Automating Physical Database Design in a Parallel Database

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 data is horizontally partitioned among multiple independent nodes. We seek to automate the process of data partitioning. Given a workload of SQL statements, we seek to determine automatically how to partition the base data across multiple nodes to achieve overall optimal (or close to optimal) performance for that workload. 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.

We present a comprehensive solution to the problem that has been tightly integrated with the optimizer of a commercial shared-nothing parallel database system. 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. We compare a rank-based enumeration method with a random-based one. Our experimental results show that the former is more effective.

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, Microsoft, NCR, Oracle, Sybase, 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. 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. We build a tool called *partition advisor* to automate the process of partition selection 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 a given workload. Our approach therefore avoids redundancy – and possible inconsistency – between the partition advisor and the query optimizer. Our partition advisor has been developed in IBM Universal Database for Unix, Windows, and OS/2, Enterprise Extended Edition (referred to as DB2 in the rest of the paper), and has been tested comprehensively on both benchmark and real customer data.

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 give some background information on DB2 in Section 4. We then describe our optimizer extension, cost estimation, and components on the client side in Sections 5, 6, 7, respectively. Our experimental results are presented in Section 8. We discuss usability issues in Section 9 and conclude in Section 10.

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, SMR00]. 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. Work in this area started as early as 1988 [FST88], in which the authors proposed to use the optimizer to evaluate the goodness of index structures. The Comfort automatic tuning project [WHMZ94] has investigated architectural principles of self-tuning database and developed self-tuning methods for specific performance tuning problems. 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 [Cor00c] have similar projects of building 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 that store redundant data, i.e., in addition to the base table. 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 previous algorithms for selecting auxiliary structures cannot be applied directly to the selection of optimal partitions.

3 Overview of Our Approach

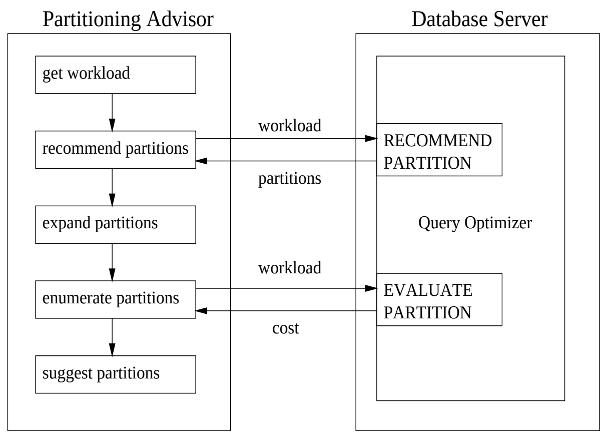


Figure 1: Architecture of the Partition Advisor

Given a workload of SQL statements and their frequency of occurrence, we want to determine automatically the optimal (or close to optimal) way of partitioning the data so that the overall workload cost is minimized. We use the cost estimates of the optimizer as our metrics. Figure 1 shows the architecture of the partition advisor in DB2. A similar architecture has been used for other kinds of advisors in DB2.

We first describe the changes we made at the database server end. Given a query, an optimizer will normally generate alternative plans based on various physical properties (partition, indexes, etc) the underlying tables have. We refer to this query optimization process as the *regular* mode. We augment DB2's optimizer with two additional modes: RECOMMEND PARTITION and EVALUATE PARTITION. For the sake of simplicity, we will refer to them as *RECOMMEND* and *EVALUATE* mode respectively in the rest of the paper. In RECOMMEND mode, we rely on the optimizer to recommend good candidate partitions for each statement. Previous work [FST88] generates candidate partitions outside the engine, and thus has to make multiple calls to the optimizer for each candidate. When evaluating a query in RECOMMEND mode, the optimizer accumulates a list of partitions for each table that are potentially beneficial to processing of that query and generates plans corresponding to each of these (virtual) partitions. Optimization then proceeds normally to evaluate all of these alternative plans. 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 a CANDIDATE_PARTITION table. In EVALUATE mode, the optimizer first reads from the CANDIDATE_PARTITION table those "marked" partitions and uses them to replace the real partition for the corresponding table. After that, the optimizer optimizes the query, assuming that the tables are partitioned in the newly-specified way. When a query is optimized in either RECOMMEND or EVALUATE mode, the query plan is generated without being executed.

On the client side, the partition advisor is built as an application tool. After getting a workload, it invokes the optimizer to evaluate all the statements in the workload in RECOMMEND mode. The advisor then collects all the candidate partitions (best for each individual statement) from the CANDIDATE_PARTITION table. Subsequently, it performs partition expansion to generate additional candidate partitions that might have been missed by each individual statement. Finally, an enumeration algorithm will combine candidate partitions from different tables in certain ways and evaluate the workload in EVALUATE mode for each combination. In the end, the advisor will report to the user the best partition that was chosen for each table and the corresponding cost for the workload.

