a "hot topic" in commercial database systems. Microsoft Research's AutoAdmin project [CN98, ACN00] has developed wizards that automatically select indexes and materialized views for a given workload. IBM [VZZ+00], Informix [Cor00b], and Oracle have similar efforts that have also built such tools. However, our work is the very first to consider the automatic selection of table partitioning in parallel shared-nothing database systems. There is a fundamental difference between partition selection and index/materialized view selection. Indexes (and materialized views) are auxiliary structures in addition to the base table and a table can have as many indexes (clustered index is an exception) as it needs. However, a table can only

be partitioned in exactly one specific way. This means that these previous algorithms for selecting auxiliary structures cannot be applied directly to the selection of optimal partitions.

3 Overview of Our Approach

In this section, we first introduce IBM's parallel database system. We then describe the architecture of our Partitioning Advisor.

3.1 IBM Parallel Database System

IBM Universal Database for Unix, Windows, and OS/2 Enterprise Extended Edition [BFG+95, Cor00a] (referred to as DB2 in the rest of the paper) is based on a shared-nothing architecture. A collection of processors (nodes) are used to execute queries in parallel. A given query is broken up into subtasks, and all the subtasks are executed in parallel.

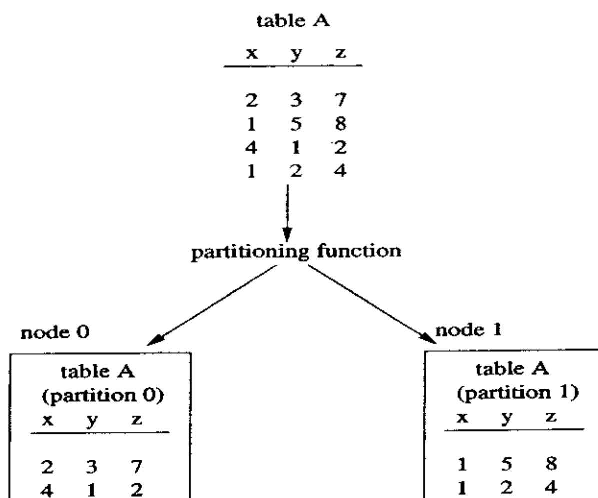


Figure 1: An Example of Horizontal Partitioning

To enable parallelism, tables are horizontally partitioned across nodes. The rows of a table are typically assigned to a node by applying some deterministic partitioning function to a subset of the columns. These columns are called the *partitioning key* of the table. Currently, DB2 supports hash-based partitioning. DB2 also allows multiple *nodegroups* to be defined. A nodegroup can have any subset of the nodes in a system. A (nodegroup, partitioning key) pair uniquely determines how a table is partitioned. An example is shown in Figure 1. Table A is partitioned on Column $A.x$ and thus the partitioning key is $A.x$. There are two nodes in the nodegroup. The partitioning function then generates hash values from column $A.x$ and distributes the hash values evenly between the two nodes. In this example, rows with even values in $A.x$ are assigned to node 0 while rows with

odd values in $A.x$ to node 1. Note that those rows having the same value on the partitioning key are guaranteed to be distributed to the same node. A table can also be replicated across all the nodes in a nodegroup. This is achieved in DB2 by defining a replicated materialized view of the table. In summary, in DB2, a partition of a table is given by a (nodegroup, partitioning key) pair or just the *nodegroup* if the table is chosen to be replicated.

3.2 Architecture

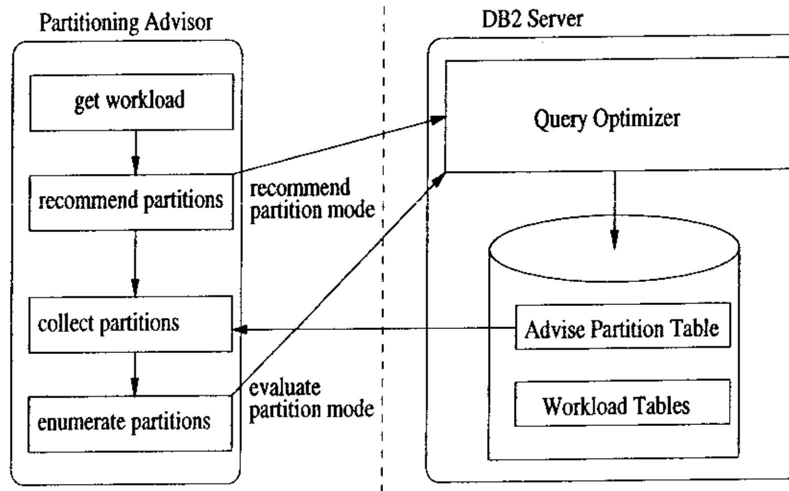


Figure 2: Architecture of DB2 Partitioning Advisor

Figure 2 shows the architecture of the partitioning advisor in DB2. A similar architecture has been used for DB2's Index Advisor [VZZ+00].

DB2's optimizer supports a number of "explain" modes. When a query is optimized in such a mode, the query plan is generated without being executed. The optimizer also writes certain information to some special tables (we refer to them as "advise" tables) in the database. For example, when the optimizer is set to be in RECOMMEND INDEX mode, the optimizer will write a list of candidate indexes to an ADVISE_INDEX table.