Our tool can be used to decide either database design initially or whenever a major reconfiguration of the database occurs. The applications of the partition advisor include (but are not limited to) the following:

- loading a prospective database
- migrating a database to a different platform or a different vendor
- the workload on a database changes substantially
- new tables are added or the database has been heavily updated
- database performance has degraded

One important aspect of our partition advisor is our materialized view support (described in Section 5.1). Besides base tables, our tool can recommend partitions for materialized views if they are used by the workload. This is useful as there are now tools [ACN00] that help to determine what materialized views to create for a given workload automatically. Without proper recommendations for partitioning, those recommended materialized views will typically use some default partition, and so won't be able to achieve their full benefit.

Our architecture assumes that we have some statistics on the database for the query optimizer's cost estimation. Statistics can be collected when data has already been loaded in the system. If data has not been loaded, the designer should supply a statistics description file. DB2 provides a utility that generates such a description file by scanning through external data. Our technique also requires that there is already an original ("real") partition for each table, but these partitions can be picked arbitrarily, e.g. the default partitions assigned by DB2.

4 Background

In this section, we first briefly describe DB2's parallel database system. We then introduce the optimizer used in DB2 and the concept of "interesting" partitions.

DB2 Parallel Database System: DB2 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. 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 allows multiple *nodegroups* to be defined. A nodegroup can be assigned to any subset of the nodes in a system. A table can be partitioned among all nodes in a nodegroup, or can also be replicated across all the nodes in a nodegroup. 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. When creating a table, users can specify a partition for the table. Otherwise, a default partitioning key (the first column) and default nodegroup (including all nodes in the system) will be used.

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 the 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 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 *repartitioned 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.

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 a parallel environment, DB2 also pre-computes beneficial partitions for a query as its "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 plan generation starts, and are accumulated and mapped to each participating base table in the query.

5 Optimizer Extension

In this section, we describe in detail the extensions we have made to the optimizer. In RECOMMEND mode, our goal is to determine good candidate partitions for each table in each individual statement. For a given SQL statement, our approach only recommends one best candidate partition for each table referenced by the query. By doing so, it's possible that we will miss the overall best partition for the workload. However, keeping the top K ($K > 1$) best candidates can be quite expensive, as we have to keep at least the top K subplans for every possible join combination. Additionally, the optimizer's infrastructure would need to be changed significantly. Instead, we try to recover some of the missing candidate partitions on the client side through partition expansion (described in Section 7.1).

In the following section, we first describe our approach to generating candidate partitions in RECOMMEND mode. We also discuss our support for materialized views and candidate partition reduction. We describe EVALUATE mode in Section 5.2.

5.1 Recommend Partitions

The goal in RECOMMEND mode is to find the optimal partition for each base table for a given query. One possible way is to generate a base table plan (a table scan plan or an index plan) for each possible partition. However, since the number of possible partitions can be large (every subset of columns can be used as partitioning keys), this approach is prohibitive. We observe that not every possible partition of a table is beneficial for a query. So, we first compute for each base table a list of candidate partitions, each of which can help reduce the cost of the query.

Interesting partitions (as described in Section 4) are certainly candidate partitions for each base table, as each might benefit some operations in that query. However, there are more candidate partitions than interesting partitions. Consider a local equality predicate of the form *col* = *constant*. If the table is partitioned on *col*, then all the rows satisfying this predicate are stored on a single node. Although this concentrates all the processing on a single node rather than spreading it out, it reduces communication cost and can be a winning partition. This partition will not be considered as interesting since local predicates are always applied before join predicates. So in RECOMMEND mode we will generate candidate partitions for every column bound to a constant by an equality predicate. Note that for systems supporting range partitioning, columns referenced in non-equality predicates might also be beneficial partitioning keys.

Another kind of candidate partition is replication. Replicating a table across all nodes reduces communication cost and can potentially improve query performance. However, since replication has storage overhead, it's probably not a good idea to replicate very large tables. Thus, for each table, we add to its candidate partition list a replicated partition only if the table size is smaller than a threshold.

We also have to consider another factor that determines a partition—the nodegroup. For each candidate partitioning key generated, we have to decide which nodegroup to use. One possibility is to use the default nodegroup for the query. 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 automatically 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 candidate partitioning keys.

Once the candidate partition lists have been generated, we augment the optimizer 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. The optimizer then continues its usual join enumeration process (with more choices on base table plans). When the optimizer returns the best overall query plan, it writes to a CANDIDATE_PARTITION table the best partition chosen for each table in that plan.

One 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 conflicting partitions. Such a 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.

Support for Materialized Views: Materialized views are cached query results and can be used to improve query performance dramatically. As for base tables, finding the right partitions for materialized views is essential for query performance.