We augment DB2's optimizer with two additional explain modes: RECOMMEND PARTITION and EVALUATE PARTITION. When evaluating a query in RECOMMEND PARTITION mode, the optimizer accumulates a list of partitions that are potentially useful for each table and generates plans corresponding to each of these (virtual) partitions. Optimization then proceeds normally to evaluate all of these alternative plans and pick the cheapest. Once the optimizer finds a plan that it considers optimal for the query, it extracts the partition of each base table subplan and writes it to an ADVISE_PARTITION table. Each row in the ADVISE_PARTITION table records information such as the nodegroup, the partitioning key, and whether the table is replicated or not. It also has

a *USE_IT* attribute, which will be turned on/off by our Partitioning Advisor on the client side. In the EVALUATE PARTITION mode, the optimizer first extracts from the ADVISE_PARTITION table those rows with the USE_IT bit turned on and replaces the real partition with the new one for the corresponding table. The optimizer then starts optimizing the query, assuming that the tables are partitioned in the newly-specified way. In both modes, query plans are generated without actually being executed.

Our Partitioning Advisor is built as an application to DB2 server. The Partitioning Advisor first collects a workload of queries and their frequency of occurrence, using the same DB2 mechanism as the Index Advisor [VZZ+00]. It then invokes the optimizer to evaluate all the statements in the workload in RECOMMEND PARTITION mode. After that, the advisor reads all the recommended partitions from the ADVISE_PARTITION table. Finally, an enumeration algorithm will combine recommended partitions from different tables in certain ways and evaluate the workload in evaluate partition mode for each combination. The best partition for each table and the corresponding cost for the workload are returned to the user in the end.

Our architecture assumes that we have some statistics on the database for the query optimizer's cost estimation. This is reasonable when data is already loaded into the system (for repartitioning). However, sometimes it is desirable to know how to partition the underlying data without loading it first. DB2 provides a utility that can piggy-back statistics calculation while importing data into the database server. We exploit that utility by only collecting the statistics from external data without actually populating the database. Our algorithm requires that there is already some partitioning for each table, but these partitions can be picked arbitrarily, e.g. the default partitions assigned by DB2. We refer to these partitions as "real" partitions.

4 Recommend Partitions

In this section, we first briefly describe the optimizer used in DB2. We then introduce "interesting" partitions. Finally, we present the server side changes to support RECOMMEND PARTITION mode.

4.1 DB2 optimizer

DB2 uses a conventional bottom-up optimizer that uses dynamic programming to prune dominated alternatives [SAC+79, GLSW93]. In a parallel environment, the optimizer considers several partitioning alternatives for (equality) joins. If two tables are both partitioned on join keys (and are in the same nodegroup), the join between the two tables can be performed locally at each node. This kind of join is called a *local join* [BFG+95]. Otherwise, at least one of the participating tables

has to be moved. If one of the tables (call it table A) is partitioned on (a superset of) the join key, we can dynamically repartition the other table (call it table B) on the join key to the nodegroup of table A. This join method is known as a *directed join*. Alternatively, the optimizer can replicate data from table B to all nodes in table A. This join method is known as a *broadcast join*. Finally, if neither table is partitioned on the join column, the optimizer could decide to repartition both tables over some completely different set of nodes using the join key as the partitioning key. This method is known as a *repartition join*. Typically, local joins are cheaper than directed and broadcast joins, which are themselves cheaper than repartitioned joins, as considerable communication cost can be saved.

4.2 Interesting Partitions

Interesting orders [SAC+79] are row orders that are beneficial in evaluating a query. The optimizer retains the cheapest subplan that produces rows in each "interesting" order and the cheapest "unordered" subplan. Those subplans with interesting orders could make later operations such as merge join, aggregation, and ordering cheaper. In the parallel environment, DB2 collects beneficial partitions for a query as "interesting" partitions. Similar to interesting orders, subplans having interesting partitions could make the whole plan cheaper. In DB2, the optimizer retains the best subplan for each interesting partition, in addition to each interesting order.

DB2 considers the following partitioning keys to be interesting: (a) columns referenced in equality join predicates, (b) any subset of grouping columns. Join columns are interesting because they make local and directed joins possible. Grouping columns are interesting because aggregations can be done locally at each node and then concatenated. These interesting partitions are generated before join enumeration starts, and are accumulated and mapped to each participating base table in the query.

4.3 RECOMMEND PARTITION Mode

When optimizing a query, we want to compute for each base table a list of *candidate partitions* that can help reducing the cost of the query. Interesting partitions are certainly candidate partitions for each base table, as each might benefit some operation in that query. However, we have to decide another factor that determines a partition—the nodegroup. For each interesting partitioning key generated, we have to decide which nodegroup to use. One possibility is to use the default nodegroup for the query (which in DB2 is the nodegroup containing all nodes but the catalog node). However, the best nodegroup for the query could be different depending on factors such as table size and communication bandwidth. A more aggressive approach is to create some additional nodegroups

and consider a partition in each of them. The problem is that this will significantly increase the plan search space for the optimizer – the number of partitions compound with the search space of joins, which could already be exponential in the number of participating tables [OL90]. The approach we take is a compromise between the two. We consider all nodegroups that have already been created in the database system, and pair them with the interesting partitioning keys.