We can assemble candidate partitions for a materialized view by collecting all candidate partitions from each table referenced in the materialized view. However, some of the candidate partitions may no longer be interesting because the predicates that induce them have already been retired by that materialized view. For example, suppose we have a materialized view and a query defined as follows:

```
Materialized View M1:
SELECT * FROM T1, T2 WHERE T1.a = T2.a
Query Q1:
SELECT * FROM T1, T2, T3
WHERE T1.a = T2.a AND T2.b = T3.b
```

Note that $M1$ can be used to answer $Q1$ by joining it to table $T3$. A partition with $T2.a$ as the partitioning key is a candidate partition for table $T2$. However, such a partition is not useful for $M1$ as the join predicate $T1.a = T2.a$ has already been applied within the materialized view. As a result, this partition shouldn't be considered as a candidate partition for $M1$. On the other hand, a partition with $T2.b$ as the partitioning key should be a candidate partition for $M1$ since the predicate $T2.b = T3.b$ has yet to be applied. So for materialized views, we make sure that each candidate partition is still useful for some predicates not yet applied by the materialized view or useful for aggregation and ordering. In the rest of the paper, we treat materialized views in the same way as base tables.

Candidate Partition Reduction: When a query is compiled in regular mode, plans with different partition properties are limited. However, in RECOMMEND mode, the optimizer will create plans having a partition property for each candidate partition. This can introduce an explosion in the number of plans generated, and thus increase both the compilation time and space consumption (to keep those plans).

We aim to reduce the search space in RECOMMEND mode by limiting the number of candidate partitions considered, without sacrificing the quality of plans too much. First of all, we observe that if two nodegroups have the same set of nodes, then for the same set of partitioning keys, we only need to consider one of those nodegroups. Similar nodegroups can exist in a system, because customers often define a new nodegroup including all the nodes in the system, the same as the default nodegroup. Thus, we check if there is a user-defined nodegroup identical to the default nodegroup. If so, we won't generate candidate partitions in the default nodegroup. Second, we notice that for a single-node nodegroup, it doesn't matter what the partitioning key is, as all the data will reside on one node. So, we make sure that for such nodegroups, we only consider one candidate partition for each table. Last, we realize that for very small tables, different partitions only slightly affect the final plan cost. As a result, we can just use the original partition for such tables. We show in our experimental results in Section 8 that, by doing all the above, the compilation time in RECOMMEND mode is reduced significantly, without appreciable loss of plan quality.

5.2 Evaluate Partitions

Before compiling a query in EVALUATE mode, certain partitions in the CANDIDATE_PARTITION table are already marked by our enumeration methods in the client utility (described in Section 7). Our enumeration methods guarantee that only one partition is marked for each table. In EVALUATE mode, the optimizer then reads in those marked partitions from the CANDIDATE_PARTITION table and uses it to replace the original partition of the corresponding table before optimization starts. The optimization then continues under the assumption that those replaced partitions are the real partitions for tables referenced in the query.

The compilation time for a query in EVALUATE mode is comparable to that in regular mode, as each base table can still generate only one possible partition. The overhead of injecting virtual partitions for each table is very little.

6 Cost Estimation

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

DB2 Cost Model: DB2 uses a detailed cost model to estimate query cost. The overall cost is a linear combination of I/O cost, CPU cost, and communication cost, but also assumes 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, index statistics, and optionally, distribution statistics such as histograms and a list of the most frequent values.

There are two kinds of statistics, one at the table level and one at a single node level (we refer to them as *per-table* and *per-node* statistics, respectively). Both sets of statistics 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 systems 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 estimates independent of how the data is partitioned. After repartitioning, DB2 will derive per-node statistics from the per-table ones based on how the data is partitioned.

Estimating Cost Under New Partitions: Observe that while per-table statistics are independent of partitions, per-node statistics will change if the underlying partition changes. In this section, we will discuss how to adjust per-node statistics with new table partitions in both RECOMMEND and EVALUATE mode.

There are basically two options: (a) use sampling, and (b) derive from the old statistics. There are trade-offs between the two. Sampling provides more accurate information, however, it can be very expensive given the number of candidate partitions that we want to simulate (even after partition reduction). Deriving new statistics always requires some assumptions that can be different from reality. Nevertheless, deriving statistics is much cheaper. Additionally, the optimizer itself has to derive some statistics when repartitioning for joins and aggregations anyway. So, in the first phase of the partition advisor, we opt for deriving statistics.

We are only interested in adjusting statistics that affect base table plans. We distinguish between statistics associated with a table (we call them *table statistics*) or an index structure (we call them *index statistics*). We illustrate our approach to adjusting these two kinds of statistics.

Table statistics that need to be adjusted include cardinality and number of pages. We make the assumption that data is uniformly distributed across all the nodes (consistent with the reason for using hash partitioning). We calculate the ratio between the number of nodes in the old and new partitions, and scale the statistics accordingly.