Besides interesting partitions, we add two other kinds of partitions as candidate partitions. Consider a local equality predicate of the form $col = constant$. If the table is partitioned on col, then all the satisfying rows are from a single node (the effective number of nodes is reduced to one). Although this increases the number of rows to be processed at a node, it reduces communication cost and can be a winning partition. So we will generate partitions for every column bound to a constant in all the existing nodegroups and add them to the candidate partition list of the corresponding table. Note that for systems supporting range partitioning, columns referenced in non-equality predicates might also be beneficial partitioning keys. Another kind of partition that may be interesting is replication. For small tables, replicating them across all nodes can potentially improve performance without increasing storage overhead too much. Thus, for each table, we add to its interesting partition list a replicated partition, if the table size is smaller than a threshold. The threshold can be adjusted by the user.

Returning all the candidate partitions directly to the Partitioning Advisor is not practical, as there are usually too many candidate partitions for each table. We instead select the best candidate partition for each table in the query. This is realized by augmenting the optimizer with a RECOMMEND PARTITION mode. Under such a mode, we change the plan enumeration rules so that instead of always generating base table plans using the real partition, the optimizer will in addition generate a plan corresponding to each candidate (virtual) partition. When the optimizer returns the best overall query plan, we write to an ADVISE_PARTITION table the best partition chosen for each table in the query.

As we already know, local predicates can change the number of effective nodes in a partition. Similarly, a pushed-down join predicate can reduce the number of effective nodes to one for a base table plan per each iteration from the outer. DB2 provides a "normalization" routine that updates the effective number of nodes based on the predicates applied. Normalization is important as it can affect query cost estimation. So for each candidate partition, we call the normalization routine to factor in the effect of local and pushed-down join predicates before using it.

Another subtle issue arises when a table is referenced multiple times in a query. When generating subplans, each table reference is considered independently. This means that a plan can have two table scans referencing the same table with different partitions. Such as plan is clearly invalid, as any table can in reality be partitioned in only one way. However, it is expensive to solve this

problem completely. This is because we would have to traverse the plan tree all the way down to the leaves to compare partition information of base tables. On the other hand, it is not crucial that we recommend exactly one partition for a table. Those partitions are just candidates themselves, and are subject to further evaluations (we will elaborate on this in subsequent sections). So, our implementation allows the optimizer to recommend different partitions for a single table within a single query.

5 Evaluate Partitions

In the EVALUATE PARTITION mode, the optimizer reads in the ADVISE_PARTITION table before optimization starts. For each row with the USE_IT bit turned on (we'll see how to set the USE_IT bit in Section 7), the optimizer reconstructs the partition and uses it to replace the real partition of the corresponding table. Our enumeration algorithm guarantees that exactly one partition can be used for each table. Similar to RECOMMEND PARTITION mode, we normalize each newly created partition before using it. The optimizer then estimates the cost of the query under the new partitions by selecting the best plan for the given partitions.

6 Cost Estimation

An important issue we haven't talked about so far is how to estimate cost when we change the real partition of a table to a new virtual partition. In this section, we first introduce the cost model used in DB2 and then describe our approach to making consistent cost estimation when changing partitions.

6.1 DB2 Cost Model

DB2 uses a detailed cost model to estimate query cost. The overall cost is a function of I/O cost, CPU cost, and communication cost, by assuming that there is some overlap among the three components. DB2 collects various kinds of statistics on the database, including table cardinality, column cardinality (number of distinct values in a column), number of data pages in a table, and optionally distribution statistics such as histograms and a list of the most frequent values. A set of parameters (we call them *cost parameters*) are then derived from the statistics. Typical cost parameters include cardinality and selectivity for each predicate. While the statistics are for the whole table, DB2 maintains two sets of some cost parameters, one at the table level and one at a single node level (we refer to them as *per-table* and *per-node* parameters, respectively). Both sets of parameters are needed for cost estimation. For example, when estimating the I/O cost of

a scan, the per-node level information (such as number of disk pages) is used. This is based on the assumption that the scan is performed in parallel across all the nodes and is the way that DB2 employs to encourage parallelism (other commercial system will need similar mechanisms). On the other hand, when collecting join results from all the nodes (for further operations such as aggregation), DB2 uses the per-table cardinality and join selectivity to estimate the number of rows to be received. This guarantees that we get consistent join cardinality estimation independent of how the data is partitioned. To maintain consistency between two sets of cost parameters, DB2 has a mechanism to derive per node level information from per-table level information, taking into consideration how the data is partitioned. In the next section, we will discuss how to use such a mechanism to derive consistent cost estimation with new table partitions.

6.2 Estimating Cost Under New Partitions

In both RECOMMEND PARTITION and EVALUATE PARTITION mode, we need to update the cost parameters when simulating new partitions. Observe that the per-table cost parameters are always the same no matter how the data is partitioned. However, per-node information could change when the underlying partition changes.

Per-table cost parameters are derived from statistics before query optimization starts. As a result, in RECOMMEND PARTITION mode, we still have the right per-table cost parameters as partitions are changed during optimization. In EVALUATE PARTITION mode, we replace the partitions after the per-table cost parameters have been derived. This way, the per-table level information remains the same as when real partitions are used.

The per-node level information is more difficult to maintain since it can change when tables are partitioned differently. Collecting per node statistics under new partitions is out of the question as these partitions are all virtual. Our approach is to always derive per-node cost parameters from the per-table ones, using some existing DB2 mechanisms and a few new ones we needed to add. In the following paragraphs, we summarize how to modify some typical cost parameters.

Cardinality: Per-node cardinality will change when the number of nodes changes or when a partition switches from non-replication to replication or vice versa. We used the following formula to estimate per-node cardinality:

$$\begin{aligned} \text{card(per-node)} &= \text{card (per-table) / number of nodes, if the table is not replicated,} \\ &= \text{card (per-table), otherwise.} \end{aligned}$$

Other cost parameters such as the number of disk pages and the number of overflow pages are adjusted in a similar way.

Column Cardinality: Per-node column cardinality is useful for estimating the number of re-evaluations of the inner in a pushed-down join plan, and thus can affect join cost. Deriving per-node column cardinality from the per-table one is a difficult problem in general. There have been lots of estimators proposed in the literature [HS98]. In DB2, we choose to use a variant of Cardenas's formula [Car75] (given below) by taking into account potential skews.

> $d = D(1-(1-\frac{n}{N})^{N/D})$, where d and D are column cardinality per-node and per-table respectively, and n and N are per-node and per-table cardinality.

Selectivity: A per-node selectivity is derived by multiplying the per-table selectivity with a *multiplier*, which is calculated as follows:

> $multiplier = np^{\frac{bc}{pc}}$, where np represents the number of nodes, pc represents the number of partitioning columns, and bc represents the number of partitioning columns that are bound to constants or outer references (through pushed-down join predicates).

For example, suppose a local equality predicate ($col = const$) has a table-level selectivity f. The per-node level selectivity depends on how the table is partitioned. If the table is partitioned on columns other than col, the per-node level selectivity should also be f (i.e., $multiplier = 1$). If, on the other hand, the table is partitioned on col, $multiplier$ is equal to the number of nodes, as all the satisfying rows are from a single node.

In EVALUATE PARTITION mode, we only need to update the per-node parameters once, based upon the new (virtual) partition that is read in for each table. However, in RECOMMEND PARTITION mode, we are considering many candidate partitions, and thus we need to update the per-node parameters within the cost estimation module for each partition that is to be evaluated.

7 Enumerating Algorithms

The best partitioning for each table depends heavily upon how it is used. Thus, our Partitioning Advisor first collects a workload consisting of a list of queries and update statements. Each query and update statement has a frequency value indicating the number of times it appears. The workload can come from a file or from a workload table stored in the server.

Given a workload, our Partitioning Advisor will evaluate each query in RECOMMEND PARTITION mode. Then it will read in the ADVISE_PARTITION table from the server and accumulate for each table referenced in the workload a list of recommended partitions. The combinations of recommended partitions from all the tables form a search space. We use a *configuration* to

refer to a combination in which each table chooses exactly one partition from its recommended partition list. Our goal is to get the "optimal" configuration so that the entire (weighted) workload has the lowest estimated cost under that configuration. Given a configuration, we can estimate the workload cost by turning on the USE_IT bit of the specified partitions in the ADVISE_PARTITION table[1] in the server and evaluating the workload in EVALUATE PARTITION mode. The question is then how to search the space of configurations wisely. The naive solution is to enumerate all the configurations and estimate the cost of each. However, this solution won't scale when there are too many configurations. Differences between the partition selection problem and index/materialized view selection prevents us from using previous algorithms directly. We propose two new approaches to reduce the search space.

7.1 Rank-based Enumeration

Our first approach is to calculate a benefit value for each recommended partition. We then rank all the partitions based on their benefit and enumerate partitions in such a way that partitions with higher benefit are considered earlier. The enumeration process stops when we reach a time limit, which can be set by the user. We estimate the benefit of all partitions in a query as the difference between the optimizer's estimated cost using the real partitions and its cost when evaluated in RECOMMEND PARTITION mode. We then accumulate for each candidate partition the total benefit of all queries in the workload that use that partition. Algorithm 7.1 shows the details of this approach. Observe that the benefit we have accumulated is not the real benefit each recommended partition will bring. This is because we assign the benefit for the whole query to all of the underlying new partitions, while in fact, some partitions may contribute more to the benefit than others. Nevertheless, we will show in our experimental results (Section 8) that this simplification nonetheless provides good search guidance.