Typical index statistics include number of leaf pages and number of levels. An index leaf page contains a sequence of (unique) keys, each of which has one or more row IDs (*RID*). Both of these can change given a new table partition. The number of *RIDs* per node can be adjusted in a fashion similar to the per-node cardinality that was described earlier. We use a standard distinct values estimator (Cardenas' and Inverse Cardenas' formula [Car75]) to first get the per-table key count and then scale it down to per-node under the new partition. Once we have obtained the new per-node key and *RID* counts, we use them to estimate the new number of leaf pages assuming that the key size and page occupancy rate are still the same. The number of index levels can be adjusted based on the new number of leaf pages assuming the same fanout. A detailed description of our adjustment is beyond the scope of this paper.

Avoid Data Skew: When estimating new statistics, one of the assumptions we made is a uniform distribution. Clearly, such an assumption won't hold if data is skewed. We observe that there are two cases where skew can exist: (a) when there are very few key values in the partitioning key, and (b) when hash buckets are not distributed evenly by the mapping function. The latter is alleviated by the ability in DB2 to define alternative mappings from the hash buckets to nodes (a level of indirection) [Cor00a]. To avoid (a), in RECOMMEND mode, we check the key count of each candidate partition and only consider partitions having enough key values (more than a threshold). The threshold is proportional to the number of nodes in the system.

7 Advisor on the Client Side

Given a workload, our partition advisor will evaluate each query in RECOMMEND mode. After reading all the partitions from the CANDIDATE_PARTITION table, it assembles a list of candidate partitions for each table referenced in the workload. In this section, we first describe how to generate additional candidate partitions through partition expansion and then describe our enumeration methods.

7.1 Partition Expansion

Each partition in the CANDIDATE_PARTITION table is the best partition for at least one statement in the workload. However, a partition that's not the best of any query could be the overall best partition for the workload. Consider a simple example. Suppose that the partitioning key of the best partition for table T is $< T.a, T.b >$ for query 1 and $< T.a, T.c >$ for query 2 (in the same nodegroup). $< T.a >$ could be the best partitioning key for query 1 and 2 together, as it can benefit both queries, but neither query chooses that partition.

We say that a partition $P1$ *subsumes* $P2$ if $P1$ and $P2$ are in the same nodegroup and the partitioning key of $P1$ is a superset of $P2$. In general, if partition $P1$ subsumes $P2$, $P2$ will benefit at least as many queries as $P1$ does. The only exception is if $P2$ has too few key values, which will introduce potential skew. During partition expansion, we will generate additional candidate partitions for each table with partitioning keys that are a common subset from two or more candidate partitions from the same table, as long as the number of key values is greater than the threshold. We will also include the original partition in the candidate partition list if it wasn't recommended for any query in RECOMMEND mode.

7.2 Enumerating Algorithms

The combinations of partitions from all the tables form a search space. Suppose that we have n tables, each with p_i (i = 1 to n) candidate partitions. We define a *configuration* $C = (c_1, c_2, ...c_n)$, where c_i is one of the p_i candidate partitions from table i. For a statement q, let $Cost_q(C)$ be the cost of q under partition configuration C. Given a workload Q, we want to find $C_{optimal}$ such that $\sum_{q \in Q} Cost_q(C_{optimal}) = min \sum_{q \in Q} Cost_q(C)$, over all p_1*p_2...*p_n possible C.

Given a configuration C, we can mark all the candidate partitions in C in the CANDIDATE_PARTITION table and compile the workload in EVALUATE mode. The cost estimation returned by the optimizer will be the cost of that configuration. Our goal is to quickly find the optimal configuration, so that the entire (weighted) workload cost is minimized. Each evaluation of a configuration is expensive, so unguided enumeration of all configurations won't scale with respect to the number of tables and workload size.

To facilitate our search, we first calculate a benefit value for each candidate partition in each query, which equals the difference between the estimated cost of the query evaluated in regular mode and in RECOMMEND mode. We then accumulate over all queries in the workload the total benefit for each distinct candidate partition. Subsumed partitions (including those additional partitions generated during expansion) will inherit the benefit of all the subsuming partitions. Observe that the benefit we have accumulated is not the real benefit each candidate partition will bring. This is because we assign the benefit for the whole query to the underlying partitions of all tables in that query, while in fact, some partitions may contribute more to the benefit than others. Trying to figure out the exact benefit of each candidate partition is difficult because the benefit of one partition may not show up unless another partition is present. Nevertheless, our experimental results show that our simplified benefit estimation provides good search guidance. With the help of benefit values, we consider two kinds of enumeration methods: rank-based and random-based enumeration.

7.3 Rank-based Enumeration

The rank-based method models the problem as a general searching problem. Each configuration corresponds to a node in the search tree. We start with the *root* node, which corresponds to the configuration with each table using the partition having the highest benefit value among its candidate partitions. To expand a node C, we consider all the configurations (referred to as *child configurations*) that differ from C in exactly one partition. The different partition has the next highest benefit value. Observe that if the benefit values are reliable, a parent configuration should always be better than a child configuration for the workload and thus should be considered earlier than its children during the search. All the expanded nodes will be ranked and kept in an ordered queue. The first configuration (with the highest rank) in the queue will be the next search point. The enumeration process stops when we reach a user specified time limit. The key issue here is to design a good ranking function to guide our search.