There are various ways to enumerate configurations based on the ranking of each partition. Consider a simple example in which there are two tables, each having five recommended partitions. Figure 3 shows two possible enumeration methods, with each dot in the figure representing a specific configuration (there are 25 of them). In Figure 3(a), the enumeration starts with the first partition in *table*1 and tries all the partitions in *table*2 before moving on to the next partition in *table*1. We refer to this method as *linear enumeration* as it scans linearly in one dimension. Figure 3(b) shows a different approach, in which enumeration is performed diagonally (thus referred to as *diagonal enumeration*). While linear enumeration favors partitions in one dimension, diagonal enumeration considers all dimensions uniformly. Diagonal enumeration has the potential of considering good configurations early. For example, configuration (2,1) (with the second-rank partition in *table*1 and

[1] We guarantee that exactly one partition is turned on for each table by checking the timestamp.

the first-rank partition in *table*2) is considered in the third iteration in diagonal enumeration but not until the sixth iteration in linear enumeration. The difference will be more significant when there are more tables (dimensions) and more recommended partitions in each table. The details of both algorithms are described in Algorithm A.1 and Algorithm A.2 in Appendix A. We will validate both enumeration algorithms in Section 8.

Algorithm 7.1: Rank-based Enumeration

```
for each query q in the workload
{
  q.initial_cost = query cost using real partitions
  q.recommend_cost = query cost in RECOMMEND PARTITION mode
}
read from ADVISE_PARTITION table and generate a unique recommended partition list for each table
set the benefit of each recommended partition to 0
for each unique recommended partition p in each table
  for each query q that recommends p
    p.benefit+ = q.initial_cost − q.recommend_cost
sort recommended partitions for each table in descending benefit order
initialize configuration enumeration process
while ( there is next configuration)
{
  turn on the USE_IT bit of the partitions specified in the configuration
  let the optimizer estimate the workload cost in EVALUATE PARTITION mode under the next configuration
  update the best cost and best configuration if necessary
}
```

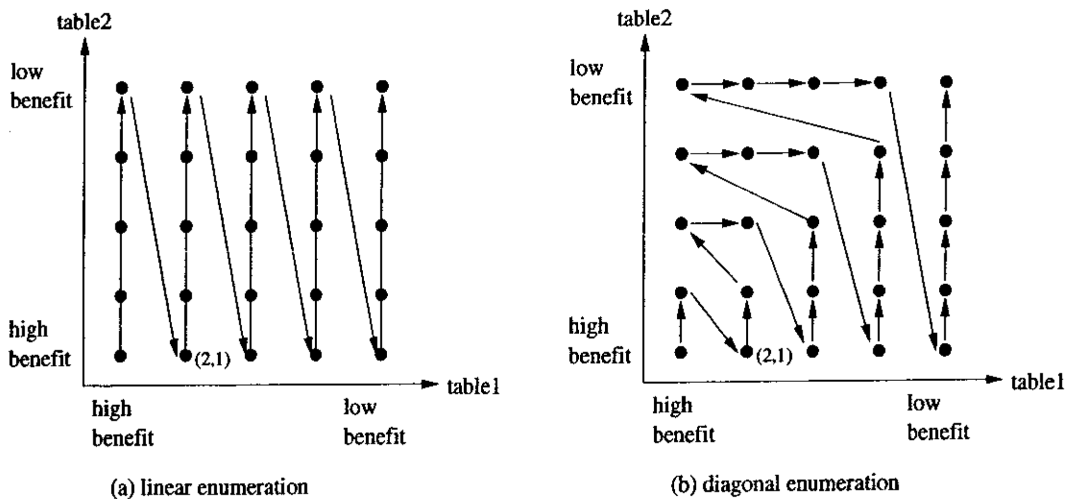


Figure 3: Enumeration Orders

We can also reduce the number of configurations to be iterated by considering only the first-rank recommended partition for small tables. The partitioning of small tables typically won't affect workload cost significantly.

7.2 Randomized Enumeration

Our second approach is to use randomized search algorithms. *Simulated annealing* [KGV83] has been used in data clustering in centralized database systems [HLL94] and query optimization [IK91]. However, we chose to use *Genetic Algorithms*, as they have some good searching features and can be mapped to our problem more easily.

Genetic algorithms [Gol89] are search algorithms based on the mechanics of natural selection and natural genetics. The whole search space is modeled as a set of genes with each gene having some number of gene types. Each search point is then modeled as a species with all the genes set to specific gene types. The algorithm starts with an initial population (which is chosen randomly) consisting of a set of species and tries to derive better search points by evolving next generations. There are typically two ways to evolve. The first one is called *crossover*, in which two species are taken from the population and their genes are randomly recombined to form two descendants. The intuition here is that good parents are likely to provide better children. The second one is called *mutation*, in which we select a species and randomly change some of its genes to derive a descendant. Descendants are compared with their parents and will replace the parents if they are better. In this way, we always keep a fixed population size. The best solution is chosen from the final population when the algorithm finishes (after a certain number of evolutions, for example).

Genetic algorithms surpass some other randomized algorithms in that they search from a population of points, not a single point. Thus they are more likely to avoid local optima. Mapping genetic algorithms to our partition selection problem is straightforward. We treat each table as a gene and each recommended partition as a specific gene type. A partition configuration for all tables defines a species. Mutation and crossover then involve swapping partitions and choosing new partitions for some of the tables.

7.3 Combining Rank-based Enumeration with Randomized Enumeration

It's possible to combine the two approaches we have described. For genetic algorithms to perform well, it's important to choose a good population to start with. We can use the benefit calculated for each recommended partition in Section 7.1 in such a way that the probability of a specific partition being selected is proportional to its benefit. As a result, the initial population is more likely to include partitions with higher benefit values.

We can also choose to use a rank-based enumeration algorithm for certain iterations and then switch to the genetic algorithm, including the best configuration found by former in the initial population.

8 Experimental Results

We prototyped our Partitioning Advisor under DB2 V8. All the tests were run on an RS6000 430 model 140 machine with a 333 MHz processor and 512M RAM, running AIX 4.3.3. DB2 allows us to create multiple virtual nodes, all running on the same machine.

DB2 provides a tool (db2look) that can collect the catalog and statistics in a real database and generate a file containing a list of update SQL statements. By running those SQL statements in an empty database, we can re-create the metadata of the real database without having to populate the actual data (of course, we can't execute a query). We used db2look to simulate a 100GB TPCH [TPC] database with 8 nodes. We also changed the database settings to simulate an environment with a 500MB buffer pool size and a 1.25MB/second communication bandwidth. Table 1 shows the initial partition for each table, as determined by a skilled human. Most of the tables are partitioned on their primary keys and are spread across all 8 nodes. The two smallest tables are created on a single node.

table	partition key	node group	number of nodes
region	r_regionkey	2	1
nation	r_nationkey	2	1
part	p_partkey	1	8
partsupp	ps_partkey	1	8
lineitem	l_orderkey	1	8
orders	o_orderkey	1	8
supplier	s_suppkey	1	8
customer	c_custkey	1	8

Table 1: Initial partitions

We use 22 TPCH queries as the workload. All the queries have the same frequency value. We omit update statements in the workload, since shared-nothing parallel systems are typically used for large data warehouses in which costs are dominated by complex queries.

We first present the results using RECOMMEND PARTITION mode. Table 2(a) shows the speedup as a percentage ($= \frac{initial_cost - recommend_cost}{recommend_cost} * 100$) for each individual query when evaluated in RECOMMEND PARTITION mode. Two queries have the biggest improvement. Query 22 was improved by two orders of magnitude in the RECOMMEND PARTITION mode. Most of the cost in the query comes from a join between the `orders` table and the `customer` table. By choosing to partition `orders` on `o_custkey` instead of `o_orderkey`, the join can be performed locally and

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Speedup (%)	10.5	0.6	0.1	0.0	0.6	0.0	1.0	173.7	44.8	0.5	3.3
Query	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Speedup (%)	0.0	5.0	2.4	5.0	0.0	19.6	0.0	0.1	0.0	0.0	13765.1

(a) Query Speedup

region	nation	part	partsupp	lineitem	orders	supplier	customer
1	3	1	2	6	3	4	4

(b) number of recommended partitions per table

benefit	nodegroup	replicated	partition key
16734737.3	1	N	l_partkey
1495842.0	1	N	l_returnflag,l_linestatus
323116.0	1	N	l_orderkey
229034.0	1	N	l_suppkey
0.0	1	N	-
-25084.0	1	Y	-

(c) Recommended Partitions for `lineitem`

benefit	nodegroup	replicated	partition key
853593623.0	1	N	o_custkey
8399539.0	1	N	o_orderkey
1026673.5	1	Y	-

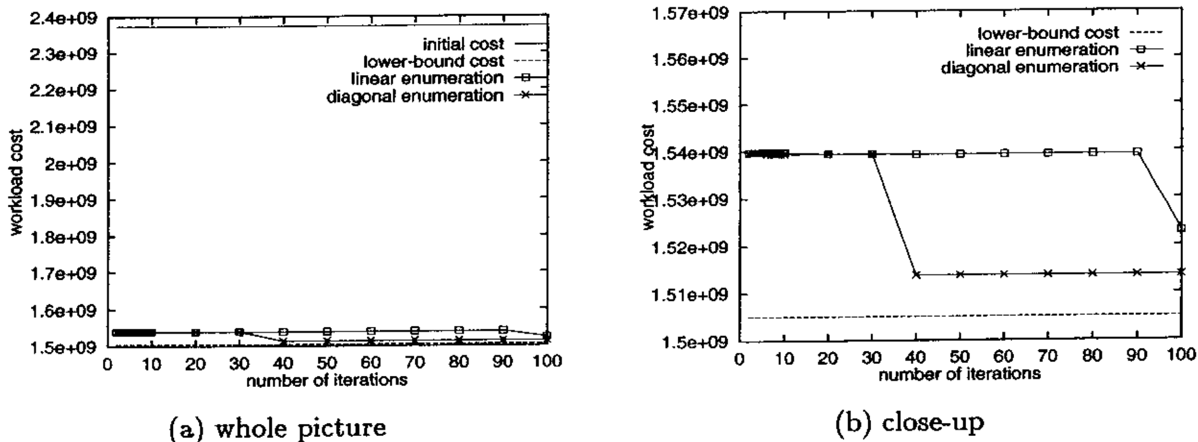
(d) Recommended Partitions for `orders`

Table 2: Results of RECOMMEND PARTITION Mode

thus its cost is reduced significantly. Q8 got 173% improvement. This query contains a three-way join among `lineitem`, `orders`, and `part`. In RECOMMEND PARTITION mode, the optimizer chose to partition table `lineitem` on `l_partkey` and join it with table `part` first. This is cheaper than the initial plan (using real partitions) where table `lineitem` is joined with table `orders` first, since the local predicate on table `part` is more selective than that on table `orders`. Some of the queries have little or no improvement, because the underlying tables are already partitioned in the optimal way. Although it's hard to discern in Table 2(a), a couple of queries had slightly higher costs when evaluated in RECOMMEND PARTITION mode. This is caused by some heuristic rules