A simple ranking function (referred to as *rank_benefit*) will be the sum of the benefit of each partition in the configuration. However, it doesn't perform well in our experiments. The problem is illustrated in the example in Table 1. In this example, a configuration with partitions $(P1, P3)$ will be considered first. However, a configuration with partitions $(P2, P3)$ will be considered better than that with $(P1, P4)$ as the former has a higher total benefit value. However, $P4$ carries a much higher benefit value than $P2$ and is likely to be more important.

T1	Partition	Benefit	T2	Partition	Benefit
	P1	10		P3	1000
	P2	9		P4	900

Table 1: Example

To overcome the above problem, we take into account three factors when designing our ranking function: (1) the cost of its parent configuration (2) the benefit of the partition that's different from its parent, and (3) the size of the table from which the different partition comes. (1) is important because child configurations share a lot of partitions with their parents. So the cost of a parent should have some influence on its children. Note that the cost of a parent configuration is accurate in the sense that it's the optimizer's estimation. (3) is important as partitions from bigger tables tend to contribute more to the workload than smaller ones. Through experiments, we choose the following ranking function (referred to as *rank_best*). We assign the cost of a configuration to be the cost of its parent less the benefit of the changed partition, weighted by the relative table size. Since a configuration can be derived from multiple parents, we are being optimistic and always pick the higher rank for the configuration.

$rank\_best(C) =$
$-(Cost(C') - P.benefit * \sqrt{\frac{P.tblcard}{max\_tblcard}})$, where C' is the parent configuration of C, P is the partition in C different from C' and $max\_tblcard$ is the size of the largest table in the workload.

To evaluate a configuration, we have to compile each statement in the workload in EVALUATE mode. If the number of statements in the workload is large, evaluating a configuration can be quite expensive. Observe that a query may not reference all the tables in the workload. We define partitions in a configuration used by a query its *footprint*. The cost of a query Q will be the same in EVALUATE mode under two configurations $C1$ and $C2$, if the footprints of Q in $C1$ and $C2$ are the same. To avoid reevaluating statements with the same footprint, we cache the cost of each query under each of its unique footprints. Before evaluating a query under a certain configuration C, we first check if the query with its footprint in C can be found in the cache. If so, we can use the cached cost directly. Otherwise, we will send the query to the server and evaluate its cost.

7.4 Randomized Enumeration

We also considered using 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* [Gol89], as they have some good searching features (such as avoiding local optima) and can be mapped to our problem easily.

Genetic algorithms are search algorithms based on the mechanics of genetics and natural selection. 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* consisting of a set of species and tries to derive better search points by evolving next generations. There are typically two ways to evolve—*crossover* and *mutation*. Crossover takes two species from the population and randomly recombines their genes to form two descendants. The intuition here is that good parents are likely to provide better children. Mutation picks a species at random and randomly changes some of its genes to derive a descendant. Descendants are compared with their parents and will replace the parents if they are better.

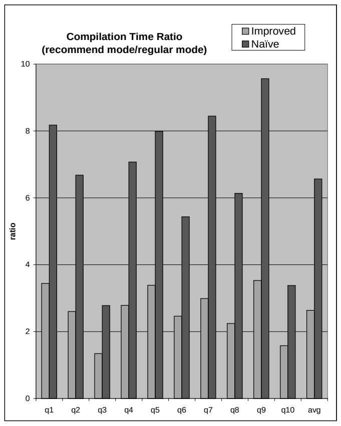
In order for the genetic algorithm to perform well, we need to set up a good initial population. Through experiments, we used the best partitions from queries with large improvement (for those tables not referenced by the query, partitions with the highest accumulated benefit are used). We also include the root configuration in our initial population. However, our experiments showed that the genetic algorithm is always dominated by our rank-based method.

8 Experimental Results

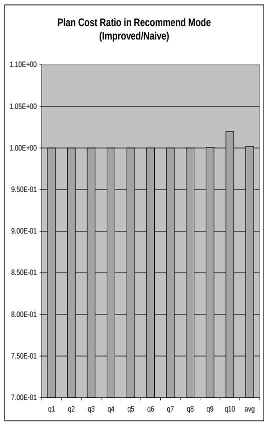
We developed our partition advisor under DB2. All the tests were run on a machine with two 400 MHz processors and 1GB of RAM. DB2 allows us to create multiple virtual nodes, all running on the same machine.

We have performed comprehensive tests of the partition advisor on several workloads. However, due to space limitations, we only present our experimental results on a 100GB TPC-H [TPC] database and one customer database. We simulated an environment with 8 nodes, each with a 1.4 GHz processor, a 500MB buffer pool, and a 100MB/second communication bandwidth. In both databases, there were two additional nodegroups besides the default nodegroup: one with all nodes and another with a single node. We omitted 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. To avoid a small number of expensive queries dominating the workload cost, we adjusted the frequency of each statement such that the weighted (initial) cost of each statement is roughly equal. The threshold of small table size for partition reduction, maximal table size for replication and minimal number of key values necessary for a candidate partition are set to be 2,000 (rows), 5 million (rows) and 5,000 respectively. Because of the size of the databases we used, it's difficult to create a local copy of the real data. DB2 provides a tool that can collect the catalog and statistics in a real database and generate a description file. By running those SQL statements from the description file in an empty database, we can re-create the metadata of the real database without populating the actual data.

Compilation Time in RECOMMEND mode: We first present the results of RECOMMEND mode. We selected the 10 most complicated queries (with up to hundreds of joins in multiple query blocks and lots of aggregation) from a set of our customer queries. We compared the compilation time in RECOMMEND mode with (referred to as *improved*) and without doing partition reduction (referred to as *naive*) as described in Section 5.1. We report the ratio of compilation time in RECOMMEND mode to that in regular mode in Figure 2(a). As we can see, our partition reduction techniques cut the average compilation time by more than half. Yet Figure 2(b) shows that partition reduction doesn't significantly affect the quality of the plans we got in RECOMMEND mode: only one of the ten queries has a plan with slightly higher cost after partition reduction.



(a)



(b)

Figure 2: Compilation Time Ratio (a) and Plan Cost Ratio (b) in RECOMMEND Mode

TPC-H: We then present our results on the TPC-H database with 22 TPC-H queries. The limited number of tables and statements in the workload don't adequately test our enumeration methods, as there are not that many configurations. On the other hand, because of its simplicity, we can perform a more thorough analysis and gain some insights.

Table 2(a) 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	nodegroup	# 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

(a) Initial partitions

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
0.0	8.0	0.0	0.0	-0.4	0.0	-0.4	0.0
Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
-0.4	-0.2	0.0	0.0	8.0	0.0	0.0	-0.4
Q17	Q18	Q19	Q20	Q21	Q22		
31.4	0.0	0.0	0.0	-0.6	577.0		

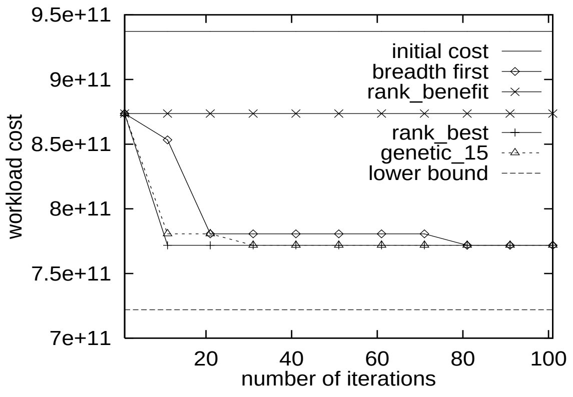
(b) Query Speedup(%)

table	benefit	NG	R	partition key	
lineitem	68,489.57	1	N	l_partkey	
	-3,567.18	1	N	l_orderkey	×
	-7,163.52	1	N	l_suppkey	
nation	23,393.15	2	N		×
region	32,257.48	2	N		×
orders	124,350.79	1	N	o_custkey	×
	-8,268.93	1	N	o_orderkey	
partsupp	15,570.06	1	N	ps_partkey	×
	10,798.27	1	N	ps_suppkey	
part	49,947.60	1	N	p_partkey	×
supplier	24,307.95	1	N	s_suppkey	×
customer	123,743.82	1	N	c_custkey	×
	484.03	1	Y		

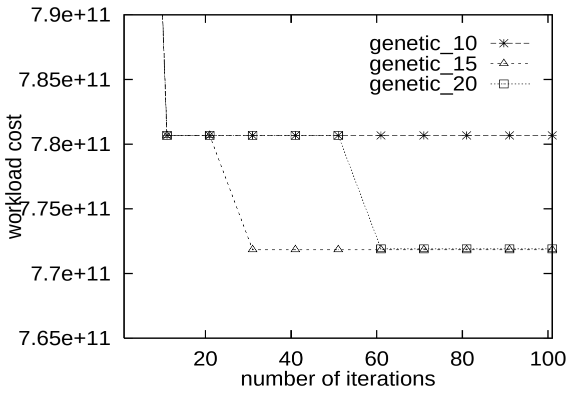
(c) Candidate Partition List

Table 2: Results on TPC-H

Table 2(b) shows the speedup as a percentage ($= \frac{initial\_cost - recommend\_cost}{recommend\_cost} * 100$) for each individual query in RECOMMEND mode. Some of the queries had little or no improvement, because the underlying tables were already partitioned in the optimal way. Query 22 had the biggest improvement. 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 thus its cost is reduced significantly. Query 17 also had significant speedup. This query contains a join between `lineitem` and `part`. In RECOMMEND mode, the optimizer chose to partition table `lineitem` on `l_partkey` so that a local join could be used instead of a directed join. A