used by DB2's query optimizer. The optimizer favors local and directed joins over repartitioned joins. If two subplans can be joined through local or directed joins, the optimizer won't even try the repartitioned joins. While such heuristic rules are justified in most cases, occasionally repartitioned joins can be cheaper. So, when using the real partitions, these queries are forced to consider repartitioned join plans. When evaluated in RECOMMEND PARTITION mode, the optimizer only picked up partitions that can form local or directed joins (which are in fact a little more expensive).

In Table 2(b), we show the number of partitions recommended for each table. These are the partitions that are the best for at least one query in the workload. Although each table gets relatively few recommendations, the total number of possible configurations already reaches 1728. For commercial workloads with thousands of base tables, the total number of configurations will be too large to try using brute-force enumeration. The recommended partitions and their corresponding benefit values for the two largest tables – `lineitem` and `orders` – are presented in Table 2(c) and Table 2(d) respectively. Most partitioning keys are either join columns or group-by columns. All the partitions are chosen to be performed in nodegroup 1, consisting of all 8 nodes in the system. As we can see, replication is among the recommended partitions for both tables. For table `lineitem`, one of the non-replicated partitions has no partitioning key. This is because the columns in the partitioning key are not referenced in any query, and thus the partition normalization routine (described in Section 4.3) reduced the number of partitioning columns to zero. The negative benefit for one partition for `lineitem` is due to the heuristic rule used in the optimizer, described earlier.



(a) whole picture (b) close-up

Figure 4: Linear and Diagonal Enumeration

Next, we compare our linear and diagonal enumeration algorithms. The result is shown in Figure 4. The x-axis represents the number of configurations considered by each algorithm, and the y-axis measures the best estimated cost for the entire workload. The line marked as "initial cost"

represents the estimated cost of the entire workload under the original (real) partitions. We add up the cost of each query in RECOMMEND PARTITION mode and use it as the "lower bound", since this is the lowest cost the workload could achieve, but may never actually be achievable in a bona fide configuration. In Figure 4(a), the best configurations found by both methods reduce the workload cost by 35% below the best human configuration. In our test, the best solution found by diagonal enumeration is in fact the "optimal" solution when we tried all the configurations. The initial configurations considered by both enumeration methods have costs close to the optimal. This suggests that the benefit calculated for each recommended partition is a good measure of its relative contribution to the workload. Further iterations only improve the workload performance marginally. However, we can still see the difference between linear and diagonal enumeration from Figure 4(b). Diagonal enumeration converges at a much faster rate than linear enumeration. Although this is relatively insignificant for the TPC-H workload (since the initial configuration is good enough), there could be a big difference between the two in larger workloads.

table	linear enumeration	diagonal enumeration
region	r_regionkey	r_regionkey
nation	REPLICATED	n_nationkey
part	p_partkey	p_partkey
partsupp	ps_partkey	ps_suppkey
lineitem	l_orderkey	l_partkey
orders	o_custkey	o_custkey
supplier	s_suppkey	s_nationkey
customer	c_custkey	c_custkey

Table 3: Best Partition Configurations

We compare the best configurations found by both enumeration methods in Table 3. The most important change from the original configuration is that table `orders` is now partitioned on `o_custkey`. This reduces the cost of a query consisting of a join between table `orders` and table `customer`, and this query's high cost weights it heavily in the workload. From Table 2(d), we can see that the partition with `o_custkey` as the partitioning key has the highest benefit value, and thus will be picked up in the first configuration considered by both enumeration algorithms. Table `lineitem` has two conflicting partitions: `l_orderkey` and `l_partkey`. While the former allows a directed join between `lineitem` and `orders`, the latter can make the join between `lineitem` and `part` a local join if `part` is also partitioned in the appropriate way. Choosing either partition will increase the cost of some queries while lowering the cost of some others. However, this won't

make too much difference to the total cost of the workload. Similar tradeoffs are found for table partsupp. The rest of the tables are relatively small and their partitions won't affect the workload cost significantly.

We also tested our genetic algorithm on the workload. Using the method described in Section 7.3, the initial population contains a configuration that's close to optimal. Consequently, we didn't see further cost reduction using various combinations of mutation and crossover. So, we omit our results for the genetic algorithm. We plan to test the algorithm on some other commercial workloads in the future.

To summarize, our experimental results validate the usefulness of our Partitioning Advisor. It can recommend partitions that can't be easily identified by human beings. The benefit values we calculated serve as good guidance to our enumeration process and cut down the search space considerably without sacrificing the quality of the chosen partitions.