(a) All Methods



(b) Genetic Algorithm

Figure 3: Cost Improvement as a Function of Number of Iterations

few queries had slightly higher costs when evaluated in RECOMMEND mode (negative speedup). This is caused by the heuristic rules of DB2's query optimizer that 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 slightly cheaper. When using the real partitions, these queries are forced to consider repartitioned join plans. When evaluated in RECOMMEND mode, since the optimizer has more partition choices, it picks up partitions that can form local or directed joins (which are in fact a little bit more expensive). However, the degradation is relatively small (no more than 0.6%), and all the configurations are subject to evaluation in EVALUATE mode to verify the workload cost.

In Table 2(c), we show the candidate partition list for each table (each row indicating the benefit of a partition, its nodegroup, whether it's replicated and its partitioning key). As we can see, most of the dimension tables have only one candidate partition, with the primary key as the partitioning key. For fact tables, there are more candidate partitions, each of which has one of the foreign keys as the partitioning key. Note that for the `customer` table, one of the candidate partitions is to replicate the table on all nodes. So replication does help in certain cases. Since there are only 24 possible configurations, it's possible for us to enumerate them all. The best configuration (partitions marked with $\times$) chosen by the partition advisor uses the original partition for all the tables except for one—`orders`, which is now recommended to be partitioned on `o_custkey` across all nodes. This reduces the cost of queries consisting of a join between table `orders` and table `customer`. Observe that the benefit value we calculated for each candidate partition is a relatively good indicator of its goodness. Every best partition except for one has the highest benefit value among candidate partitions of its corresponding table. Only the best partition of table `lineitem` has the second highest benefit value. The overall improvement of the workload is about 4%. This shows that the original partitions chosen by human beings are fairly good for this relatively simple workload.

Customer Database: We now present the results of applying the partition advisor to one of our customer databases. We used a workload consisting of 50 queries. There are 15 tables referenced in the workload. After partition recommendation, each table had from 1 to 5 candidate partitions. The total number of configurations is around 500. We compared the rank-based method with the genetic algorithm on this larger workload. For the rank-based method, we tested the two ranking functions *rank_benefit* and *rank_best*, as described in Section 7.3. We also tested a *breadth_first* method, where configurations with lower depth are considered earlier. For genetic algorithm, we tested three variants with an initial population size of 10, 15 and 20. We alternate the process of crossover and mutation. The mutation rate (the percentage of genes to be changed) is set to be 0.5.

The results are shown in Figure 3(a). The x-axis represents the number of configurations (iterations) considered by each algorithm, and the y-axis measures the best workload cost found after a certain number of iterations have been performed. The line marked as "initial cost" represents the estimated cost of the entire workload under the original (real) partitions. We added up the cost of each query in RECOMMEND mode and used it as the "lower bound", since this is the lowest cost the workload could theoretically achieve, but may never actually be achievable in a bona fide configuration. We only show the best genetic variant in this picture. For a fair comparison, we allowed each method to consider 100 configurations.

As we can see, *rank_best* converges the fastest among all the methods. It is able to find a very good solution

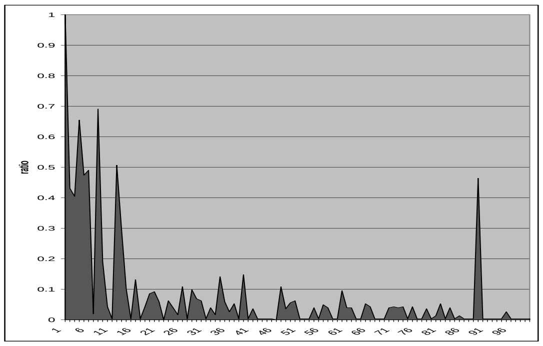


Figure 4: Relative Time per Iteration

at the 10th iteration and the optimal solution (verified after we tried all configurations) after only 26 iterations. The speedup is more than 22% on a system already tuned by human beings. *rank_benefit* doesn't improve upon the root configuration within 100 iterations. This is because it tends to try partitions with relatively small benefit values first. Genetical algorithm converges faster than *rank_benefit* and *breadth_first*. The initial population we chose is relatively good, as it consists of a configuration much better than the root configuration. However, of all the genetic methods, none of them outperformed *rank_best*.

In Figure 3(b), we compare three variants of the genetic algorithm. As we can see, increasing the size of initial population doesn't necessarily mean better performance. Only two variants of the genetic algorithm obtained further improvement later on. In both cases, the improvement was obtained through mutation. We observe that the most important feature of genetic algorithm is that good genes can be carried around in the population. On the other hand, *rank_best* does something similar by taking into account the cost of the parent configuration while calculating the rank of its children. Since the cost of the parent configuration is actually obtained from the optimizer's estimation, this provides more accurate information as which partitions are important.

In the optimal configuration returned by the partition advisor, the partition of 11 out of 15 tables doesn't change. Among the four tables whose partition does change, two of them (relatively small) chose to replicate themselves across all the nodes and the other two (two largest tables used in the workload) changed their partitioning keys. Among the 50 queries in the workload, 66% gained performance while the rest 34% either lose ground or had no improvement.

In Figure 4, we show the normalized time taken for each iteration for *rank_best* (figures for other methods are similar). After a few iterations, the time spent on each iteration is reduced significantly. This shows the effectiveness of our caching mechanism (described in Section 7.3). Subsequent iterations can benefit a lot from cached queries with an identical footprint.

To summarize, our experimental results validate that the partition advisor is able to recommend good candidate partitions for individual statement in the workload in a reasonable amount of time. Significant amount of time can be saved in RECOMMEND mode by employing our partition reduction technique without much plan quality degradation. We demonstrated the effectiveness of our rank-based method, which can quickly converge to an (close to) optimal solution. Our caching mechanism significantly reduces the cost of evaluating each configuration. Note that the databases we used have already been tuned by human beings over time. Nevertheless, further improvement can still be made by our tool over human experts. For applications with less reasonable initial partitions (e.g., materialized views with default partitions), the improvement would be even greater.

9 Usability Issues

Cost models in commercial systems have become quite sophisticated and have undergone comprehensive tuning. Thus, they are likely to be more accurate than any estimates from external tools. However, it's impossible to model every aspect that affects execution time, so cost estimates may not always be proportional to real execution time. We therefore give users the option to review all the partitioning recommendations given by our tool and to make necessary adjustments based on considerations that may not have been completely modeled by the tool. While human intervention is still necessary, our tool can reduce significant amount of work of database designers.

Repartitioning is an expensive process and is not expected to be run frequently. On the other hand, as observed from our experiments, new configurations may not require the complete dataset to be repartitioned. So a database designer has to balance the potential gain from the new configuration and the amount of migration work that needs to be done in order to make the appropriate decision.

Our partition advisor can be plugged in materialized view selection tools in a parallel database system to find out their optimal partition. We observe that there are possible interactions between materialized view selection and partition configuration, i.e., a partition configuration can change the materialized views selected for a workload and vice versa. We'd like to investigate such interaction in our future work.

Although this paper focuses on a shared-nothing database system, our work can be extended to a shared-disk system, where a partition is primarily defined by the partitioning keys and the concept of a nodegroup is less relevant.

10 Conclusion

In this paper, we described DB2's partition 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, our tool exploits the cost-based query optimizer to both recommend likely candidates and to evaluate complete solutions in detail. Our rank-based method converges to the optimal solution quickly with the ranking function we carefully designed. We exploit various techniques to reduce the amount of time considering alternative solutions while maintaining the quality of solutions. Our tool can recommend partitions for both base tables and materialized views. We have demonstrated through experiments the effectiveness of the tool.

We plan to investigate in the future the self-managing process of other kinds of database design problems, and the interactions among different database design aspects.

References

[ACN00] Sanjay Agrawal, et al. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of VLDB*, 2000.

[BFG+95] C. Baru, et al. DB2 parallel edition database systems: The future of high performance database systems. *IBM Systems Journal*, 34(2), 1995.

[CABK88] G. Copeland, et al. 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 SIGMOD*, 1998.

[CNW83] Stefano Ceri, et al. Distribution design of logical database schemas. *TSE*, 9(4), 1983.

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

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

[Cor00c] Oracle Corporation. Oracle 9i database. 2000.

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

[FST88] S. Finkelstein, et al. Physical database design for relational databases. *ACM Transactions of Database Systems*, 13(1), 1988.

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

[GLSW93] Peter Gassner, et al. Query Optimization in the DB2 Family. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(4), 1993.

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

[HLL94] Kien A. Hua, et al. A decomposition-based simulated annealing technique for data clustering. In *Proceedings of PODS*, 1994.

[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 SIGMOD*, 1991.

[KGV83] S. Kirkpatrick, et al. Optimization by simulated annealing. *Science*, 220(4598), 1983.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of VLDB*, 1990.

[RM93] Erhand Rahm and Rober Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *VLDB*, 1993.

[SAC+79] Patricia G. Selinger, et al. Access path selection in a relational database management system. In *Proceedings of SIGMOD*, 1979.

[SMR00] Thomas Stöhr, et al. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *Proceedings of VLDB*, 2000.

[SW85] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions of Database Systems*, 10(1), 1985.

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

[VZZ+00] Gary Valentin, et al. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of ICDE*, 2000.

[WHMZ94] Gerhard Weikum, et al. The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5), 1994.

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