9 Conclusion

In this paper, we described DB2's Partitioning Advisor, a tool that can automate the process of choosing the optimal way to partition data stored in a shared-nothing parallel system, for a given workload, exploiting the cost-based query optimizer to both recommend likely candidates and to evaluate complete solutions in detail. Our work has been prototyped in a commercial version of DB2 and is fully functional. We detailed the changes needed on the server side, and proposed and evaluated efficient enumeration algorithms. Our performance evaluation found that the Partitioning Wizard could even improve the partitioning decisions of expert humans by 35%. Our work is the first among major commercial database systems. In the future, we plan to expand our self-managing wizards to other kinds of database design problems, and to investigate the interactions among different database design issues.

References

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 496–505, 2000.

[BFG+95] C. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padimanabhan, G. Copeland, and W. Wilson. Db2 parallel edition database systems: The future of high performance database systems. *IBM Systems Journal*, 34(2), 1995.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of the ACM SIGMOD Conference*, pages 99–108, 1988.

[Car75] A. Cardenas. Analysis and performance of inverted data base structures. *Communications of ACM*, 18(5):253–263, 1975.

[CN98] Surajit Chaudhuri and Vivek R. Narasayya. Microsoft index tuning wizard for SQL server 7.0. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 553–554, 1998.

[CNW83] Stefano Ceri, Shamkant B. Navathe, and Gio Wiederhold. Distribution design of logical database schemas. *TSE*, 9(4):487–504, 1983.

[Cor00a] IBM Corporation. DB2 Universal Database enterprise extended edition (EEE) Version 7.0. 2000.

[Cor00b] Informix Corporation. http://www.informix.com/informix/solutions/dw/redbrick /vista. 2000.

[Cor00c] NCR Corporation. Teradata server. 2000.

[Cor00d] Oracle Corporation. Oracle 9i database. 2000.

[Cor00e] Sybase Corporation. Adaptive enterprise server 12.0. 2000.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of ACM*, 35(6):85–98, 1992.

[Gha90] S. Ghandeharizadeh. *Physical Database Design in Multi-processor Systems*. PhD thesis, Department of Computer Science, University of Wisconsin-Madison, 1990.

[GLSW93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the DB2 family. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(4):4–18, 1993.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, INC, 1989.

[HLL94] Kien A. Hua, Sheau-Dong Lang, and Wen K. Lee. A decomposition-based simulated annealing technique for data clustering. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–128, 1994.

[HS98] Peter J. Haas and Lynne Stokes. Estimating the number of classes in a finite population. *Journal of the American Statistical Association*, 93(444):1475–1487, 1998.

[IK91] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 168–177, 1991.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and Jr. M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th VLDB Conference*, pages 314–325, 1990.

[RM93] Erhand Rahm and Rober Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *Proceedings of the VLDB Conference*, pages 182–193, 1993.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomsa G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*, pages 23–34, 1979.

[SW85] Domenico Saccà and Gio Wiederhold. Database partitioning in a cluster of processors. *TODS*, 10(1):29–56, 1985.

[TPC] TPC benchmark H (decision support) revision 1.1.0. http://www.tpc.org/.

[VZZ+00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the ICDE Conference*, pages 101–110, 2000.

[Zil98] Daniel C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, Department of Computer Science, University of Toronto, 1998.

A Additional Algorithms

Algorithm A.1: Linear Enumeration

Input:	nTable, number of tables
	table[1..nTable], an array of table structure
	table.nParts, number of recommended partitions in the table
	table.parts[1..nParts], a list of recommended partitions for the table
Output:	table.curPart, the index of the partition for the table in the current configuration

```
init_configuration()
{
 for i= 1 to nTables
   table[i].curPart = 1;
}

int next_configuration()
{
 i=1;
 table[i].curPart++;
 while (table[i].curPart > table[i].nParts)
 {
   table[i].curPart=1;
   i++;
   if (i>nTable)
     return 0; //end of iteration
   table[i].curPart++;
 }
 return 1;
}
```

∎

Algorithm A.2: Diagonal Enumeration

Input:	nTable, number of tables
	table[1..nTable], an array of table structure
	table.nParts, number of recommended partitions in the table
	table.parts[1..nParts], a list of recommended partitions for the table
	table.curNParts, number of recommended partitions considered for the current iteration in the table
	expTab, the index of the table whose dimension is to be expanded
Output:	table.curPart, the index of the partition for the table in the current configuration

```
init_configuration()
{
  for i:= 1 to nTables
    table[i].curPart = 1;
    table[i].curNPart = 1;
}

int expand_dimension()
{
  old_expTab=expTab;
  do {
    expTab++;
    if (expTab > nTables)
      expandTable=1;
  }
  while (table[expTab].curNParts == table[expTab].nParts AND expTab!=old_expTab);
  if (table[expTab].curNParts < table[expTab].nParts)
  {
    table[expTab].curNParts++;
    table[expTab].curParts=table[expTab].curNParts;
    table[old_expTab].curParts=1;
    return 1; //successfully expanded the dimension
  }
  return 0; //no more dimensions to expand
}

int next_configuration()
{
  i=1;
  if (i==expTab)
    i++;
  table[i].curPart++;
  while (table[i].curPart > table[i].nParts)
  {
    table[i].curPart=1;
    i++;
    if (i==expTab)
      i++;
    if (i>nTable)
      if (expand_dimension())
        return 1;
      else
        return 0; //end of iteration
    table[i].curPart++;
  }
  return 1;
}
```
▮